IDA Pro. Руководство по дизассемблированию

Русский перевод книги об интерактивном дизассемблере IDA Pro: от основ обратной разработки и навигации по дизассемблированному коду до расширений, загрузочных и процессорных модулей, анализа уязвимостей и встроенного отладчика.

Больше крутых материалов в моём телеграм канале @grogrigs

Глава 1. Введение в дизассемблирование

Вы можете задаваться вопросом, чего ожидать от книги, целиком посвящённой IDA Pro. Хотя IDA занимает в ней центральное место, эта книга не задумывалась как руководство пользователя IDA Pro. Мы используем IDA как основной инструмент для обсуждения методов обратной разработки, полезных при анализе самого разного программного обеспечения - от уязвимых приложений до вредоносных программ.

Там, где это уместно, приводятся подробные последовательности действий в IDA. Поэтому знакомство с возможностями программы идёт не по пунктам меню, а по задачам: от первых шагов при изучении неизвестного файла до расширенного применения и настройки IDA для сложных случаев обратной разработки. Мы не пытаемся охватить абсолютно все функции IDA, но подробно рассматриваем те из них, которые чаще всего нужны на практике. Эта книга должна помочь превратить IDA в один из наиболее мощных инструментов вашего арсенала.

Прежде чем переходить к особенностям IDA, полезно разобраться в основах дизассемблирования и кратко рассмотреть другие инструменты анализа скомпилированного кода. Ни один из них не предоставляет весь набор возможностей IDA, однако каждый покрывает определённую часть её функций и помогает лучше понять отдельные стороны работы IDA. Остальная часть главы посвящена процессу дизассемблирования.

Теория дизассемблирования

Тот, кто изучал языки программирования, вероятно, встречался с делением языков по поколениям. Ниже приведено краткое напоминание.

Языки первого поколения

Это самый низкий уровень представления программы. Обычно он состоит из единиц и нулей либо использует более компактную запись, например шестнадцатеричную. Читать такой код способны главным образом специалисты, привыкшие работать с двоичными данными. На этом уровне особенно трудно отличить данные от инструкций, поскольку внешне они выглядят одинаково.

Языки первого поколения также называют машинными языками, а иногда байт-кодом. Программы на машинном языке часто называют двоичными или бинарными файлами.

Языки второго поколения

Языки второго поколения, то есть языки ассемблера, отделены от машинного кода фактически одной таблицей соответствий. Определённым битовым шаблонам или кодам операций, то есть опкодам, сопоставляются короткие запоминающиеся последовательности символов - мнемоники. Иногда мнемоники действительно помогают вспомнить назначение инструкции.

Ассемблером называют программу, которая переводит написанный на языке ассемблера исходный текст в машинный код, пригодный для исполнения.

Языки третьего поколения

Эти языки делают ещё один шаг в сторону выразительности естественного языка. В них появляются ключевые слова и конструкции, из которых программист строит программу. Обычно язык третьего поколения не зависит от платформы, хотя конкретная программа может оказаться платформенно-зависимой, если использует особенности определённой операционной системы.

К часто приводимым примерам относятся FORTRAN, COBOL, C и Java. Для перевода программ на язык ассемблера, непосредственно в машинный код или в близкую форму вроде байт-кода применяются компиляторы.

Языки четвёртого поколения

Такие языки существуют, но к теме книги отношения не имеют и далее не рассматриваются.

Что такое дизассемблирование

В традиционной модели разработки программ компиляторы, ассемблеры и компоновщики по отдельности или совместно создают исполняемую программу. Чтобы пройти этот путь в обратном направлении, то есть выполнить обратную разработку, применяют инструменты, отменяющие результаты ассемблирования и компиляции. Они называются дизассемблерами и декомпиляторами.

Дизассемблер обращает процесс ассемблирования: на вход он получает машинный код, а на выходе формирует текст на языке ассемблера. Декомпилятор стремится восстановить представление на языке высокого уровня, используя в качестве исходных данных ассемблерный или машинный код.

Обещание восстановить исходный код неизменно привлекательно на конкурентном рынке программного обеспечения, поэтому разработка практичных декомпиляторов остаётся активной областью исследований. Декомпиляция сложна по нескольким причинам.

При компиляции часть информации теряется

В машинном коде отсутствуют имена переменных и функций, а тип переменной приходится определять по способу использования данных, а не по явному объявлению. Увидев перенос 32 бит данных, аналитик должен выяснить, что именно они представляют: целое число, 32-разрядное число с плавающей точкой или 32-разрядный указатель.

Компиляция имеет отношение «многие ко многим»

Одну исходную программу можно преобразовать в ассемблерный код множеством способов. В обратную сторону один и тот же машинный код также допускает множество вариантов представления на языке высокого уровня. Поэтому исходный файл, который сначала скомпилировали, а затем сразу декомпилировали, нередко сильно отличается от первоначального текста.

Декомпилятор зависит от языка и библиотек

Если двоичный файл, созданный компилятором Delphi, обработать декомпилятором, рассчитанным на генерацию C, результат может выглядеть весьма странно. Аналогично, декомпилятор без сведений о программном интерфейсе Windows вряд ли даст полезный результат при обработке программы для Windows.

Для декомпиляции требуется почти безошибочное дизассемблирование

Ошибки и пропуски на этапе дизассемблирования почти неизбежно переходят в декомпилированный код. Один из наиболее развитых декомпиляторов, Hex-Rays, рассматривается в главе 23.

Зачем нужно дизассемблирование

Дизассемблеры обычно применяют, чтобы понять программу при отсутствии исходного кода. Наиболее распространены следующие задачи:

• анализ вредоносных программ;
• поиск уязвимостей в программах с закрытым исходным кодом;
• изучение закрытых программ для обеспечения совместимости;
• анализ сгенерированного компилятором кода для проверки производительности и корректности компилятора;
• отображение инструкций программы во время отладки.

Анализ вредоносных программ

Если речь не идёт о черве, написанном на сценарном языке, автор вредоносной программы вряд ли предоставит её исходный код. Без исходников остаётся лишь ограниченный набор способов выяснить точное поведение программы.

Два основных подхода к анализу вредоносного ПО - динамический и статический. При динамическом анализе программе позволяют выполняться в тщательно контролируемой среде, или песочнице, и записывают все наблюдаемые аспекты её поведения с помощью средств системного мониторинга. Статический анализ пытается понять поведение без запуска, изучая код программы. Для вредоносного ПО таким кодом обычно служит листинг дизассемблирования.

Анализ уязвимостей

Упрощённо аудит безопасности можно разделить на три этапа: обнаружение уязвимости, её анализ и разработка эксплойта. Эти этапы одинаковы независимо от наличия исходного кода, но при работе только с двоичным файлом трудозатраты существенно возрастают.

Сначала в программе находят потенциально эксплуатируемое состояние. Для этого часто используют динамические методы, например фаззинг[^1], хотя поиск возможен и с помощью статического анализа, обычно ценой больших усилий. После обнаружения проблемы требуется выяснить, можно ли вообще использовать её для атаки и при каких условиях.

Листинг дизассемблирования даёт точные сведения о том, как компилятор распределил переменные. Например, полезно знать, что объявленный программистом массив из 70 символов

компилятор фактически разместил в области размером 80 байт. Только дизассемблированный код позволяет точно определить порядок глобальных и локальных переменных, выбранный компилятором. Пространственные отношения между переменными нередко имеют решающее значение при разработке эксплойта. В конечном счёте совместное применение дизассемблера и отладчика позволяет создать и проверить эксплойт.

[^1]: Фаззинг - метод поиска уязвимостей, при котором для программы генерируют множество различных входных данных в надежде, что один из вариантов вызовет обнаруживаемый, анализируемый и потенциально эксплуатируемый сбой.

Совместимость программ

Если программа распространяется только в двоичном виде, конкурентам трудно создать совместимое решение или полноценную замену. Типичный пример - драйвер оборудования, выпущенный только для одной платформы. Когда производитель медленно добавляет поддержку других платформ или вовсе отказывается это делать, для создания альтернативного драйвера может потребоваться значительная работа по обратной разработке.

В таких случаях статический анализ кода зачастую является единственным выходом. Иногда исследовать приходится не только драйвер, но и встроенное программное обеспечение устройства.

Проверка компилятора

Назначение компилятора или ассемблера состоит в генерации машинного кода, поэтому для проверки соответствия результата проектным требованиям необходим хороший дизассемблер. Аналитики также могут искать дополнительные возможности оптимизации кода. С точки зрения безопасности важно убедиться, что сам компилятор не был скомпрометирован и не вставляет в создаваемые программы потайные механизмы доступа.

Представление кода в отладчике

Вероятно, чаще всего дизассемблер используется внутри отладчика для отображения исполняемых инструкций. К сожалению, встроенные в отладчики механизмы дизассемблирования нередко довольно примитивны. Обычно они не поддерживают пакетную обработку и иногда отказываются продолжать работу, если не могут определить границы функции.

Поэтому качественный отдельный дизассемблер лучше применять вместе с отладчиком: он даёт более полное представление о контексте и состоянии программы.

Как работает дизассемблирование

Рассмотрим типичную задачу дизассемблера: получить 100 КБ содержимого, отличить код от данных, перевести код на язык ассемблера и ничего не пропустить. К этому можно добавить поиск функций, распознавание таблиц переходов и определение локальных переменных, что ещё больше усложняет работу.

Для выполнения этих требований дизассемблер сочетает различные алгоритмы при разборе входного файла. Качество листинга непосредственно зависит от качества алгоритмов и их

реализации. Далее рассматриваются два основных подхода к дизассемблированию машинного кода и их недостатки. Понимание ограничений позволит распознать ситуацию, когда инструмент ошибается, и вручную улучшить результат.

Базовый алгоритм дизассемблирования

Составим простой алгоритм, который принимает машинный код и выдаёт ассемблерный текст. Он показывает основные сложности, допущения и компромиссы автоматического анализа.

Шаг 1. Найти область кода

Сначала необходимо определить область, которую следует дизассемблировать. Это не всегда очевидно: инструкции часто перемежаются данными, и важно различать их.

Исполняемый файл обычно соответствует известному формату, например Portable Executable (PE) в Windows или Executable and Linking Format (ELF) во многих Unix-подобных системах. Такие форматы содержат заголовки и другие механизмы, позволяющие найти секции кода и точки входа[^2].

[^2]: Точка входа программы - адрес инструкции, которой операционная система передаёт управление после загрузки программы в память.

Шаг 2. Декодировать инструкцию

Получив начальный адрес инструкции, дизассемблер читает значение по этому адресу или смещению в файле и по таблице сопоставляет двоичный код операции с ассемблерной мнемоникой.

В зависимости от сложности набора команд это может быть простая операция или последовательность дополнительных действий: распознавание префиксов, меняющих поведение инструкции, и определение операндов. Для архитектур с инструкциями переменной длины, таких как Intel x86, приходится читать дополнительные байты, пока инструкция не будет декодирована полностью.

Шаг 3. Сформировать строку листинга

После чтения инструкции и декодирования операндов её ассемблерное представление форматируется и добавляется в листинг. Иногда можно выбрать один из нескольких синтаксисов. Для x86 наиболее распространены синтаксисы Intel и AT&T.

> **Синтаксис x86: AT&T и Intel.** В синтаксисе AT&T перед именами регистров ставится `%`, перед непосредственными значениями - `$`, а исходный операнд находится слева, целевой справа. Прибавление четырёх к EAX записывается как `add $0x4,%eax`. Этот синтаксис используют GNU Assembler и многие инструменты GNU, включая `gcc` и `gdb`. В синтаксисе Intel префиксы регистров и констант не нужны, а порядок операндов обратный: цель слева, источник справа. Та же инструкция выглядит как `add eax,0x4`. Синтаксис Intel применяют MASM, TASM и NASM.

Шаг 4. Перейти к следующей инструкции

После вывода инструкции необходимо найти следующую и повторять процесс, пока не будет обработан весь код.

Существуют разные способы выбрать начальную точку, найти следующую инструкцию, отличить код от данных и определить момент завершения. Два основных алгоритма - линейный проход и рекурсивный спуск.

Дизассемблирование линейным проходом

Линейный проход использует очень простое правило: следующая инструкция начинается там, где заканчивается предыдущая. Главная трудность заключается в выборе начальной точки.

Обычно предполагается, что всё содержимое секции, помеченной заголовками файла как код, представляет собой машинные инструкции. Дизассемблирование начинается с первого байта секции и линейно движется вперёд до её конца. Поток управления и нелинейные инструкции, например переходы, при этом не анализируются.

Указатель отмечает начало текущей инструкции. После декодирования вычисляется её длина, по которой определяется адрес следующей. Архитектуры с командами фиксированной длины, например MIPS, анализировать несколько проще.

Преимущество линейного прохода - полное покрытие секции кода. Основной недостаток - неспособность учитывать данные, помещённые внутрь кода. В листинге 1-1 показана функция с оператором `switch`, реализованным компилятором через таблицу переходов. Таблица встроена непосредственно в функцию. Инструкция `jmp` по адресу `401250` обращается к таблице, начинающейся с `401257`, но линейный дизассемблер принимает её байты за инструкции.

```
40123f: 55                    push   ebp
401240: 8b ec                 mov    ebp,esp
401242: 33 c0                 xor    eax,eax
401244: 8b 55 08              mov    edx,DWORD PTR [ebp+8]
401247: 83 fa 0c              cmp    edx,0xc
40124a: 0f 87 90 00 00 00     ja     0x4012e0
401250: ff 24 95 57 12 40 00  jmp    DWORD PTR [edx*4+0x401257]
401257: e0 12                 loopne 0x40126b
401259: 40                    inc    eax
40125a: 00 8b 12 40 00 90     add    BYTE PTR [ebx-0x6fffbfee],cl
401260: 12 40 00              adc    al,BYTE PTR [eax]
401263: 95                    xchg   ebp,eax
401264: 12 40 00              adc    al,BYTE PTR [eax]
401267: 9a 12 40 00 a2 12 40  call   0x4012:0xa2004012
40126e: 00 aa 12 40 00 b2     add    BYTE PTR [edx-0x4dffbfee],ch
401274: 12 40 00              adc    al,BYTE PTR [eax]
401277: ba 12 40 00 c2        mov    edx,0xc2004012
40127c: 12 40 00              adc    al,BYTE PTR [eax]
40127f: ca 12 40              lret   0x4012
401282: 00 d2                 add    dl,dl
401284: 12 40 00              adc    al,BYTE PTR [eax]
401287: da 12                 ficom  DWORD PTR [edx]
401289: 40                    inc    eax
40128a: 00 8b 45 0c eb 50     add    BYTE PTR [ebx+0x50eb0c45],cl
401290: 8b 45 10              mov    eax,DWORD PTR [ebp+16]
401293: eb 4b                 jmp    0x4012e0
```

Листинг 1-1. Результат линейного дизассемблирования

Если рассматривать последовательные четырёхбайтовые группы начиная с `401257` как значения little-endian[^3], каждая из них оказывается указателем на соседний адрес перехода: `004012e0`,

`0040128b`, `00401290` и так далее. Следовательно, `loopne` по адресу `401257` вовсе не является инструкцией. Это ошибка линейного алгоритма, не сумевшего отличить встроенные данные от кода.

[^3]: В архитектуре big-endian первым сохраняется старший байт многобайтового значения, а в little-endian - младший.

Линейный проход используют механизмы дизассемблирования в GNU Debugger (`gdb`), Microsoft WinDbg и утилите `objdump`.

Дизассемблирование рекурсивным спуском

Рекурсивный спуск ищет инструкции иначе. Он ориентируется на поток управления и решает, нужно ли дизассемблировать некоторую область, исходя из наличия ссылок на неё из других инструкций. Чтобы понять алгоритм, полезно классифицировать инструкции по влиянию на указатель команд процессора.

Инструкции последовательного выполнения

Такие инструкции передают управление непосредственно следующей инструкции. К ним относятся простые арифметические команды вроде `add`, операции передачи данных между регистрами и памятью вроде `mov`, а также работа со стеком через `push` и `pop`. Здесь дизассемблирование продолжается так же, как при линейном проходе.

Условные переходы

Условная инструкция, например x86 `jnz`, создаёт два возможных пути. Если условие истинно, выполняется переход и указатель команд получает адрес цели. Если условие ложно, выполнение линейно продолжается со следующей инструкции.

При статическом анализе обычно невозможно заранее узнать результат условия, поэтому рекурсивный алгоритм исследует оба пути. Сначала он продолжает последовательное дизассемблирование, а адрес цели перехода помещает в список отложенных адресов.

Безусловные переходы

Безусловный переход не соответствует линейной модели. Выполнение может продолжиться только по одному адресу, но он не обязан находиться сразу после инструкции перехода. Более того, после безусловного перехода вообще может не быть инструкции, поэтому автоматически дизассемблировать следующие байты нельзя.

Рекурсивный дизассемблер пытается определить цель перехода и добавить её в список ещё не исследованных адресов. Однако косвенные переходы создают проблему. Если целевой адрес зависит от значения времени выполнения, статически определить его может быть невозможно. Например, цель `jmp eax` хранится в регистре EAX только во время работы программы. При статическом анализе значение регистра неизвестно, поэтому неизвестно и место продолжения дизассемблирования.

Вызовы функций

Инструкция вызова похожа на безусловный переход, включая проблему с косвенным вариантом `call eax`. Однако после завершения функции обычно ожидается возврат к инструкции, следующей непосредственно за `call`. В этом смысле вызов похож на условный переход, поскольку порождает два направления анализа. Адрес вызываемой функции откладывается, а разбор продолжается со следующей инструкции.

Рекурсивный спуск может ошибиться, если функция возвращает управление не так, как ожидается. В следующем намеренно необычном примере функция `foo` увеличивает сохранённый адрес возврата на единицу.

```
foo proc near
  FF 04 24        inc  dword ptr [esp]  ; увеличить сохранённый адрес возврата
  C3              retn
foo endp
; ------------------------------------
bar:
  E8 F7 FF FF FF  call foo
  05 89 45 F8 90  add  eax, 90F84589h   ; ошибочная интерпретация
```

Управление фактически не попадает на показанную после `call` инструкцию `add`. Правильный листинг выглядит так:

```
foo proc near
  FF 04 24        inc  dword ptr [esp]
  C3              retn
foo endp
; ------------------------------------
bar:
  E8 F7 FF FF FF  call foo
  05              db   5                ; бывший первый байт add
  89 45 F8        mov  [ebp-8], eax
  90              nop
```

Здесь реальный поток управления возвращается к `mov`. Линейный дизассемблер также разобрал бы этот фрагмент неправильно, хотя и по другой причине.

Инструкции возврата

Иногда у рекурсивного алгоритма заканчиваются пути для продолжения. Инструкция возврата из функции, например x86 `ret`, не содержит сведений о следующем адресе. Во время исполнения адрес берётся с вершины стека, но статический дизассемблер не располагает стеком времени выполнения.

Поэтому текущий путь анализа заканчивается. Дизассемблер извлекает один из ранее отложенных адресов и продолжает с него. Именно повторная обработка списка отложенных ветвей придаёт алгоритму рекурсивный характер.

Главное преимущество рекурсивного спуска - гораздо более надёжное различение кода и данных. Поскольку решение опирается на поток управления, данные существенно реже ошибочно принимаются за инструкции. Основной недостаток - невозможность непосредственно проследить косвенные вызовы и переходы, использующие таблицы указателей. Эвристики, распознающие указатели на код, позволяют обеспечить хорошее покрытие и высокую точность.

Листинг 1-2 показывает, как рекурсивный дизассемблер обрабатывает тот же оператор `switch`, который ранее сбил с толку линейный алгоритм.

```
0040123F  push ebp
00401240  mov  ebp, esp
00401242  xor  eax, eax
```

```
00401244  mov  edx, [ebp+arg_0]
00401247  cmp  edx, 0Ch
0040124A  ja   loc_4012E0
00401250  jmp  ds:off_401257[edx*4]

00401257 off_401257:
00401257  dd offset loc_4012E0
00401257  dd offset loc_40128B
00401257  dd offset loc_401290
00401257  dd offset loc_401295
00401257  dd offset loc_40129A
00401257  dd offset loc_4012A2
00401257  dd offset loc_4012AA
00401257  dd offset loc_4012B2
00401257  dd offset loc_4012BA
00401257  dd offset loc_4012C2
00401257  dd offset loc_4012CA
00401257  dd offset loc_4012D2
00401257  dd offset loc_4012DA

0040128B loc_40128B:
0040128B  mov  eax, [ebp+arg_4]
0040128E  jmp  short loc_4012E0
```

Листинг 1-2. Результат дизассемблирования рекурсивным спуском

Таблица адресов переходов здесь распознана и оформлена как данные. IDA Pro - наиболее известный пример дизассемблера, использующего рекурсивный спуск. Понимание этого процесса помогает заметить случаи, когда IDA выдаёт неоптимальный результат, и выбрать способ его исправления.

Итоги

Обязательно ли глубоко знать алгоритмы дизассемблирования, чтобы пользоваться дизассемблером? Нет. Полезно ли это? Безусловно. При обратной разработке не стоит тратить время на борьбу с собственными инструментами.

Одно из преимуществ IDA состоит в том, что, в отличие от большинства дизассемблеров, она позволяет направлять анализ и отменять автоматические решения. В результате можно получить точный листинг, превосходящий полностью автоматический вывод других средств.

В следующей главе рассматриваются инструменты, полезные в различных задачах обратной разработки. Хотя они не являются частью IDA, многие из них влияли на неё и сами испытывали её влияние. Знакомство с ними также объясняет разнообразие информационных представлений в интерфейсе IDA.

Глава 2. Инструменты обратной разработки и дизассемблирования

Теперь, когда мы знакомы с основами дизассемблирования, перед подробным изучением IDA Pro полезно рассмотреть другие инструменты обратной разработки двоичных файлов. Многие из них появились раньше IDA и по-прежнему удобны для быстрой первичной оценки файла или независимой проверки результатов IDA.

IDA объединяет многие возможности таких программ в единой среде. Отладчики в этой главе не рассматриваются: им посвящены главы 24, 25 и 26.

Средства классификации

При первой встрече с неизвестным файлом хочется ответить на простой вопрос: что это такое? Главное правило - никогда не определять тип файла только по расширению. Это же можно считать вторым, третьим и четвёртым правилом. Расширение легко изменить, а иногда оно вообще отсутствует.

file

Команда `file` входит в большинство Unix-подобных систем и доступна в Windows в составе Cygwin или MinGW. Она пытается определить тип файла по характерным полям содержимого. Иногда достаточно известной строки, например `#!/bin/sh` для сценария оболочки или `<html>` для HTML-документа. Для двоичных данных программа проверяет соответствие известному формату и ищет специальные значения, называемые магическими числами.

Примеры начальных байтов распространённых форматов:

```
Исполняемый файл Windows PE
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..............
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......

Изображение JPEG
00000000  FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 60  ......JFIF.....`
00000010  00 60 00 00 FF DB 00 43 00 0A 07 07 08 07 06 0A  .`.....C........

Файл класса Java
00000000  CA FE BA BE 00 00 00 32 00 98 0A 00 2E 00 3E 08  .......2......>.
00000010  00 3F 09 00 40 00 41 08 00 42 0A 00 43 00 44 0A  .?..@.A..B..C.D.
```

Правила распознавания хранятся в базе `magic`. Её расположение зависит от системы; часто встречаются `/usr/share/file/magic`, `/usr/share/misc/magic` и `/etc/magic`.

> **Среда Cygwin.** Cygwin предоставляет в Windows оболочку и набор программ в стиле Linux. При установке можно выбрать компиляторы `gcc` и `g++`, интерпретаторы Perl, Python и Ruby, сетевые инструменты `nc` и `ssh` и множество других пакетов. Многие программы для Linux после этого можно собрать и запускать в Windows.

Иногда `file` различает варианты одного формата. В следующем примере она определяет архитектуру ELF, способ компоновки и наличие таблицы символов:

```
$ file ch2_ex_*
ch2_ex.exe:                 MS-DOS executable PE for MS Windows (console), Intel 80386 32-bit
ch2_ex_upx.exe:             MS-DOS executable PE for MS Windows (console), Intel 80386 32-bit, UPX compressed
```

```
ch2_ex_freebsd:              ELF 32-bit LSB executable, Intel 80386, dynamically linked, not stripped
ch2_ex_freebsd_static:       ELF 32-bit LSB executable, Intel 80386, statically linked, not stripped
ch2_ex_freebsd_static_strip: ELF 32-bit LSB executable, Intel 80386, statically linked, stripped
ch2_ex_linux:                ELF 32-bit LSB executable, Intel 80386, dynamically linked, not stripped
ch2_ex_linux_static:         ELF 32-bit LSB executable, Intel 80386, statically linked, not stripped
ch2_ex_linux_static_strip:   ELF 32-bit LSB executable, Intel 80386, statically linked, stripped
```

Удаление символов из двоичного файла называют stripping. Объектные файлы получают символы при компиляции: часть нужна компоновщику для разрешения межфайловых ссылок, другая часть помогает отладчику. После компоновки многие символы больше не требуются. Их можно удалить параметрами компоновщика или отдельной утилитой `strip`. Файл станет меньше, но его поведение не изменится.

`file` и похожие инструменты могут ошибаться. Если заменить первые четыре байта произвольного файла на `CA FE BA BE`, он будет принят за класс Java. Текст из двух символов `MZ` может быть назван исполняемым файлом MS-DOS. Поэтому результат любого инструмента следует сопоставлять с выводом других программ и ручным анализом.

PE Tools

PE Tools - набор средств анализа запущенных процессов и исполняемых файлов Windows. Главное окно показывает список процессов и предоставляет доступ к остальным функциям.

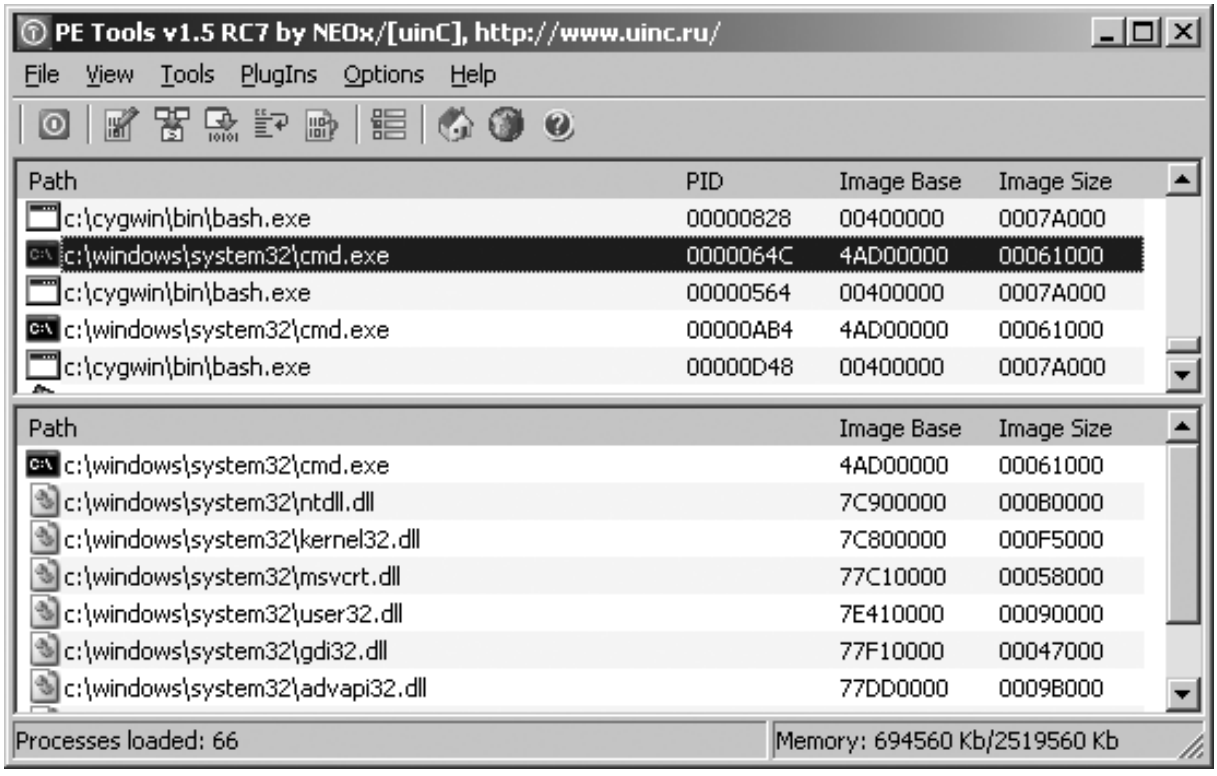

Рис. 2-1. Утилита PE Tools

Из списка процессов можно сохранить образ памяти процесса или запустить PE Sniffer, чтобы определить компилятор и известные средства обфускации. Аналогичные действия для файлов на диске доступны через меню Tools. Встроенный PE Editor показывает и позволяет изменять поля заголовка PE. Это особенно важно при восстановлении корректного PE-файла из обфусцированного варианта.

Обфускацией называют намеренное усложнение понимания истинного смысла. Применительно к исполняемому файлу это попытка скрыть реальное поведение программы. Так защищают

собственные алгоритмы или маскируют вредоносные действия. Почти все вредоносные программы используют ту или иную обфускацию. Её инструменты, методы и влияние на обратную разработку подробно рассматриваются в главе 21.

PEiD

PEiD - ещё один инструмент Windows. Его основные задачи - определить компилятор PE-файла и распознать программу, использованную для обфускации. На рисунке PEiD определяет ASPack в одном из вариантов червя Gaobot.

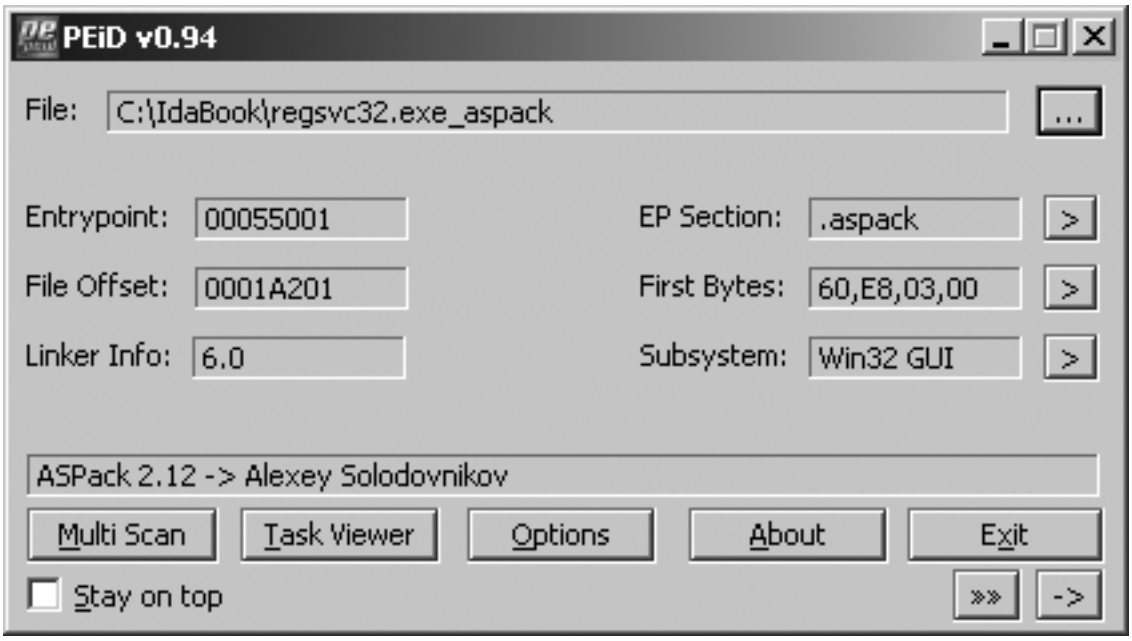

Рис. 2-2. Утилита PEiD

Дополнительные функции PEiD частично совпадают с PE Tools: сводка заголовков PE, сведения о процессах и базовое дизассемблирование.

Средства получения сводной информации

После первичной классификации требуются более специализированные программы. Они знают внутреннее устройство конкретных форматов и извлекают определённые сведения.

nm

При компиляции исходных файлов в объектные компилятор должен сохранить местоположение глобальных, то есть внешних, символов. Компоновщик использует их для разрешения ссылок при создании программы. Если символы не удалены, многие из них переходят в итоговый исполняемый файл. Назначение `nm` - вывести символы объектного файла.

Для промежуточного файла `.o` стандартный вывод содержит имена функций и глобальных переменных:

```
$ gcc -c ch2_example.c
$ nm ch2_example.o
         U __stderrp
```

```
         U exit
         U fprintf
00000038 T get_max
00000000 t hidden
00000088 T main
00000000 D my_initialized_global
00000004 C my_unitialized_global
         U printf
         U rand
         U scanf
         U srand
         U time
00000010 T usage
```

Буква рядом с символом описывает его тип:

• `U` - неопределённый символ, обычно внешняя ссылка;
• `T` - глобальный символ в секции кода, обычно имя функции;
• `t` - локальный символ в секции кода, например статическая функция C;
• `D` - инициализированные данные;
• `C` - неинициализированные данные.

Прописные буквы обозначают глобальные символы, строчные - локальные. Полный перечень приведён в справочной странице `nm`.

После компоновки часть символов получает виртуальные адреса, а компоновщик может добавить собственные имена:

```
$ gcc -o ch2_example ch2_example.c
$ nm ch2_example
         U exit
         U fprintf
080485c0 t frame_dummy
08048644 T get_max
0804860c t hidden
08048694 T main
0804997c D my_initialized_global
08049a9c B my_unitialized_global
08049a80 b object.2
08049978 d p.0
         U printf
         U rand
         U scanf
         U srand
         U time
0804861c T usage
```

Здесь некоторые имена получили адреса, появился `frame_dummy`, а типы части символов изменились. Неопределённые имена остались внешними: в динамически скомпонованной программе их реализации находятся в общей библиотеке C.

ldd

Ссылки на библиотечные функции разрешаются статической или динамической компоновкой. Одна программа может использовать оба способа.

При статической компоновке объектные файлы приложения объединяются с копией необходимого библиотечного кода. Во время запуска искать библиотеку не требуется. Вызовы немного быстрее, а распространять программу проще, поскольку она меньше зависит от окружения. Недостатки - больший размер и необходимость заново компоновать программу после обновления библиотеки.

Для аналитика статическая компоновка создаёт дополнительные трудности: нелегко понять, какие библиотеки вошли в файл и какие функции являются библиотечными. Этому посвящена глава 12.

При динамической компоновке библиотечный код не копируется. В исполняемый файл добавляются ссылки на файлы `.so` или `.dll`, поэтому размер обычно меньше. Замена единственной общей библиотеки обновляет сразу все использующие её программы. Недостаток - обязательное наличие совместимых библиотек в целевой системе.

Разница в размере может быть существенной:

```
$ gcc -o ch2_example_dynamic ch2_example.c
$ gcc -o ch2_example_static ch2_example.c --static
$ ls -l ch2_example_*
-rwxr-xr-x  1 root wheel    6017 Sep 26 11:24 ch2_example_dynamic
-rwxr-xr-x  1 root wheel  167987 Sep 26 11:23 ch2_example_static
```

Динамический файл обязан перечислять свои зависимости. Утилита `ldd` показывает необходимые библиотеки:

```
$ ldd /usr/local/sbin/httpd
/usr/local/sbin/httpd:
        libm.so.4 => /lib/libm.so.4 (0x280c5000)
        libaprutil-1.so.2 => /usr/local/lib/libaprutil-1.so.2 (0x280db000)
        libexpat.so.6 => /usr/local/lib/libexpat.so.6 (0x280ef000)
        libiconv.so.3 => /usr/local/lib/libiconv.so.3 (0x2810d000)
        libapr-1.so.2 => /usr/local/lib/libapr-1.so.2 (0x281fa000)
        libcrypt.so.3 => /lib/libcrypt.so.3 (0x2821a000)
        libpthread.so.2 => /lib/libpthread.so.2 (0x28232000)
        libc.so.6 => /lib/libc.so.6 (0x28257000)
```

`ldd` доступна в Linux и BSD. В OS X аналогичную задачу выполняет `otool -L filename`, а в Windows - `dumpbin /dependents filename` из Visual Studio.

objdump

В отличие от узкоспециализированной `ldd`, `objdump` умеет показывать множество сведений об объектных файлах. Более тридцати параметров командной строки позволяют получить, среди прочего:

• заголовки секций и сводку по каждой секции;
• приватные заголовки, схему размещения в памяти и список библиотек;
• встроенную отладочную информацию;
• таблицу символов, похожую на вывод `nm`;
• листинг дизассемблирования.

`objdump` линейно дизассемблирует секции, помеченные как код. Для x86 доступен синтаксис AT&T или Intel. Результат можно сохранить как текстовый статический листинг, но по большому текстовому файлу трудно перемещаться и ещё труднее согласованно вносить изменения.

Утилита входит в GNU binutils и доступна в Linux, FreeBSD и Cygwin. Для разбора объектных форматов она использует библиотеку Binary File Descriptor (`libbfd`), поддерживающую, в частности, ELF и PE. Отдельная утилита `readelf` предоставляет сходные функции для ELF, но не зависит от `libbfd`.

otool

`otool` можно считать аналогом `objdump` для OS X. Она разбирает двоичные файлы Mach-O, показывает заголовки и таблицы символов, дизассемблирует секцию кода и перечисляет динамические зависимости:

```
$ file osx_example
```

```
osx_example: Mach-O executable ppc
$ otool -L osx_example
osx_example:
  /usr/lib/libstdc++.6.dylib (compatibility version 7.0.0, current version 7.4.0)
  /usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 1.0.0)
  /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.1.5)
```

dumpbin

`dumpbin` входит в Microsoft Visual Studio и выводит широкий набор сведений о PE-файлах. Например, зависимости стандартного калькулятора Windows:

```
$ dumpbin /dependents calc.exe
Microsoft (R) COFF/PE Dumper Version 8.00.50727.762

Dump of file calc.exe
File Type: EXECUTABLE IMAGE

  Image has the following dependencies:
    SHELL32.dll
    msvcrt.dll
    ADVAPI32.dll
    KERNEL32.dll
    GDI32.dll
    USER32.dll
```

Другие параметры извлекают символы, импортированные и экспортированные функции, содержимое секций и дизассемблированный код.

c++filt

Языку с перегрузкой функций нужен способ различать варианты с одинаковым исходным именем:

```
void demo(void);
void demo(int x);
void demo(double x);
void demo(int x, double y);
void demo(double x, int y);
void demo(char* str);
```

Объектный файл обычно не может содержать несколько символов с одним именем. Компилятор создаёт уникальное имя, включая в него сведения о типах аргументов. Этот процесс называется декорированием или искажением имён, name mangling.

```
$ g++ -o cpp_test cpp_test.cpp
$ nm cpp_test | grep demo
0804843c T _Z4demoPc
08048400 T _Z4demod
08048428 T _Z4demodi
080483fa T _Z4demoi
08048414 T _Z4demoid
080483f4 T _Z4demov
```

Стандарт C++ не задаёт единую схему, поэтому она зависит от компилятора. `c++filt` распознаёт декорированные имена и восстанавливает читаемую форму. Неизвестное слово выводится без изменений.

```
$ nm cpp_test | grep demo | c++filt
0804843c T demo(char*)
08048400 T demo(double)
08048428 T demo(double, int)
080483fa T demo(int)
08048414 T demo(int, double)
```

```
080483f4 T demo()
```

Декорированное имя содержит сведения, которых обычный вывод `nm` не показывает. В сложных случаях из него можно узнать имя класса и соглашение о вызове функции, что очень полезно при обратной разработке.

Средства глубокого исследования

До сих пор рассматривались либо поверхностные классификаторы, либо программы, хорошо знающие конкретную структуру файла. Следующие инструменты извлекают определённый вид информации независимо от формата.

strings

Иногда нужен простой ответ на вопрос: есть ли в файле строки? Условно строкой можно считать непрерывную последовательность печатных символов, дополнив определение минимальной длиной и набором символов. Например, можно искать последовательности не короче четырёх печатных ASCII-символов. Структура файла при этом не важна: поиск одинаково применим к ELF и документу Word.

`strings` специально предназначена для извлечения текстовых последовательностей. Стандартные настройки, то есть не менее четырёх 7-битных ASCII-символов, могут дать такой результат:

```
$ strings ch2_example
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
exit
srand
puts
time
printf
stderr
scanf
usage: ch2_example [max]
A simple guessing game!
Please guess a number between 1 and %d.
Invalid input, quitting!
Congratulations, you got it in %d attempt(s)!
Sorry too low, please try again
Sorry too high, please try again
```

Здесь видны как предполагаемые сообщения программы, так и имена функций и библиотек. Нельзя делать вывод о поведении только на основании `strings`: наличие строки в файле ещё не означает, что программа когда-либо её использует.

При работе с `strings` нужно помнить следующее:

• по умолчанию в исполняемом файле могут сканироваться только загружаемые инициализированные секции; параметр `-a` заставляет исследовать весь файл;
• стандартный вывод не указывает положение строки; параметр `-t` добавляет смещение в файле;
• для других кодировок параметр `-e` позволяет искать, например, широкие 16-битные символы Unicode.

Потоковые дизассемблеры

`dumpbin`, `objdump` и `otool` создают статические листинги PE, ELF и Mach-O соответственно, но не работают с произвольным блоком байтов. Неизвестный формат требует средства, которому можно явно указать начальное смещение.

Для x86 такими потоковыми дизассемблерами являются `ndisasm` из NASM и diStorm. Ниже `ndisasm` обрабатывает 62-байтовый шелл-код, созданный Metasploit:

```
$ ndisasm -u fs
00000000  31D2          xor edx,edx
00000002  52            push edx
00000003  89E5          mov ebp,esp
00000005  6A07          push byte +0x7
00000007  5B            pop ebx
00000008  6A10          push byte +0x10
0000000A  54            push esp
0000000B  55            push ebp
0000000C  52            push edx
0000000D  89E1          mov ecx,esp
0000000F  FF01          inc dword [ecx]
00000011  6A66          push byte +0x66
00000013  58            pop eax
00000014  CD80          int 0x80
00000016  66817D02115C  cmp word [ebp+0x2],0x5c11
0000001C  75F1          jnz 0xf
0000001E  5B            pop ebx
0000001F  6A02          push byte +0x2
00000021  59            pop ecx
00000022  B03F          mov al,0x3f
00000024  CD80          int 0x80
00000026  49            dec ecx
00000027  79F9          jns 0x22
00000029  52            push edx
0000002A  682F2F7368    push dword 0x68732f2f
0000002F  682F62696E    push dword 0x6e69622f
00000034  89E3          mov ebx,esp
00000036  52            push edx
00000037  53            push ebx
00000038  89E1          mov ecx,esp
0000003A  B00B          mov al,0xb
0000003C  CD80          int 0x80
```

Потоковый подход полезен, например, при анализе сетевой атаки, когда пакет содержит шелл-код. Можно дизассемблировать только вредоносную полезную нагрузку. Другой пример - образ ROM без документации по структуре: одни области содержат данные, другие код, и аналитик выбирает предполагаемые участки кода вручную.

Итоги

Рассмотренные программы не обязательно являются лучшими в своих категориях, но они широко доступны и представляют основные классы инструментов обратной разработки двоичных файлов. Именно потребности, которые они закрывают, во многом определили развитие IDA.

Знакомство с этими средствами помогает понять интерфейс IDA и назначение множества доступных в ней информационных представлений.

Глава 3. Общие сведения об IDA Pro

Interactive Disassembler Professional, более известный как IDA Pro или просто IDA, является продуктом бельгийской компании Hex-Rays. Главный разработчик IDA - Ильфак Гильфанов. Программа начиналась более десяти лет назад как консольное приложение для MS-DOS, и это происхождение до сих пор заметно в её интерфейсе. Для всех поддерживаемых платформ поставлялись не только графические, но и консольные версии, унаследовавшие стиль первых выпусков DOS.

В основе IDA лежит рекурсивный спуск, дополненный большим количеством логики и эвристик. Они помогают находить код, пропущенный основным алгоритмом. IDA старается не просто отличить код от данных, но и определить точный тип представленных данных.

Хотя пользователь видит язык ассемблера, одна из главных целей IDA - приблизить результат к исходному коду. Программа добавляет сведения о типах, выводит предполагаемые имена функций и переменных, уменьшает количество необработанных шестнадцатеричных значений и заменяет их понятными символическими обозначениями.

Отношение Hex-Rays к пиратству

IDA является основным продуктом Hex-Rays, поэтому компания серьёзно относится к несанкционированному распространению. В прошлом появление пиратских версий напрямую связывали со снижением продаж. Прежний издатель DataRescue даже публиковал имена нарушителей в специальном «зале позора». Для ограничения пиратства и соблюдения лицензий IDA применяет несколько механизмов.

Каждая копия получает уникальную водяную метку, связывающую её с покупателем. Если такая копия появляется на пиратском ресурсе, Hex-Rays может определить первоначального владельца и отказаться продавать ему продукты в дальнейшем.

Версия IDA для Windows также ищет другие экземпляры с тем же лицензионным ключом в локальной сети. При запуске она передаёт широковещательный UDP-пакет на порт `23945` и ожидает ответы. Количество обнаруженных копий сравнивается с числом разрешённых рабочих мест. Если предел превышен, IDA не запускается. При этом одна лицензия допускает несколько одновременно открытых экземпляров на одном компьютере.

Третий механизм основан на файле ключа `ida.key`, привязанном к покупателю. При запуске IDA ищет действительный ключ и немедленно завершает работу, если его нет. Этот же файл подтверждает право на обновление. По сути, `ida.key` является квитанцией о покупке, поэтому его необходимо надёжно хранить.

Получение IDA Pro

IDA не является свободным программным обеспечением. Для знакомства с базовыми возможностями Hex-Rays распространяла бесплатную версию с ограниченной функциональностью, которая отставала от текущего выпуска. На момент написания книги это была урезанная IDA 5.0 при актуальной версии 6.1. Бесплатный вариант подробнее рассматривается в приложении A.

Кроме того, существовала демонстрационная версия актуального выпуска с дополнительными ограничениями. Она позволяла оценить продукт и поддержку перед покупкой.

Варианты IDA

Начиная с версии 6.0 графическая и консольная IDA выпускалась для Windows, Linux и OS X. Кроссплатформенная библиотека Qt обеспечивала единый графический интерфейс.

По функциональности IDA Pro предлагалась в стандартном и расширенном вариантах. Главным отличием было количество поддерживаемых архитектур. На момент издания стандартная версия поддерживала более 30 семейств процессоров, расширенная - более 50. Только в расширенный вариант входили, в частности, x64, AMD64, MIPS, PowerPC и SPARC.

Лицензии IDA

Предлагались два типа лицензий:

• именная лицензия связывалась с конкретным пользователем и позволяла ему устанавливать IDA на несколько компьютеров;
• компьютерная лицензия связывалась с одним компьютером, за которым могли по очереди работать разные пользователи.

Именная лицензия не разрешала другим людям пользоваться установленными копиями. При одной лицензии IDA могла работать только на одном из компьютеров владельца в каждый конкретный момент.

В отличие от лицензий многих закрытых продуктов, лицензия IDA прямо разрешала пользователю выполнять обратную разработку самой IDA.

Покупка IDA

До версии 6.0 покупатель получал графическую версию для Windows и консольные версии для Windows, Linux и OS X. С версии 6.0 требовалось выбрать операционную систему. В комплект IDA 6.x входили консольная и Qt-версия только для выбранной платформы. Лицензии для дополнительных систем продавались дешевле полной лицензии.

IDA можно было приобрести у авторизованного поставщика или напрямую у Hex-Rays по электронной почте либо факсу. Поставка выполнялась на компакт-диске или в виде загрузки и включала год поддержки и обновлений. На диске находились установщик, SDK и дополнительные утилиты. При электронной доставке пользователь обычно получал только установочный комплект и скачивал остальные компоненты отдельно.

Hex-Rays могла ограничивать продажи в отдельных странах из-за уровня пиратства. Компания также вела список пользователей, нарушивших лицензию, и могла отказаться работать с ними или их работодателями.

Обновление IDA

Меню Help содержало команду проверки обновлений. IDA также предупреждала о приближении конца периода поддержки, используя дату из ключа. Для обновления обычно требовалось отправить `ida.key` в Hex-Rays. Компания проверяла ключ и сообщала порядок получения новой версии. Владельцам просроченных ключей могли предложить льготную цену.

Файл ключа следует защищать: посторонний человек способен запросить полагающееся владельцу обновление и тем самым помешать законному обновлению.

Перед установкой новой версии рекомендуется сделать резервную копию существующей IDA или выбрать совершенно другой каталог. Иначе можно потерять изменённые конфигурационные файлы. Настройки придётся повторно перенести в новую версию. Пользовательские плагины также нужно переместить, пересобрать или заменить совместимыми выпусками. Плагины рассматриваются в главе 17.

Ресурсы поддержки IDA

При возникновении вопросов пользователю были доступны несколько источников.

Встроенная справка

Система помощи из меню IDA описывала главным образом интерфейс и встроенный язык сценариев. В ней не было полноценной документации SDK и готовых ответов на многие практические вопросы.

Страница поддержки и форумы Hex-Rays

Официальная страница содержала ссылки на ресурсы и форумы для лицензированных пользователей. Ильфак и другие основные разработчики Hex-Rays регулярно отвечали там. Форумы служили и неофициальной поддержкой SDK, поскольку опытные пользователи делились практическими решениями.

На вопросы об SDK нередко отвечали советом изучать заголовочные файлы. Обычная покупка IDA не включала официальную поддержку SDK. Для введения в тему авторы рекомендуют материал Стива Микаллефа «IDA Plug-in Writing in C/C++».

OpenRCE.org

Сообщество OpenRCE публиковало статьи о нестандартном применении IDA и поддерживало активные форумы. Опытные пользователи помогали решать практически любые проблемы, связанные с программой.

Форумы RCE

Форумы Reverse Code Engineering содержали множество обсуждений IDA Pro, а также более широкий набор тем по инструментам и методам обратной разработки двоичных файлов.

IDA Palace

Сайт IDA Palace, несмотря на неоднократные переезды, собирал статьи, сценарии и плагины, расширяющие IDA.

Блог Ильфака

В блоге Ильфака публиковались решения задач дизассемблирования, отладки и анализа вредоносных программ. Другие сотрудники Hex-Rays рассказывали о новых и разрабатываемых возможностях IDA.

Установка IDA

На установочном диске каталоги `utilities` и `sdk` содержали дополнительные утилиты и комплект разработки. Они рассматриваются далее в книге. В корне находился установочный файл: обычная программа установки для Windows или архив `.tar.gz` для Linux и OS X.

Установка в Windows

Установщик Windows запрашивал пароль, полученный вместе с диском или письмом при электронной доставке. Единственным существенным выбором был каталог установки. Далее он обозначается как `<IDADIR>`.

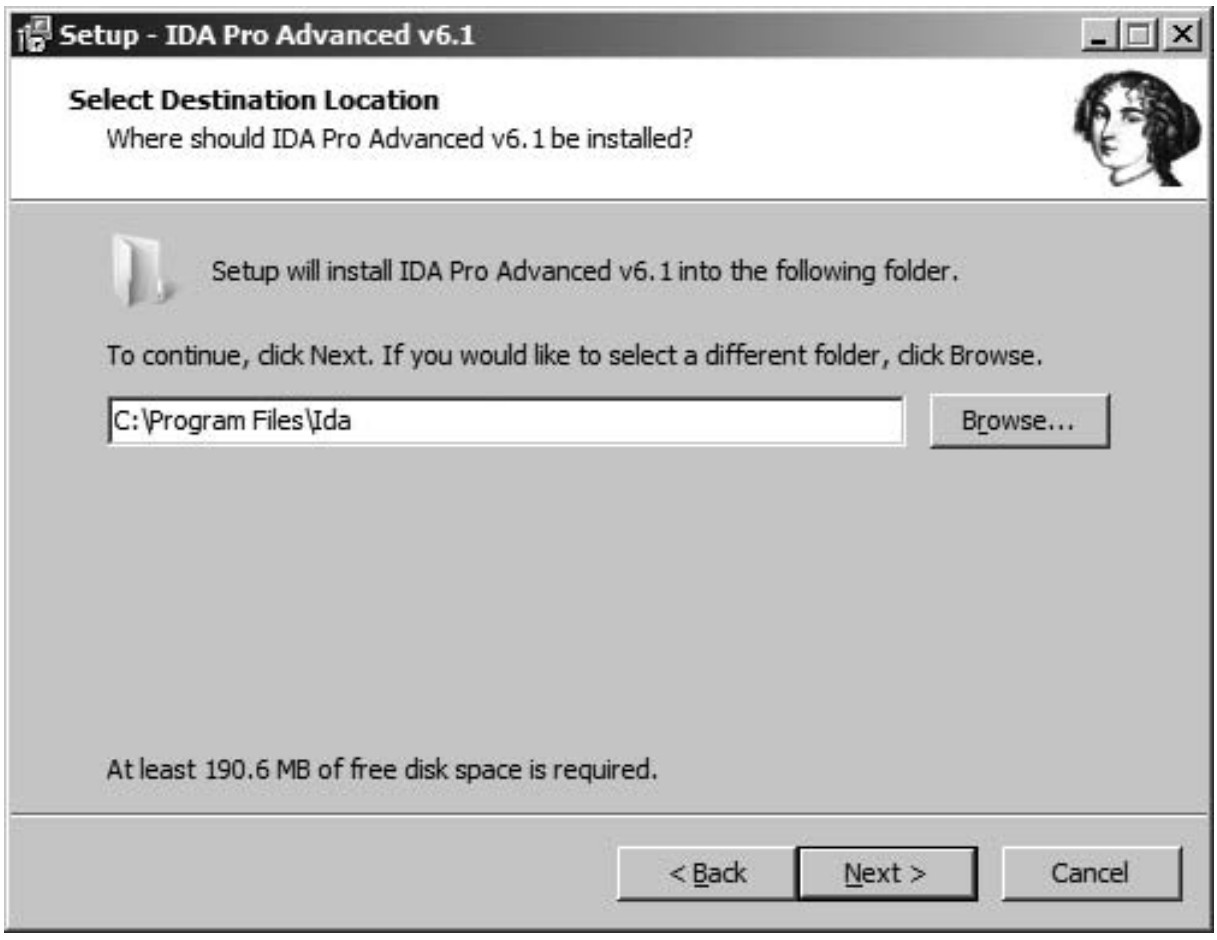

Рис. 3-1. Выбор каталога установки

В `<IDADIR>` находился `ida.key` и несколько исполняемых файлов:

• `idag.exe` - исходная графическая версия для Windows, объявленная устаревшей после перехода на Qt;
• `idaq.exe` - графическая Qt-версия для Windows начиная с IDA 6.0;
• `idaw.exe` - текстовая версия для Windows.

С переходом на Qt в IDA 6.0 нативная Windows-версия `idag.exe` была выведена из развития и, согласно книге, должна была исчезнуть из поставки версии 6.2.

Установка в OS X и Linux

Архив требовалось распаковать в выбранное место:

```
# Linux
tar -xvzf ida61l.tgz

# OS X
tar -xvzf ida61m.tgz
```

В обоих случаях создавался верхний каталог `ida` со всеми необходимыми файлами. Графическая программа называлась `idaq`, консольная - `idal`. Консольный интерфейс был похож на версию Windows.

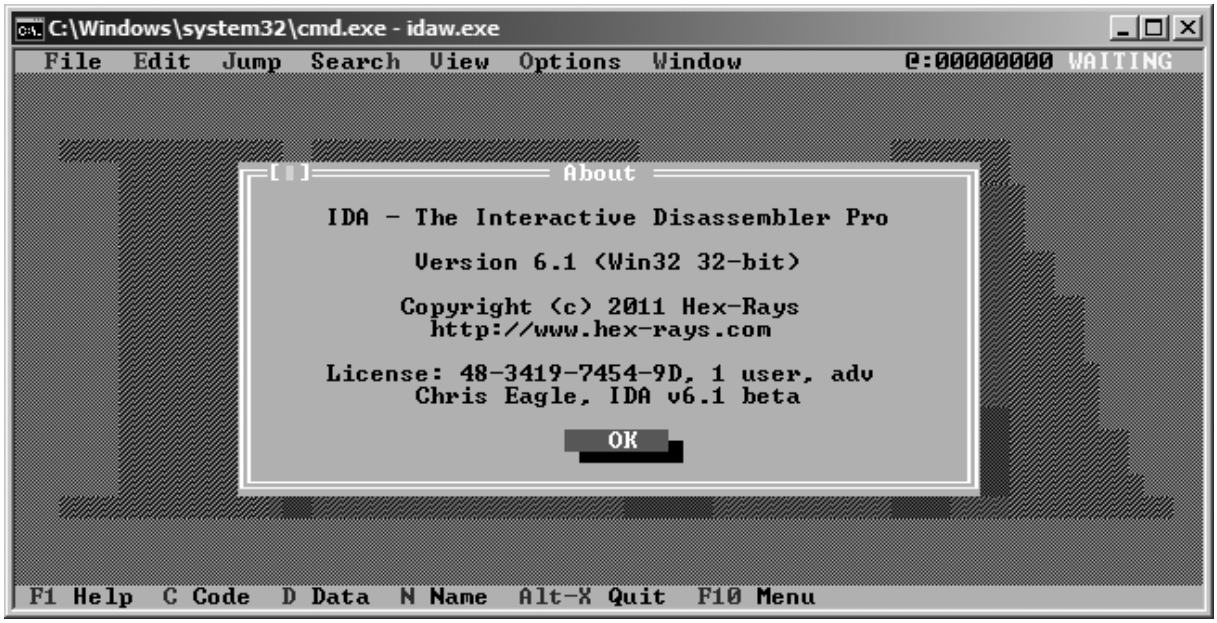

Рис. 3-2. Консольная версия IDA Pro

В Linux полезно проверить через `ldd`, доступны ли все необходимые общие библиотеки. Плагин IDAPython того времени ожидал Python 2.6, поэтому иногда требовалось обновить Python или создать символические ссылки.

IDA и SELinux

При включённом SELinux загрузка процессорного модуля могла завершиться сообщением `cannot enable executable stack as shared object`. Для конкретного модуля проблему исправляла команда:

```
execstack -c <IDADIR>/procs/pc.ilx
```

32-разрядная и 64-разрядная IDA

В расширенной версии присутствовали пары файлов вроде `idag.exe` и `idag64.exe` либо `idaq` и `idaq64`. Вариант с суффиксом `64` мог дизассемблировать 64-разрядный код, хотя сами программы IDA того времени оставались 32-разрядными.

Поэтому в 64-разрядной системе вспомогательные компоненты должны были быть доступны в 32-разрядном виде. Например, для IDAPython в 64-разрядной Linux требовалась 32-разрядная сборка Python.

Структура каталогов IDA

Знать все каталоги перед первым запуском необязательно, но эта структура становится важной при расширенной настройке. В Windows и Linux перечисленные каталоги находились в `<IDADIR>`, а в OS X - в `<IDADIR>/idaq.app/Contents/MacOS`.

cfg

Конфигурационные файлы, включая основной `ida.cfg`, настройки графического интерфейса `idagui.cfg` и текстового интерфейса `idatui.cfg`. Настройка рассматривается в главе 11.

idc

Основные файлы встроенного языка сценариев IDC. Сценариям посвящена глава 15.

ids

Файлы символов IDS, описывающие содержимое общих библиотек. Они перечисляют экспортируемые элементы, типы и количество параметров функций, типы возвращаемых значений и соглашения о вызове.

loaders

Модули загрузки, которые распознают и разбирают известные форматы, например PE и ELF. Они рассматриваются в главе 18.

plugins

Плагины, добавляющие новые возможности, часто созданные самими пользователями. Архитектуре плагинов посвящена глава 17.

procs

Процессорные модули поддерживаемых архитектур. Они преобразуют машинные инструкции в ассемблерный текст, показываемый IDA. Их устройство разбирается в главе 19.

sig

Сигнатуры известного кода для сопоставления с образцами. Благодаря им IDA распознаёт библиотечные функции и экономит время анализа. Сигнатуры создаются технологией Fast Library Identification and Recognition Technology, или FLIRT, из главы 12.

til

Библиотеки типов, описывающие структуры данных различных компиляторов и библиотек. Их настройка рассматривается в главе 13.

Особенности интерфейса IDA

Наследие MS-DOS до сих пор заметно в IDA. И текстовый, и графический интерфейс активно используют горячие клавиши. Это удобно, но приводит к неожиданным действиям, если пользователь думает, что вводит текст. Почти каждое нажатие может оказаться командой.

Практически все данные вводятся через диалоговые окна. Если вы хотите что-либо ввести, сначала убедитесь, что на экране открыт подходящий диалог. Исключение - редактирование шестнадцатеричных данных в окне Hex View.

Наконец, в описываемой версии IDA нет отмены. Если случайная клавиша запустила команду, искать Undo бесполезно. Истории команд, по которой можно было бы понять выполненное действие, также нет.

Итоги

После знакомства с происхождением, вариантами поставки и устройством установки можно переходить к практической работе. Следующие главы показывают базовый анализ файлов, объясняют информационные окна IDA и учат изменять представление данных, чтобы лучше понимать поведение программы.

Глава 4. Начало работы с IDA

Пора перейти к практическому использованию IDA. Остальная часть книги посвящена функциям программы и их применению в обратной разработке. В этой главе рассматриваются запуск IDA, открытие двоичного файла, создание базы данных и основные элементы интерфейса.

Для единообразия примеры далее показаны в графической Qt-версии для Windows, кроме случаев, когда задача требует другой платформы, например при отладке в Linux.

Запуск IDA

После запуска кратко появляется заставка со сведениями о лицензии, затем диалог с тремя способами перейти к рабочему столу.

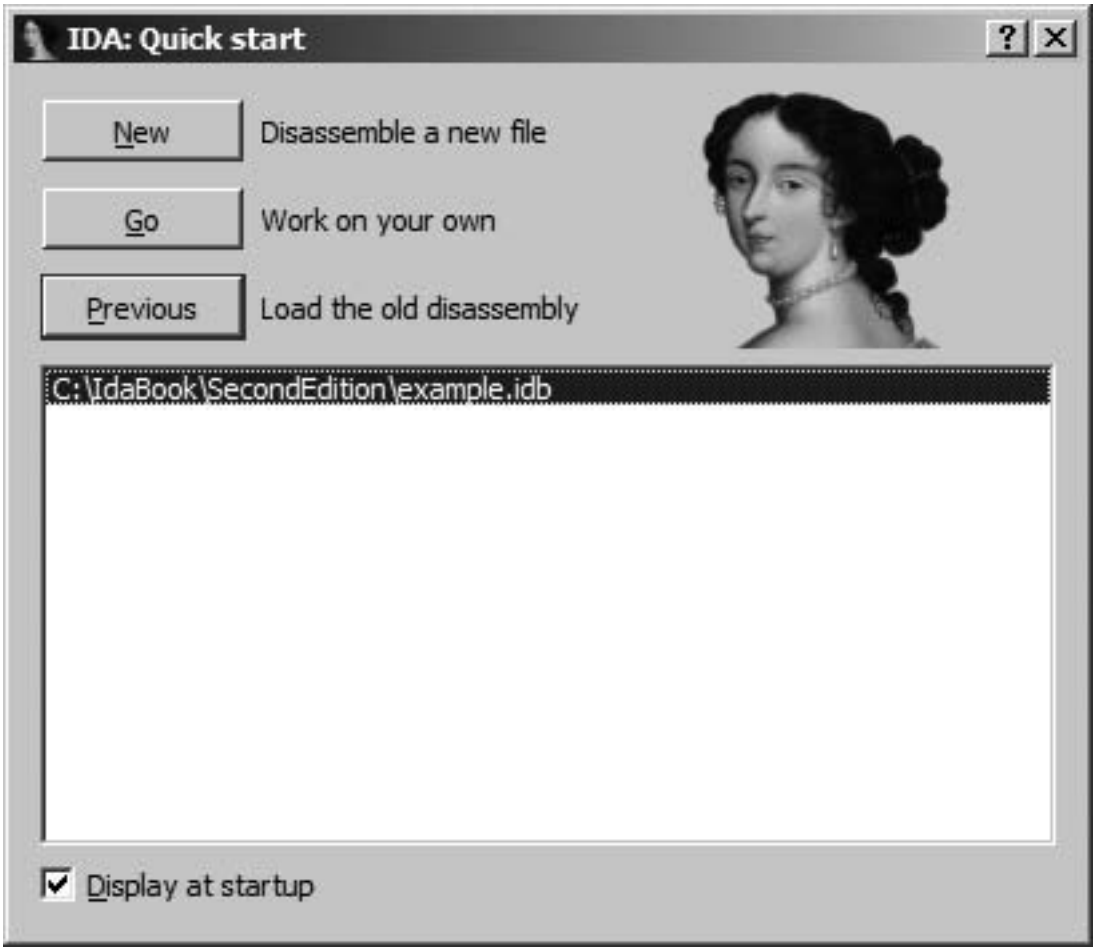

Рис. 4-1. Запуск IDA

Если снять флажок Display at startup, в дальнейшем IDA будет сразу открывать пустое рабочее пространство, как после нажатия Go. Вернуть приветственный диалог можно, установив параметр реестра `DisplayWelcome` в `1`, либо командой Windows > Reset hidden messages, которая восстанавливает все ранее скрытые сообщения.

В Windows настройки IDA хранились в ветви `HKEY_CURRENT_USER\Software\Hex-Rays\IDA`. В других системах использовался двоичный файл `$HOME/.idapro/ida.reg`, который неудобно редактировать вручную.

New

Кнопка New открывает стандартный выбор файла. После выбора появляются дополнительные диалоги с параметрами анализа, затем файл загружается, исследуется и отображается.

Go

Кнопка Go завершает процедуру загрузки и открывает пустой рабочий стол. Файл можно перетащить в окно IDA или открыть через File > Open.

В диалоге выбора по умолчанию действует фильтр известных расширений. Его иногда нужно изменить на All Files, особенно в Unix-подобных системах, где исполняемые файлы часто не имеют расширения. IDA пытается автоматически определить тип, но в следующем диалоге следует проверить выбранный загрузчик.

Previous

Кнопка Previous открывает файл из списка последних баз. Список берётся из подраздела History в реестре Windows либо из `ida.reg`. Изначально в нём не более десяти записей; через `idagui.cfg` или `idatui.cfg` предел можно увеличить до 100. Это самый удобный способ продолжить недавнюю работу.

Загрузка файла в IDA

При открытии нового файла IDA показывает диалог загрузки. Вверху перечислены подходящие форматы, то есть загрузочные модули из каталога `loaders`, которые распознали содержимое.

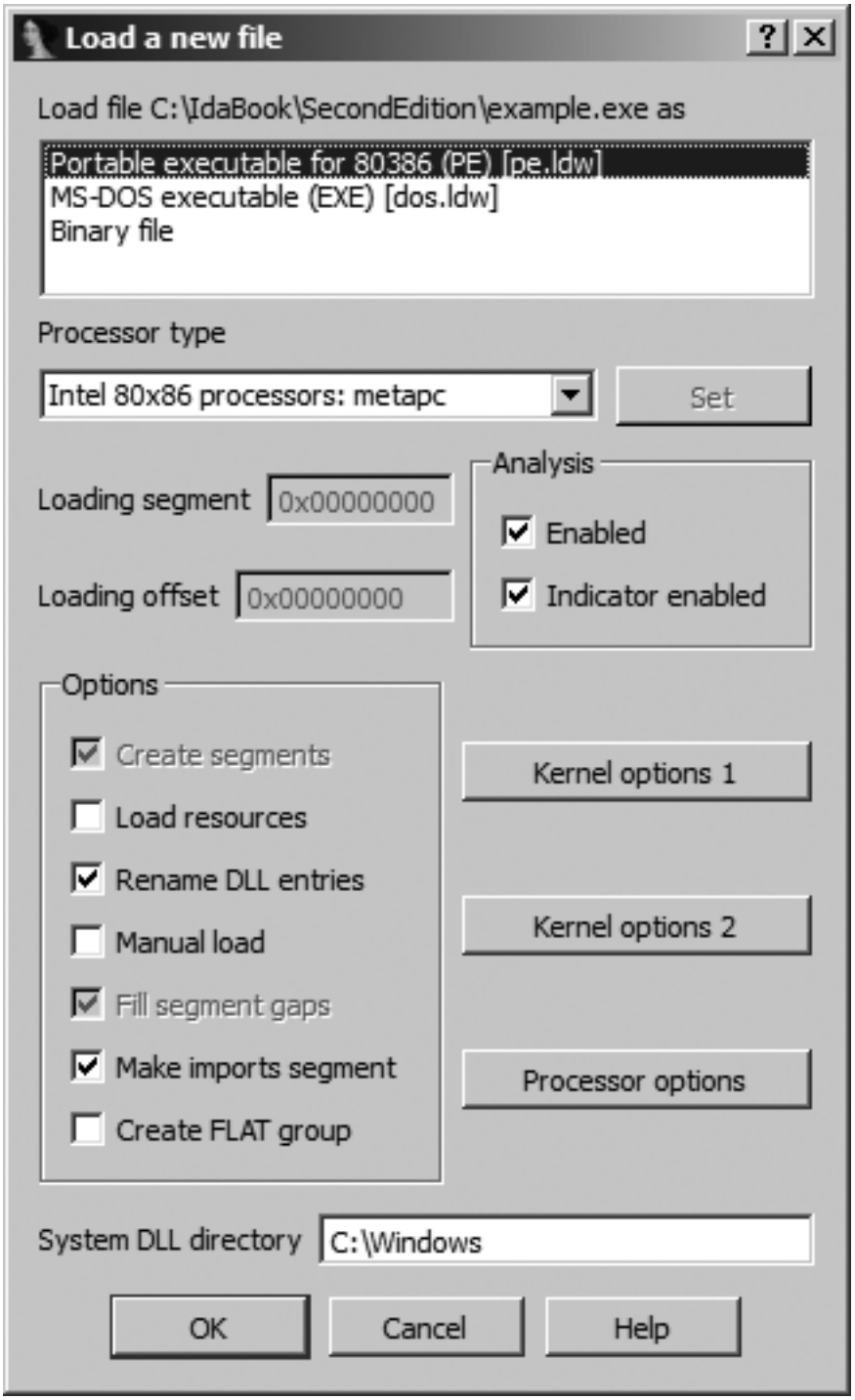

Рис. 4-2. Диалог Load a New File

На рисунке файл одновременно распознали загрузчики Windows PE `pe.ldw` и MS-DOS EXE `dos.ldw`. Это естественно, поскольку PE расширяет формат DOS EXE. Пункт Binary File присутствует всегда и позволяет загрузить любое содержимое на самом низком уровне. Если нет оснований сомневаться, обычно разумно оставить первый автоматически выбранный вариант.

Когда доступен только Binary File, ни один специализированный загрузчик не распознал файл. В таком случае пользователь должен самостоятельно выбрать правильную архитектуру процессора.

Список Processor Type определяет процессорный модуль из каталога `procs`. Обычно IDA выбирает его по заголовку исполняемого файла. Если это невозможно, архитектуру необходимо указать вручную.

Поля Loading Segment и Loading Offset доступны для двоичного файла с процессором x86. Загрузчик не знает схему памяти, поэтому сегмент и смещение объединяются в базовый адрес содержимого. Позднее базу можно изменить через Edit > Segments > Rebase Program.

Кнопки Kernel Options настраивают анализ, дополняющий рекурсивный спуск. Почти всегда стандартные параметры дают лучший результат. Processor Options открывает настройки конкретного процессорного модуля, если тот их предоставляет. Остальные флажки уточняют процедуру загрузки и обычно также не требуют изменения. Особые случаи рассматриваются в главе 21.

Использование загрузчика Binary File

При загрузке необработанного двоичного содержимого значительная часть работы ложится на аналитика. Отсутствуют заголовки, по которым можно автоматически создать секции, выбрать адреса и найти код. Такой режим нужен, например, для образов ROM или полезной нагрузки эксплойта, извлечённой из сетевого пакета либо журнала.

Для x86 IDA спрашивает, следует ли интерпретировать код как 16-разрядный или 32-разрядный. Аналогичный выбор возможен для ARM и MIPS.

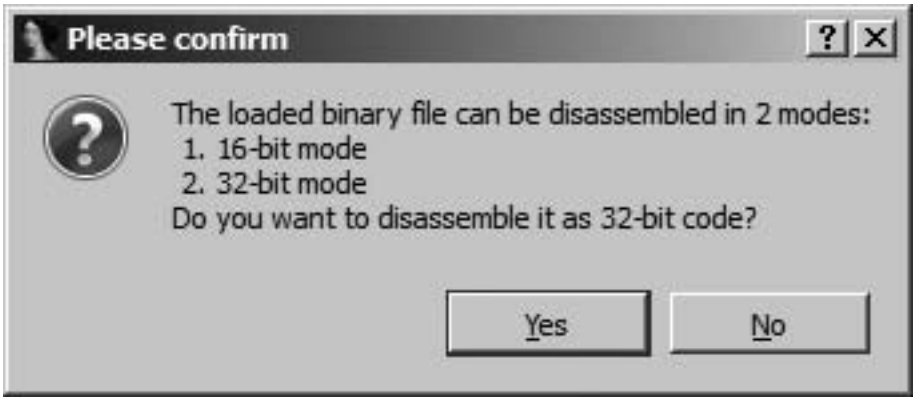

Рис. 4-3. Выбор разрядности x86

Для x86 базовый адрес задаётся в исходном диалоге. Для других архитектур открывается окно организации памяти. В нём можно создать диапазоны RAM и ROM, выбрать часть исходного файла и указать адрес, по которому она должна быть отображена.

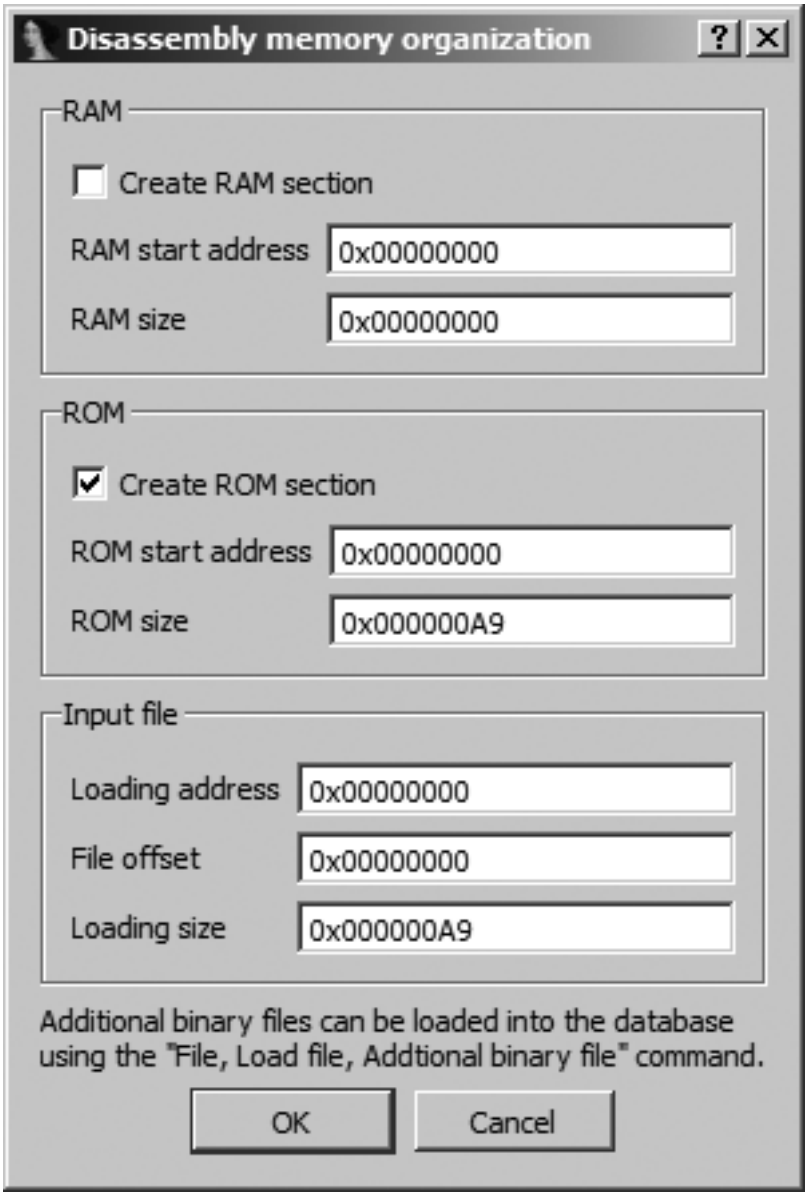

Рис. 4-4. Диалог Memory Organization

Последнее сообщение напоминает, что IDA не может отличить код от данных без заголовка. Пользователь должен перейти к предполагаемой точке входа и клавишей C преобразовать байты по этому адресу в код. До указания хотя бы одного фрагмента кода начальное дизассемблирование двоичного файла не выполняется.

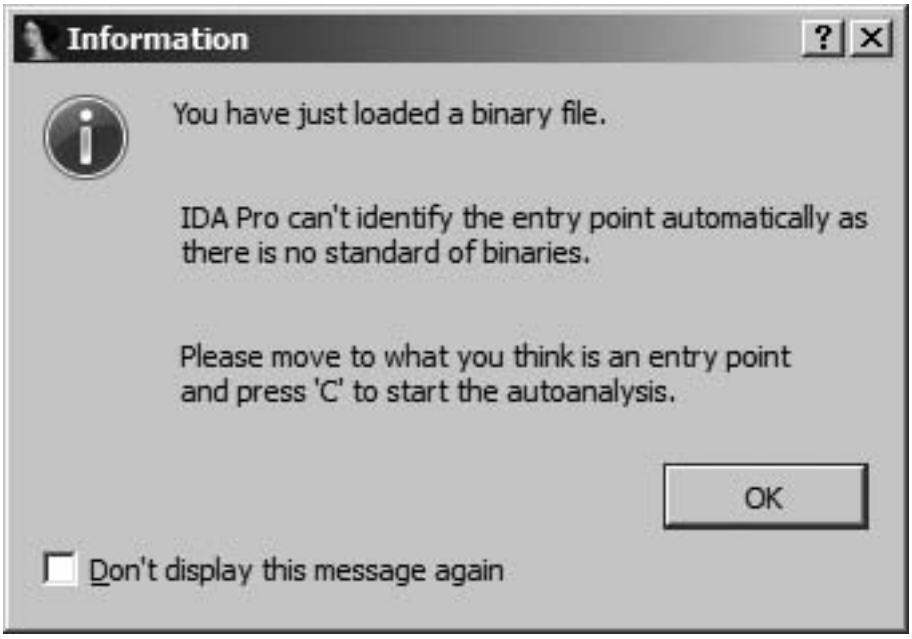

Рис. 4-5. Завершение загрузки двоичного файла

Файлы базы данных IDA

После подтверждения параметров IDA загружает исполняемый файл в память, анализирует важные области и создаёт базу данных. В описываемой версии она во время работы состояла из четырёх файлов с именем исходной программы:

- `.id0` - содержимое базы типа B-tree;
- `.id1` - флаги, описывающие каждый байт программы;
- `.nam` - индекс именованных адресов, показываемых в окне Names;
- `.til` - локальные определения типов этой базы.

Форматы являются внутренними и плохо подходят для изменения вне IDA. При нормальном закрытии четыре компонента архивируются, при желании сжимаются, в единый файл IDB. Обычно выражение «база IDA» означает именно `.idb`.

Несжатая база может быть примерно в десять раз больше исходного файла. После корректного закрытия промежуточные `.id0`, `.id1`, `.nam` и `.til` исчезают. Если они остались, IDA, вероятно, завершилась аварийно, а данные могут быть повреждены.

Предупреждения загрузчика

Загрузчику иногда требуется дополнительный ввод. Если PE связан с отладочной информацией PDB, IDA предлагает найти соответствующий файл в локальном хранилище или на сервере символов Microsoft.

Обфусцированные программы часто нарушают спецификацию формата. Если таблица импорта PE выглядит повреждённой, загрузчик предупреждает, что файл мог быть упакован или изменён для затруднения анализа, и предлагает повторить загрузку без создания секции импорта. Практические примеры приводятся в главе 21.

После создания базы исходный исполняемый файл больше не нужен, если только аналитик не собирается запускать его во встроенном отладчике. Это удобно при работе с вредоносной программой: коллегам можно передать базу без самого опасного образца.

По своей сути IDA является приложением базы данных. Загрузчик заполняет базу сведениями из программы, а окна интерфейса показывают разные представления этих данных. Изменения пользователя сохраняются в базе, но не меняют исходный исполняемый файл. Сила IDA заключается в средствах анализа и преобразования базы.

Создание базы

Сначала выбранный загрузочный модуль читает файл, разбирает заголовки, создаёт секции кода и данных и находит точки входа. Его поведение напоминает системный загрузчик: по заголовкам определяется виртуальная схема памяти, которая переносится в базу.

Затем механизм дизассемблирования передаёт процессорному модулю адреса. Модуль определяет тип и длину инструкции, а также возможные следующие адреса. Когда список инструкций сформирован, второй проход создаёт их текстовое ассемблерное представление.

После дизассемблирования автоматический анализ добавляет полезные сведения.

Определение компилятора

Знание компилятора помогает понять соглашения о вызове и предполагаемые библиотеки. IDA пытается распознать компилятор, затем ищет типовые последовательности служебного кода. Известные библиотечные функции выделяются цветом, сокращая объём ручного анализа.

Аргументы и локальные переменные

В каждой найденной функции IDA отслеживает изменение указателя стека, распознаёт обращения к данным в стеке и восстанавливает устройство кадра. Переменные получают автоматически сгенерированные имена в зависимости от роли локального значения или аргумента.

Сведения о типах

Зная распространённые библиотечные функции и их параметры, IDA добавляет комментарии в местах передачи аргументов. Это экономит время, которое иначе ушло бы на поиск сведений в документации API.

Закрытие базы

При закрытии IDA или переходе к другой базе появляется диалог сохранения.

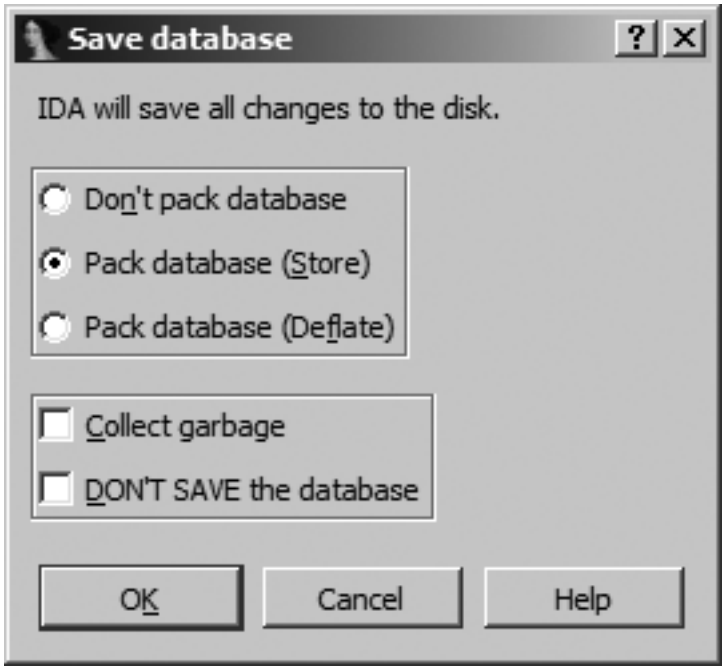

Рис. 4-6. Диалог Save Database

При первом сохранении расширение исходного файла заменяется на `.idb`: `example.exe` превращается в `example.idb`. Если расширения нет, `.idb` добавляется к имени.

Доступны следующие варианты:

• **Don't pack database** сохраняет изменения только в четырёх промежуточных файлах и не создаёт IDB. Для обычного закрытия вариант не рекомендуется.
• **Pack database (Store)** объединяет четыре файла в IDB без сжатия, без подтверждения заменяет предыдущий IDB и удаляет компоненты.
• **Pack database (Deflate)** выполняет то же самое со сжатием.
• **Collect garbage** удаляет неиспользуемые страницы базы. Вместе с Deflate создаёт минимальный IDB, но обычно нужен только при дефиците места.
• **DON'T SAVE the database** удаляет промежуточные файлы, оставляя прежний IDB неизменным. Это единственный способ отбросить все изменения после последнего сохранения и ближайший аналог отката.

Повторное открытие базы

Существующая база открывается как обычный файл и загружается гораздо быстрее, потому что повторный анализ не нужен. IDA также восстанавливает состояние рабочего стола на момент закрытия.

После аварии остаются промежуточные файлы, потенциально находящиеся в несогласованном состоянии. Если база ранее сохранялась, рядом могут находиться последний исправный IDB и более новые промежуточные данные. IDA предлагает восстановить упакованную версию или продолжить с неупакованной.

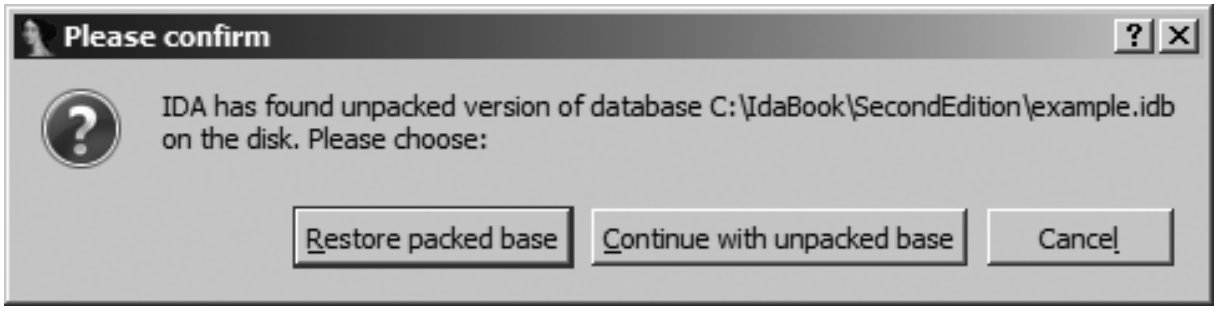

Рис. 4-7. Диалог восстановления базы

Продолжение с неупакованной базой не гарантирует возврат последних изменений. При обнаружении несогласованности IDA предлагает ремонт, но сама рекомендует использовать сохранённую копию.

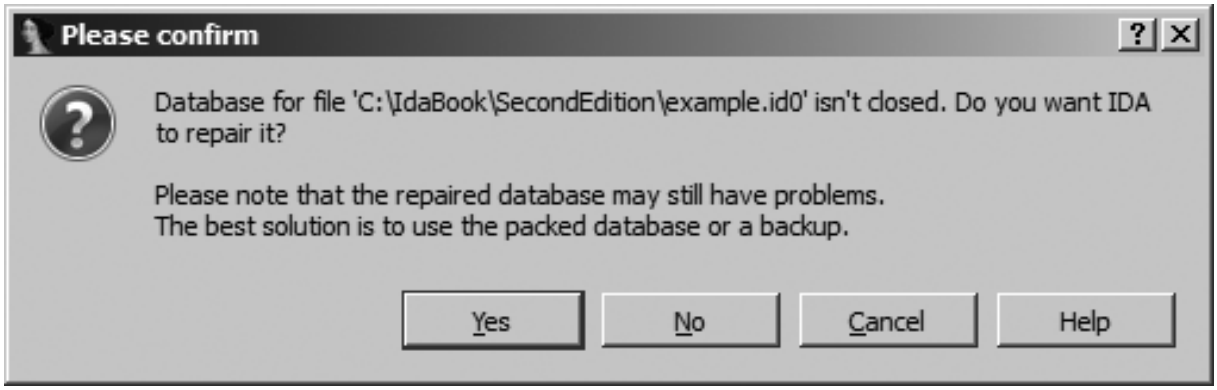

Рис. 4-8. Диалог ремонта базы

Если база ещё ни разу не сохранялась, после аварии останутся только компоненты. Диалог ремонта появится при следующей попытке открыть исходную программу.

Рабочий стол IDA

На рабочем столе IDA аналитик проводит много времени, поэтому важно понимать его основные области.

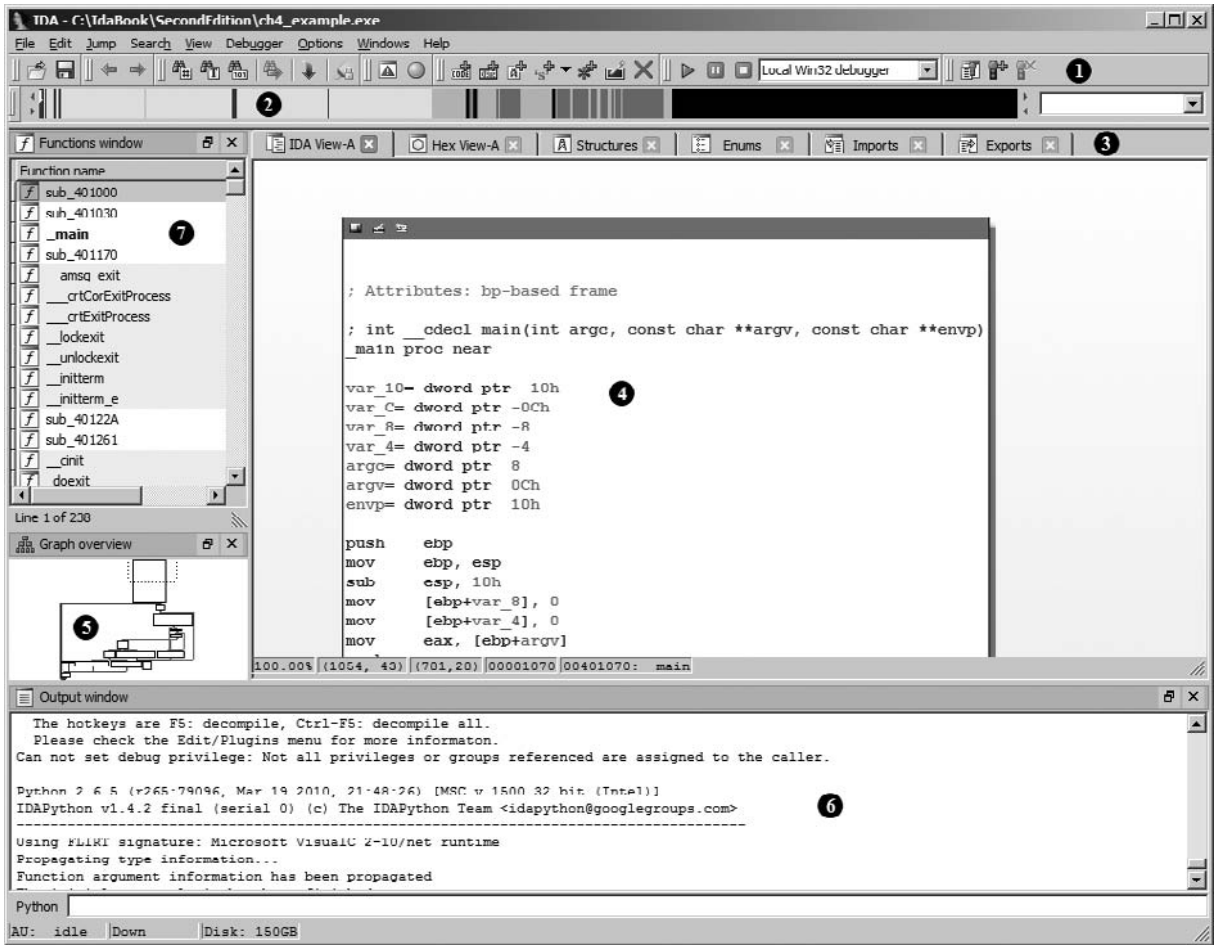

Рис. 4-9. Рабочий стол IDA

1. Панели инструментов

В верхней части расположены кнопки часто используемых команд. Панели включаются через View > Toolbars и перетаскиваются в удобное место. На рисунке показан базовый режим с одним рядом. Advanced mode добавляет три полных ряда кнопок.

2. Навигационная полоса

Горизонтальная цветная полоса даёт линейный обзор адресного пространства файла. Контекстное меню позволяет менять масштаб. Цвета обозначают код, данные, библиотечные функции, внешние символы и другие типы содержимого.

Небольшой индикатор текущей позиции, по умолчанию жёлтый, соответствует адресу в окне дизассемблирования. При наведении показывается описание области, а щелчок переносит основной вид к выбранному адресу. Цвета меняются через Options > Colors. Полосу можно отделить от рабочего стола.

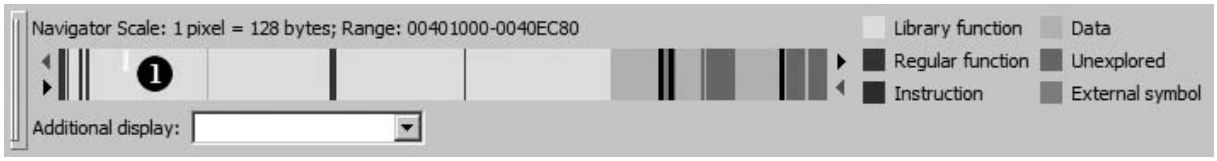

Рис. 4-10. Навигатор обзора

3. Вкладки представлений

Каждое открытое представление базы имеет вкладку. Основная работа выполняется именно через такие окна. По умолчанию видны IDA View, Functions и Graph Overview. Дополнительные окна открываются и восстанавливаются через View > Open Subviews.

4. Окно дизассемблирования

Главное представление работает в двух режимах: граф функции и линейный листинг. Граф показывает блоки одной функции и связи между ними. Пробел переключает режимы, когда активно IDA View. Чтобы сделать листинг стандартным видом, нужно снять Use graph view by default в Options > General на вкладке Graph.

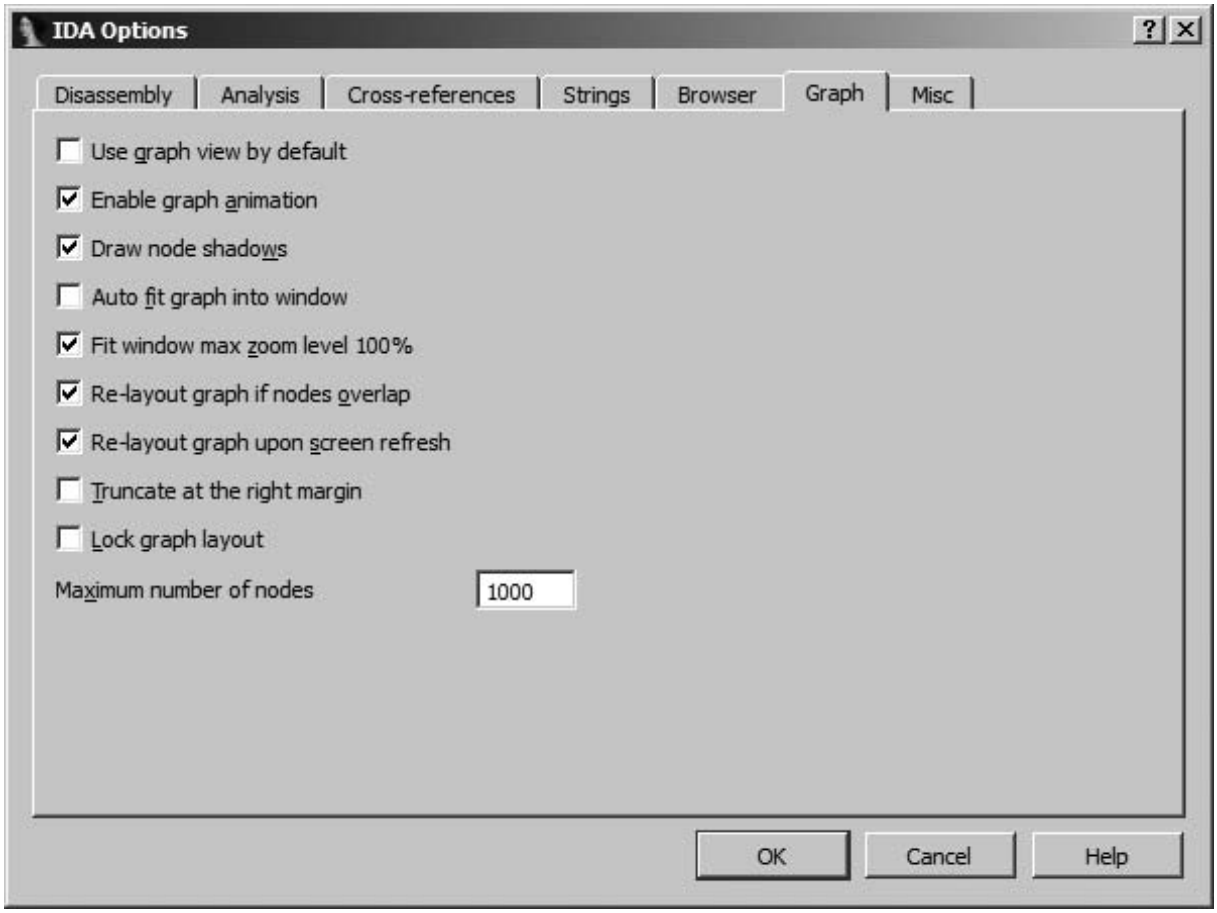

Рис. 4-11. Параметры графа IDA

5. Graph Overview

Большой граф функции редко помещается целиком. Уменьшенный обзор показывает общую структуру, а пунктирный прямоугольник отмечает текущую видимую область. Щелчок по обзору перемещает основной граф.

6. Output

Здесь IDA выводит ход анализа, информационные сообщения и ошибки операций. Это приблизительный аналог консоли.

7. Functions

Окно Functions перечисляет распознанные функции и подробно рассматривается в главе 5.

Поведение рабочего стола при начальном анализе

Во время автоанализа нового файла интерфейс активно меняется:

• сообщения о ходе работы появляются в Output;
• создаётся начальный листинг;
• заполняется и обновляется Functions;
• навигационная полоса меняет цвета по мере распознавания кода, данных, функций и библиотечных фрагментов;
• индикатор позиции движется по анализируемым областям.

Типичный журнал выглядит так:

```
Loading file 'C:\IdaBook\ch4_example.exe' into database...
Detected file format: Portable executable for 80386 (PE)
  0. Creating a new segment (00401000-0040C000) ... OK
  1. Creating a new segment (0040C000-0040E000) ... OK
  2. Creating a new segment (0040E000-00411000) ... OK
Reading imports directory...
Plan FLIRT signature: Microsoft VisualC 2-10/net runtime
Assuming __cdecl calling convention by default
main() function at 401070, named "_main"
Marking typical code sequences...
File is successfully loaded into the database.
IDA is analysing the input file...
You may start to explore the input file right now.
Using FLIRT signature: Microsoft VisualC 2-10/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

Сообщение `You may start to explore the input file right now` означает, что данных уже достаточно для навигации. Однако до завершения анализа лучше ничего не менять: поздний этап может перезаписать изменение или пользователь способен помешать алгоритму.

После `The initial autoanalysis has been finished` автоматические изменения прекращаются, и базу можно безопасно редактировать.

Советы по организации рабочего стола

• Чем больше экранного пространства выделено IDA, тем удобнее работа; полезен большой монитор или несколько мониторов.
• View > Open Subviews возвращает случайно закрытое представление.
• Windows > Reset Desktop быстро восстанавливает исходную раскладку.
• Windows > Save Desktop сохраняет удобную раскладку, Windows > Load Desktop загружает её.

• Шрифт можно изменить только для окна дизассемблирования через Options > Font.

Сообщение об ошибках

Как и любая сложная программа, IDA иногда содержит ошибки. Поддержка Hex-Rays традиционно отвечала быстро, нередко лично от Ильфака.

Сообщить об ошибке можно было по адресу `support@hex-rays.com` или на форуме Bug Reports. Перед обращением следует убедиться, что проблема воспроизводится, и подготовить базу, на которой она проявляется.

Обычная лицензия не включала поддержку SDK. С ошибкой стороннего плагина нужно обращаться к его автору. При разработке собственного плагина остаются форумы пользователей и помощь сообщества.

Итоги

Знание рабочего пространства упрощает дальнейшую работу: обратная разработка и без того сложна, чтобы дополнительно бороться с интерфейсом. Параметры начальной загрузки и автоанализ создают основу всей последующей работы.

Для простого файла автоматического результата иногда достаточно. Дальнейшие главы объясняют, почему IDA называется интерактивной: они рассматривают основные представления, ситуации их применения и способы уточнять базу вручную.

Глава 5. Представления данных IDA

После завершения начального анализа управление переходит к пользователю. Лучший способ освоиться - изучить вкладки, которые IDA заполнила сведениями о двоичном файле. Чем увереннее вы ориентируетесь в интерфейсе, тем быстрее и эффективнее идёт обратная разработка.

Перед подробным обзором окон стоит запомнить несколько правил:

• в рассматриваемой версии IDA нет отмены, поэтому последствия случайного нажатия приходится исправлять вручную;
• почти у каждого действия есть пункт меню, горячая клавиша и кнопка панели, причём панели и сочетания клавиш настраиваются;
• контекстное меню по правой кнопке предлагает самые распространённые операции для текущего места, хотя и не перечисляет все возможности.

Основные представления IDA

В стандартной конфигурации IDA 6.1 при загрузке нового файла создавала семь окон. Их вкладки находятся под навигационной полосой. Сразу видны IDA View, Functions и Output. Любое рассмотренное здесь окно можно повторно открыть через View > Open Subviews.

Клавиша `Esc` в окне дизассемблирования работает как кнопка Back в браузере и возвращает к предыдущему адресу. В большинстве других окон она закрывает активное представление, поэтому случайно исчезнувшую вкладку иногда приходится восстанавливать через меню.

Окно дизассемблирования

IDA View является главным средством анализа и изменения базы. Оно работает в двух форматах: графическом и текстовом. Предпочтение зависит от того, как пользователю удобнее воспринимать поток управления.

Граф используется по умолчанию. Чтобы сделать стандартным текстовый листинг, нужно отключить Use graph view by default на вкладке Graph в Options > General. Когда активно окно дизассемблирования, пробел переключает оба режима.

Граф IDA

Граф разбивает функцию на базовые блоки и показывает переходы между ними. Базовый блок - максимальная последовательность инструкций с одной точкой входа и одной точкой выхода, которая выполняется от начала до конца без внутреннего ветвления.

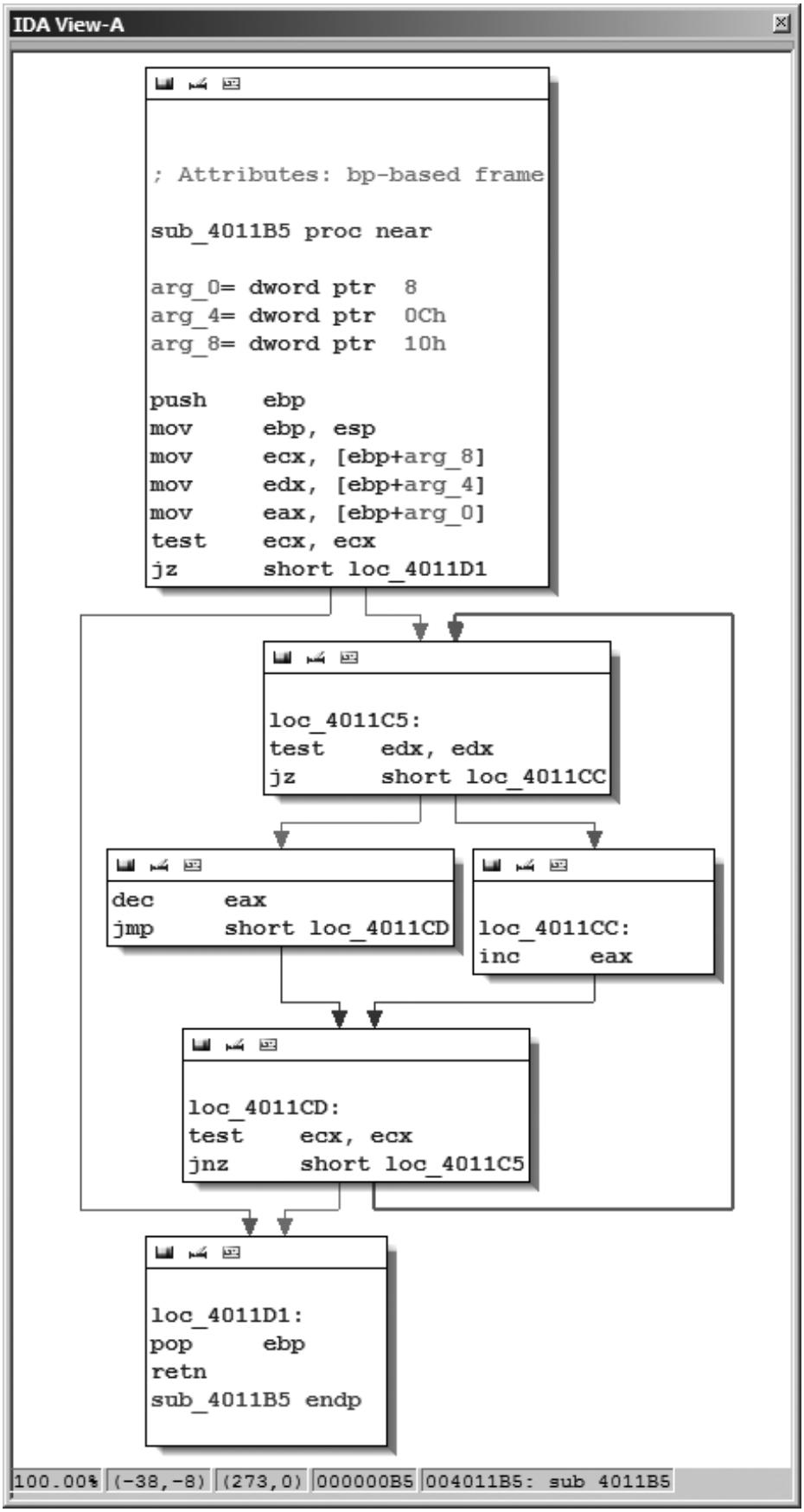

Рис. 5-1. Граф IDA

Цвет стрелки обозначает тип потока:

• зелёная стрелка Yes соответствует выполненному условному переходу;

• красная No соответствует пути, когда переход не выполнен;
• синяя Normal соединяет блок с единственным следующим блоком.

В графическом режиме одновременно показывается одна функция. Масштаб меняется `Ctrl` с колёсиком мыши либо сочетаниями `Ctrl` и клавишами + или - цифровой клавиатуры.

Большая функция быстро превращается в сложную схему. Окно Graph Overview всегда показывает её целиком, а пунктирная рамка отмечает текущую видимую область. Рамку можно перетаскивать, быстро перемещаясь по графу.

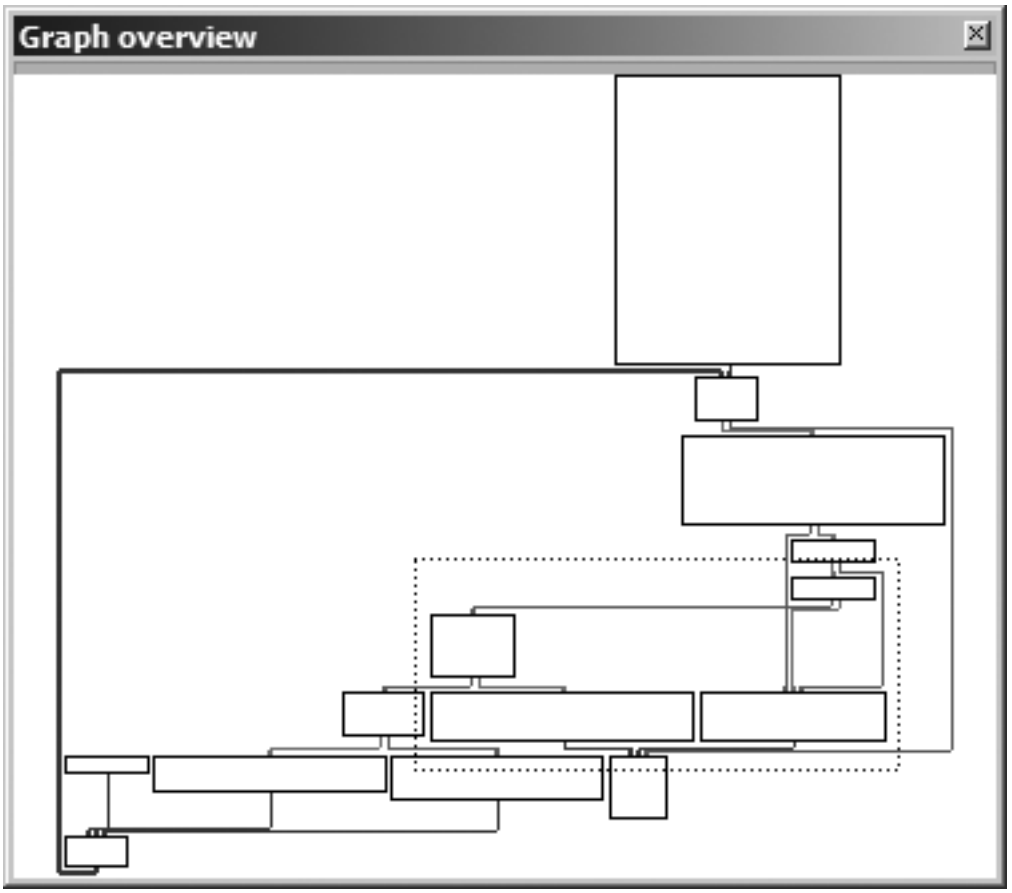

Рис. 5-2. Окно Graph Overview

Граф можно настраивать несколькими способами.

Перемещение. Кроме Graph Overview, схему можно двигать, перетаскивая пустой фон.

Дополнительные данные строк. Ради компактности в блоках скрыты некоторые традиционные элементы, например виртуальные адреса. Их можно включить на вкладке Disassembly в Options > General. Параметр line prefixes добавляет адрес перед каждой строкой.

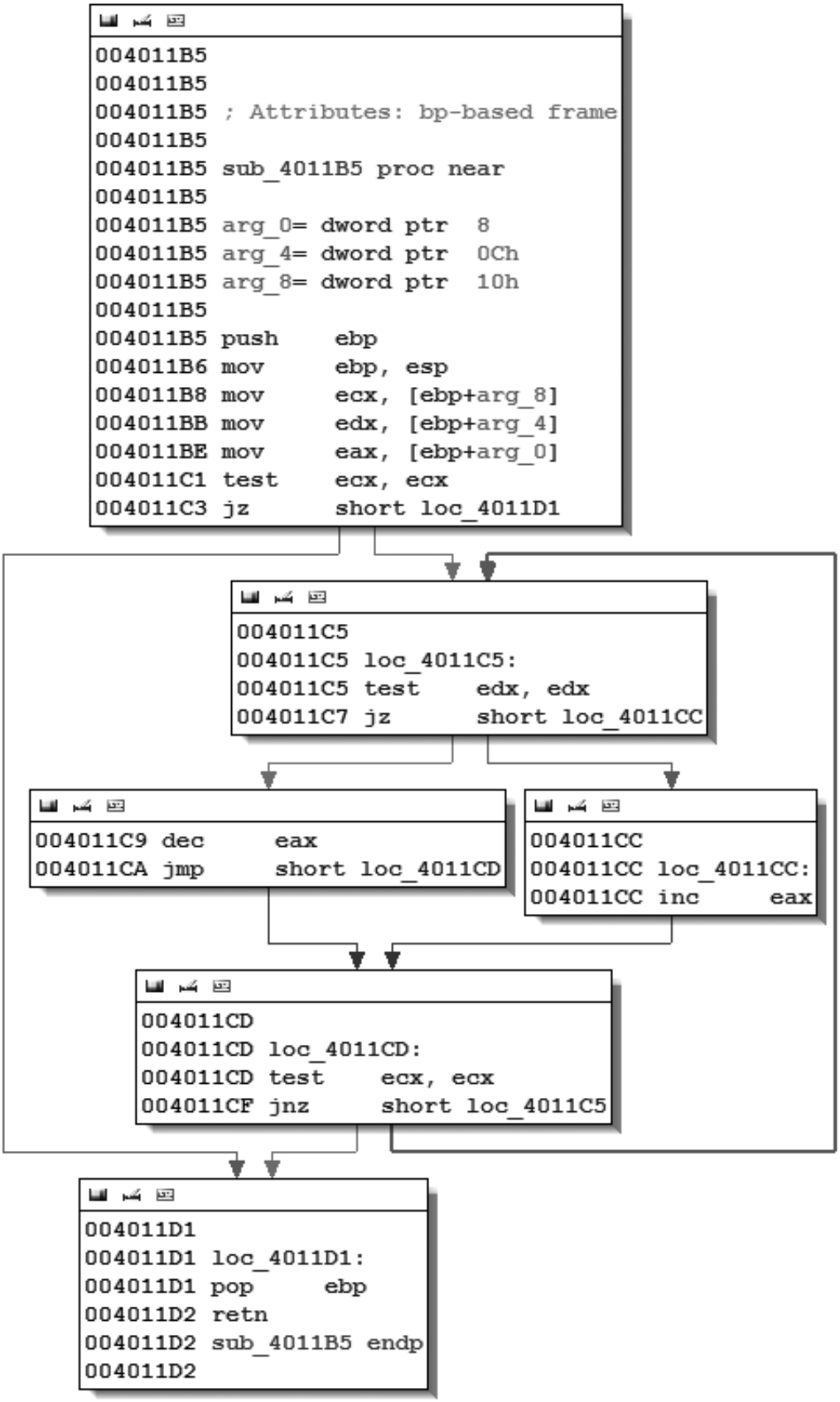

Рис. 5-3. Адреса строк в графе

Перестановка блоков. Блок перемещается за заголовок. IDA лишь минимально перенаправляет стрелки, поэтому вершины линий иногда нужно двигать вручную. Новую вершину можно создать двойным щелчком по ребру при зажатой `Shift`. Команда Layout Graph в контекстном меню

восстанавливает автоматическую раскладку.

Группировка и сворачивание. Один или несколько блоков можно объединить и свернуть через Group Nodes. Это удобно, чтобы убрать с экрана уже исследованные части.

Дополнительные окна. View > Open Subviews > Disassembly создаёт ещё одно независимое представление. Первое называется IDA View-A, следующие - IDA View-B, IDA View-C и так далее. В одном можно оставить граф, в другом листинг или открыть несколько разных функций одновременно.

Другие возможности графов рассматриваются в главе 9.

Текстовый вид

Текстовое представление - традиционный линейный листинг всей программы. В отличие от графа одной функции, только здесь видны области данных. Все сведения графического режима в той или иной форме доступны и в тексте.

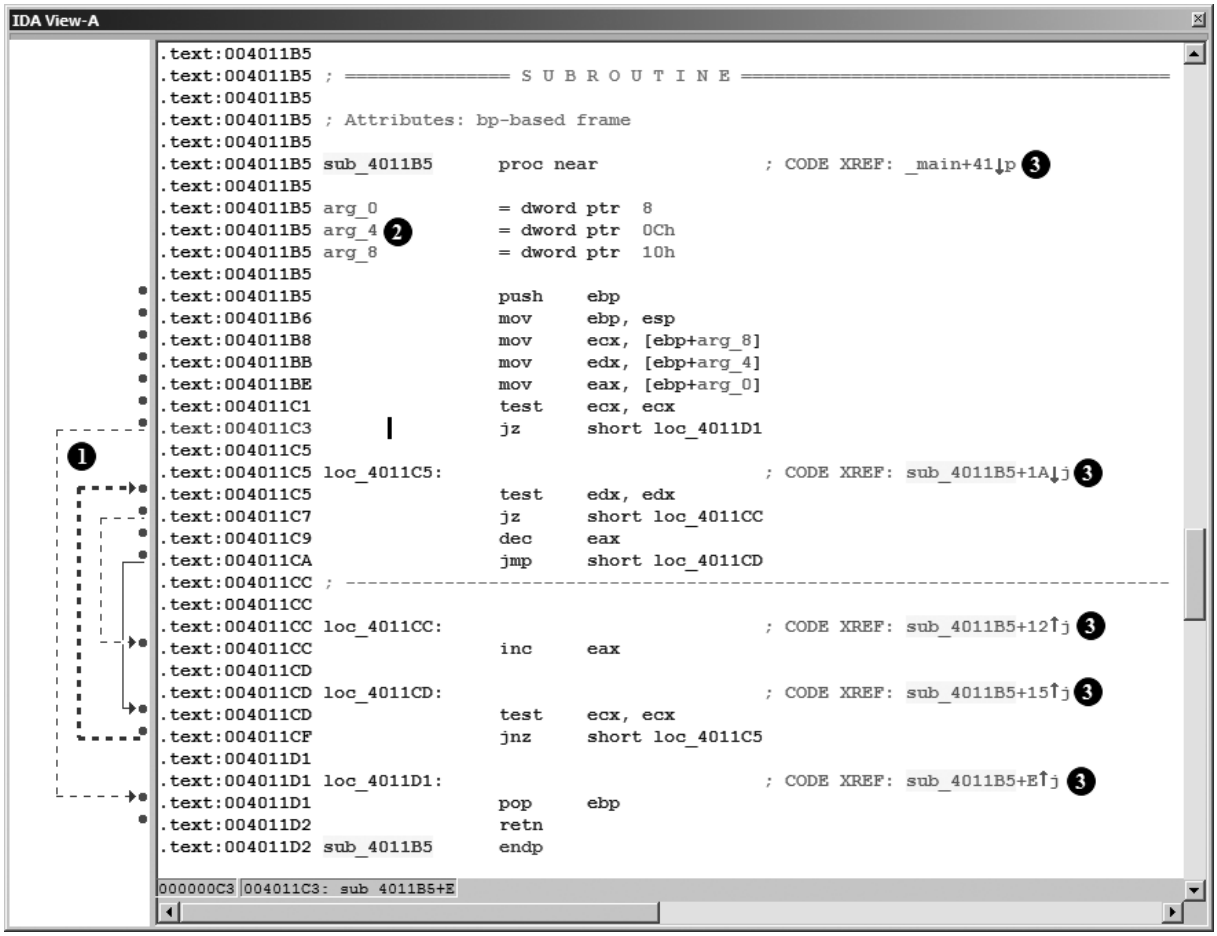

Рис. 5-4. Текстовый вид IDA

Виртуальный адрес обычно имеет вид `[СЕКЦИЯ]:[АДРЕС]`, например `.text:004011C1`.

Слева находится область стрелок, показывающая нелинейный поток функции. Сплошные линии обозначают безусловные переходы, пунктирные - условные. Обратный переход к меньшему адресу рисуется утолщённой линией и часто указывает на цикл. На рисунке цикл идёт от `004011CF` к `004011C5`.

Объявления `arg_0`, `arg_4` и `arg_8` отражают оценку IDA структуры кадра стека. Она строится по изменениям указателя стека и регистра кадра. Подробнее стеки рассматриваются в главе 6.

Комментарии после точки с запятой содержат перекрёстные ссылки. Ссылка CODE XREF означает, что другая инструкция передаёт управление на этот адрес. Перекрёстным ссылкам посвящена глава 9.

Далее в книге для примеров преимущественно используется текстовый вид, а граф - там, где он делает структуру существенно яснее. Изменение и аннотирование листинга рассматриваются в главе 7.

Окно Functions

Functions перечисляет все распознанные функции. Пример строки:

```
malloc  .text  00BDC260 00000180 R . . . B . .
```

Здесь `malloc` находится в секции `.text` по адресу `00BDC260`, имеет длину `0x180`, возвращает управление вызывающему коду (`R`) и использует EBP как базу локальных переменных (`B`). Значение всех флагов приведено во встроенной справке и в диалоге Properties контекстного меню. Двойной щелчок переносит IDA View к функции.

Окно Output

Output служит консолью IDA. При загрузке в нём показываются фазы анализа и действия по созданию базы, позднее - состояния операций и ошибки. Контекстное меню позволяет скопировать или очистить содержимое. Сценарии и плагины также часто выводят результаты именно сюда.

Вторичные представления

Дополнительные вкладки дают специализированные виды базы. Их полезность зависит от двоичного файла и конкретной задачи.

Hex View

Название несколько условно: окно поддерживает разные форматы и одновременно является шестнадцатеричным редактором. По умолчанию оно показывает 16 байт в строке и соответствующие ASCII-символы.

Можно открыть Hex View-A, Hex View-B и последующие независимые окна. Первое по умолчанию синхронизировано с IDA View-A. При прокрутке одного второе переходит к тому же виртуальному адресу, а выбранной инструкции соответствуют выделенные байты.

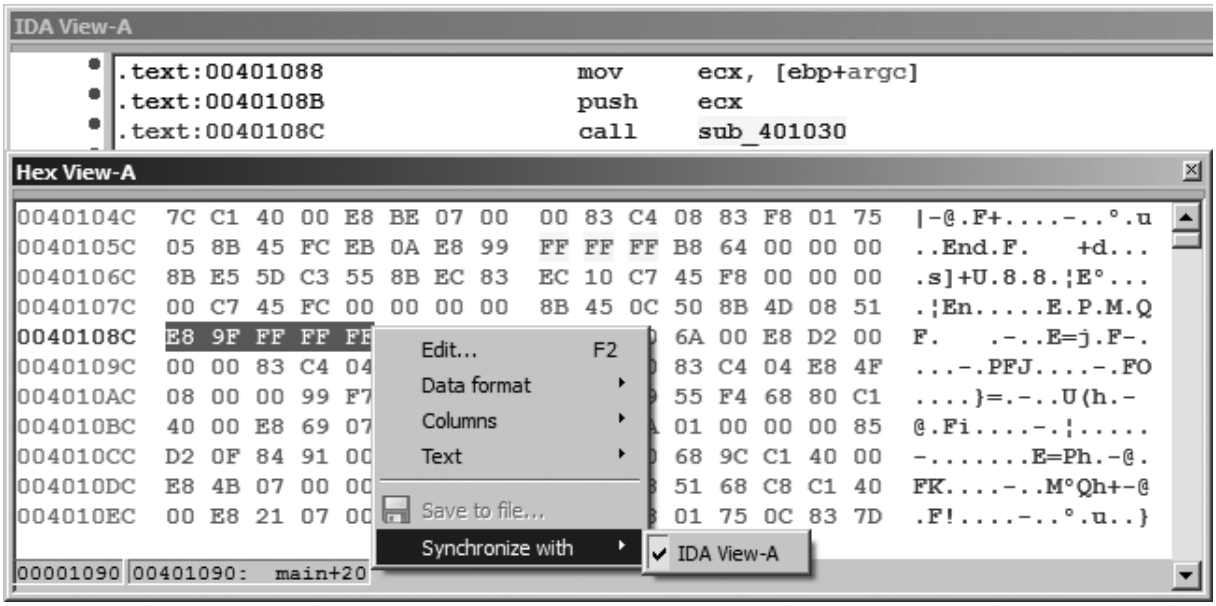

Рис. 5-5. Синхронизация Hex View и IDA View

Контекстное меню выбирает окно для синхронизации или отключает связь. Edit переводит представление в режим редактирования; затем изменения нужно подтвердить либо отменить. Data Format переключает значения размером 1, 2, 4 или 8 байт, знаковые и беззнаковые десятичные числа и форматы с плавающей точкой. Columns меняет число столбцов, Text включает текстовую часть.

Знаки вопроса вместо байтов означают, что IDA не знает содержимое виртуального диапазона. Типичный пример - секция BSS: в файле она почти не занимает места, но загрузчик выделяет память для неинициализированных статических переменных и заполняет её нулями только при запуске.

Exports

Exports перечисляет точки входа: начальный адрес исполнения и экспортируемые функции или данные. Экспорт особенно характерен для общих библиотек, например DLL. Показываются имя, виртуальный адрес и при наличии порядковый номер.

```
LoadLibraryA  7C801D77 578
```

Даже у обычной программы есть как минимум запись `start`. Двойной щелчок переходит к адресу. Аналогичные сведения дают `objdump -T`, `readelf -s` и `dumpbin /EXPORTS`.

Порядковый номер позволяет экспортировать функцию по числу, а не по имени. Это ускоряет поиск адреса и может скрыть исходное имя; такой способ встречается в DLL Windows.

Imports

Imports перечисляет функции и данные из общих библиотек. У полностью статически скомпонованного файла внешних зависимостей нет, поэтому импортов не будет. Каждая строка содержит имя элемента и библиотеку:

```
0040E108  GetModuleHandleA  KERNEL32
```

Адрес указывает на запись таблицы импорта, куда системный загрузчик во время запуска запишет фактический адрес функции. При статическом анализе содержимое этой памяти неизвестно и в Hex View отображается как `?? ?? ?? ??`.

Окно показывает только символы, которые программа поручает обработать динамическому загрузчику. Функции, загружаемые вручную через `dlopen` и `dlsym` либо `LoadLibrary` и `GetProcAddress`, здесь отсутствуют.

Structures

Structures показывает сложные типы данных, например структуры и объединения C, которые IDA считает используемыми. Во время анализа она сопоставляет типы параметров известных функций с памятью программы. Например, вызов `connect` может привести к добавлению стандартной структуры `sockaddr`, содержащей сетевой адрес и порт.

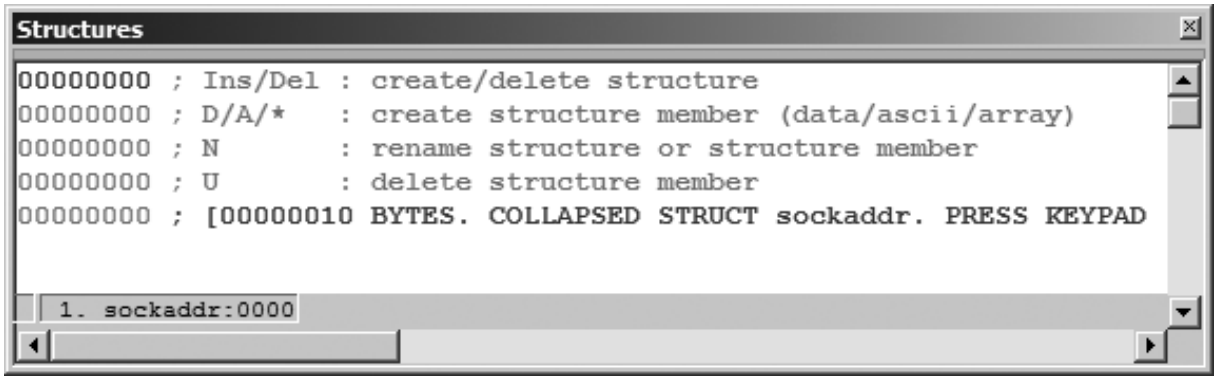

Рис. 5-6. Окно Structures

Двойной щелчок раскрывает поля, их смещения и размеры. Окно служит справочником стандартных типов и редактором собственных структур, которые затем применяются как шаблоны разметки памяти. Подробности приведены в главе 8.

Enums

Enums сходно со Structures, но работает с перечислимыми типами C. Замена числовых констант именованными элементами перечисления делает листинг гораздо понятнее. Пользователь может создавать и применять собственные enum.

Дополнительные представления

Следующие окна не открываются автоматически, но доступны через View > Open Subviews.

Strings

Strings является расширенным встроенным аналогом утилиты `strings`. Оно показывает найденную строку и её адрес. Двойной щелчок переходит к данным. Вместе с перекрёстными ссылками это позволяет быстро найти интересный текст и код, который его использует. Например, строка `SOFTWARE\Microsoft\Windows\CurrentVersion\Run` может привести к месту, где программа

настраивает автоматический запуск.

IDA не хранит результаты поиска строк постоянно. При каждом открытии окна база сканируется заново согласно текущим настройкам. Setup из контекстного меню выбирает допустимые типы. По умолчанию ищутся завершённые нулём строки C не короче пяти 7-битных ASCII-символов.

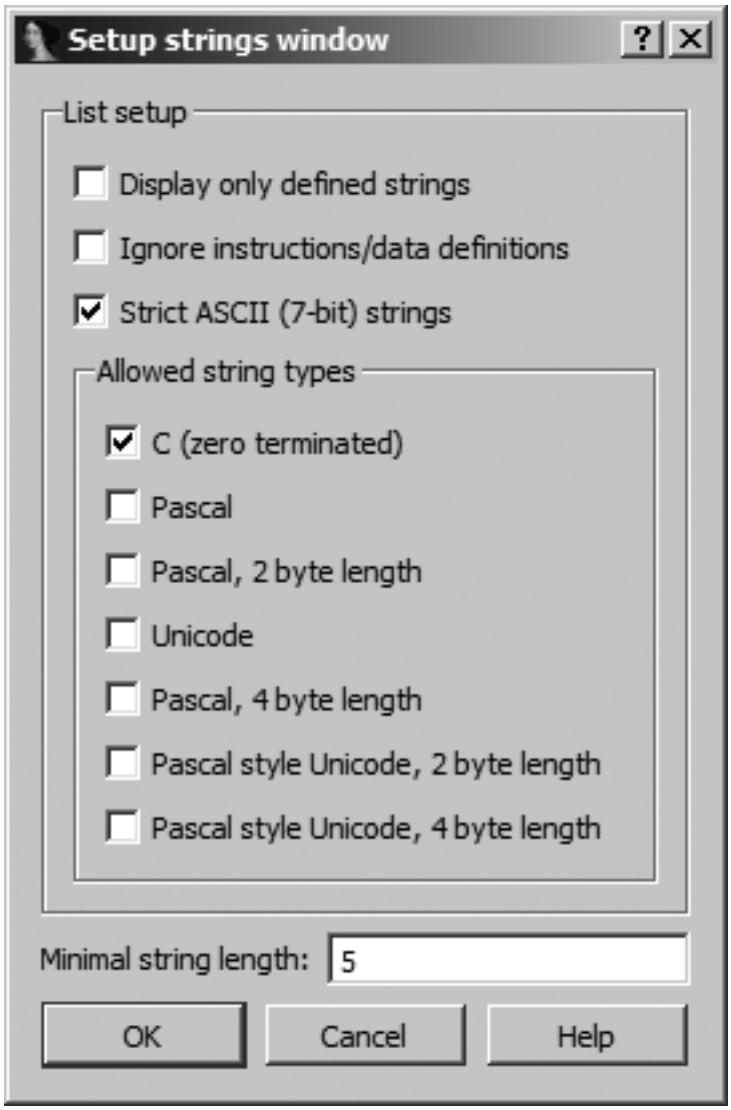

Рис. 5-7. Диалог Setup Strings

Для Windows часто нужно включить Unicode, а для двоичных файлов Delphi - строки Pascal с двухбайтовой длиной. После подтверждения настроек IDA повторно сканирует базу.

Два параметра особенно важны:

• **Display only defined strings** оставляет только уже определённые именованные строки и отключает автоматический поиск остальных;
• **Ignore instructions/data definitions** ищет последовательности внутри существующих инструкций и определений данных. Так обнаруживаются строки, ошибочно разобранные как код или массив чисел, но также появляется много случайного мусора. Результат похож на `strings -a`.

На следующем рисунке второй параметр отключён, поэтому строка по адресу `.rdata:0040C19C` не попала в список: ранее эти байты были определены как 32-разрядные числа.

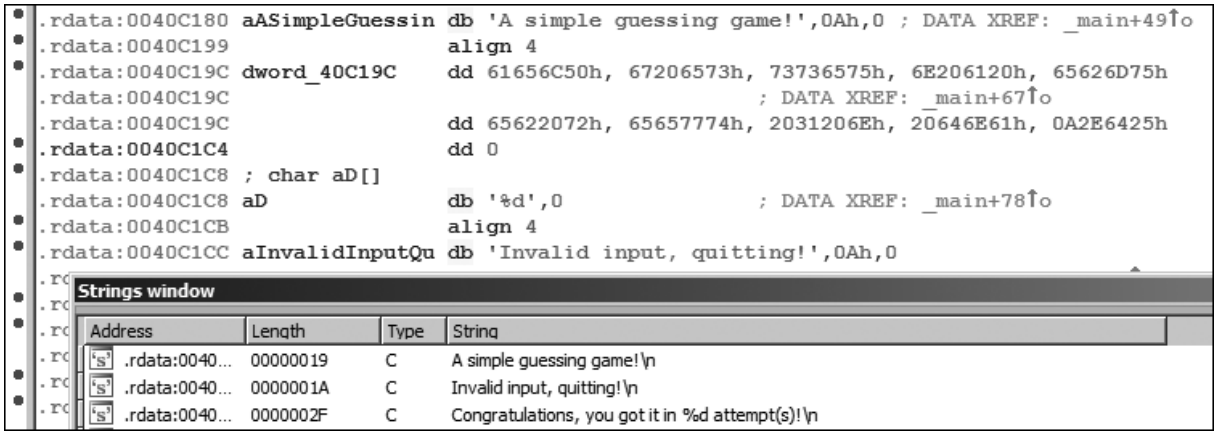

Рис. 5-8. Пример нераспознанной строки

Следует заранее понимать ожидаемые кодировки и места размещения и соответствующим образом настроить поиск.

Names

Names содержит глобальные именованные адреса. Первоначальные имена IDA получает из таблицы символов и сопоставления сигнатур. Список сортируется по имени или адресу. Двойной щелчок немедленно переносит листинг к выбранному объекту.

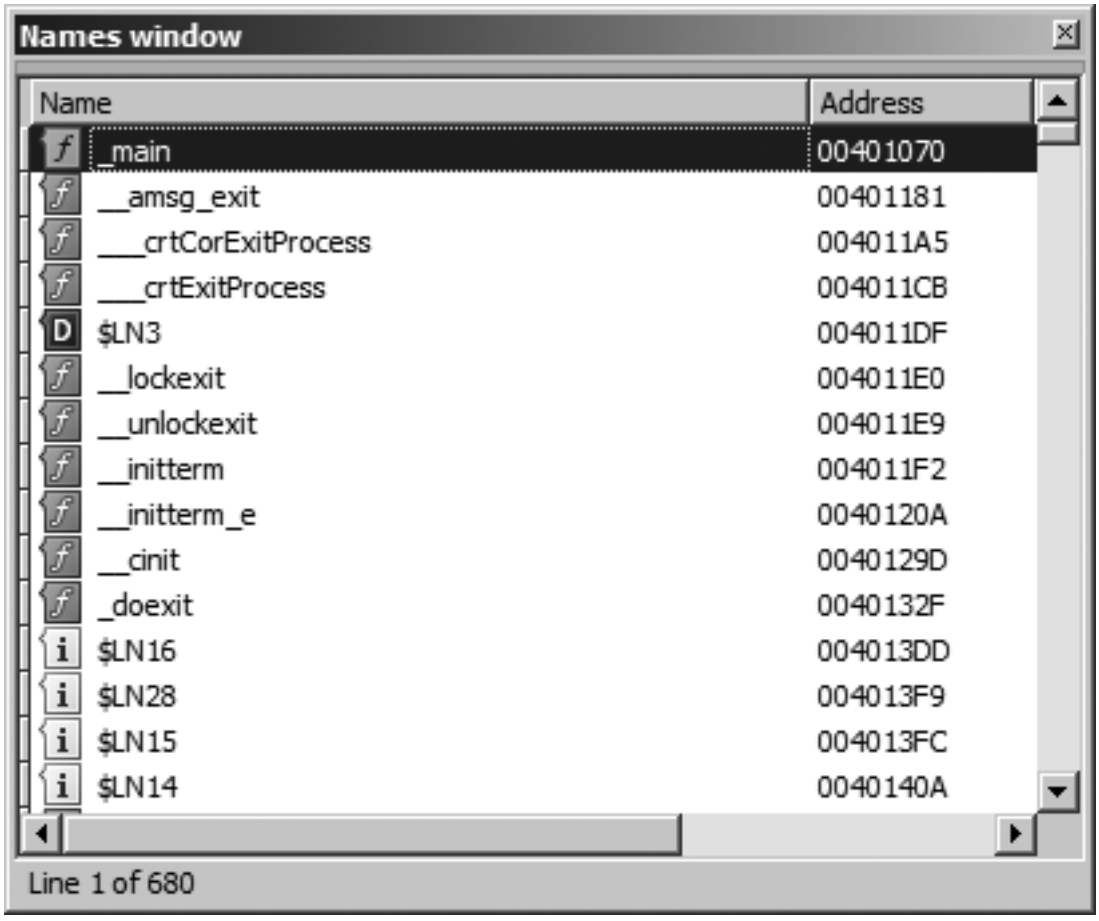

Рис. 5-9. Окно Names

Буква и цвет описывают категорию:

• F - обычная функция, не распознанная как библиотечная;
• L - библиотечная функция, найденная по сигнатуре;
• I - импортированный символ, кода которого в файле нет;
• C - именованный код вне распознанной функции;
• D - именованные данные, обычно глобальная переменная;
• A - строковые данные известного IDA типа.

В листинге встречается множество имён, отсутствующих в Names. IDA автоматически именует все непосредственно указанные цели переходов, вызовов и обращений к данным. Если таблица символов содержит имя, используется оно. Иначе создаётся уникальное имя из типа и адреса:

• sub_xxxxxx - подпрограмма по адресу xxxxxx;
• loc_xxxxxx - адрес инструкции;
• byte_xxxxxx - 8-битные данные;
• word_xxxxxx - 16-битные данные;
• dword_xxxxxx - 32-битные данные;
• unk_xxxxxx - данные неизвестного размера.

Такие автоматически созданные локальные имена в Names не показываются.

Segments

Segments содержит секции двоичного файла, хотя IDA исторически называет их сегментами. Не следует путать их с аппаратными сегментами памяти процессоров. Для каждой секции указаны имя, начальный и конечный виртуальные адреса и разрешения.

```
Name   Start    End      R W X  Class
UPX0   00401000 00407000 R W X  CODE
UPX1   00407000 00408000 R W X  CODE
UPX2   00408000 0040803C R W .  DATA
.idata 0040803C 00408050 R W .  XTRN
UPX2   00408050 00409000 R W .  DATA
```

Нестандартные имена и одновременно доступные для записи и исполнения секции намекают на упаковку и самомодифицирующийся код. Размер секции не означает, что IDA знает каждый её байт: в памяти она может быть больше, чем в файле. Неизвестная часть показывается вопросительными знаками.

Двойной щелчок переходит к началу секции. Контекстное меню создаёт, удаляет и редактирует сегменты, что особенно полезно для нестандартного формата, который загрузчик разобрал не полностью. Консольные аналоги: `objdump -h`, `readelf -S`, `dumpbin /HEADERS`.

Signatures

IDA использует сигнатуры известных блоков кода. По шаблонам распознаются стартовые последовательности компилятора и библиотечные функции, добавленные статической компоновкой. Аналитику не приходится тратить время на внутреннее устройство стандартного `printf`.

Signatures перечисляет применённые к текущему файлу наборы:

```
File     State    #func Library name
vc32rtf  Applied  501   Microsoft VisualC 2-8/net runtime
```

В данном случае набор `vc32rtf` из `<IDADIR>/sigs` распознал 501 библиотечную функцию.

Дополнительные сигнатуры применяют, когда IDA не определила компилятор или когда пользователь создал собственный набор для отсутствующей библиотеки. `Insert` либо Apply new signature в контекстном меню открывает список. Создание FLIRT-сигнатур рассматривается в главе 12.

Type Libraries

Библиотеки типов содержат накопленные IDA сведения о типах и прототипах функций из заголовочных файлов популярных компиляторов. Благодаря им программа знает параметры библиотечных функций, размеры и разметку сложных структур и аннотирует листинг.

Информация собрана в файлах TIL из `<IDADIR>/til`. Для автоматического выбора IDA должна определить используемые библиотеки. Дополнительный набор загружается клавишей `Insert` или командой Load type library. Подробнее это рассматривается в главе 13.

Function Calls

Функция может вызывать другие функции и сама вызываться из разных мест. Полный граф вызовов показывает все отношения, но иногда нужны только непосредственные соседи текущей функции: кто вызывает её и кого вызывает она.

Function Calls строится для функции под курсором. Верхняя таблица содержит вызывающие места, нижняя - сделанные ею вызовы.

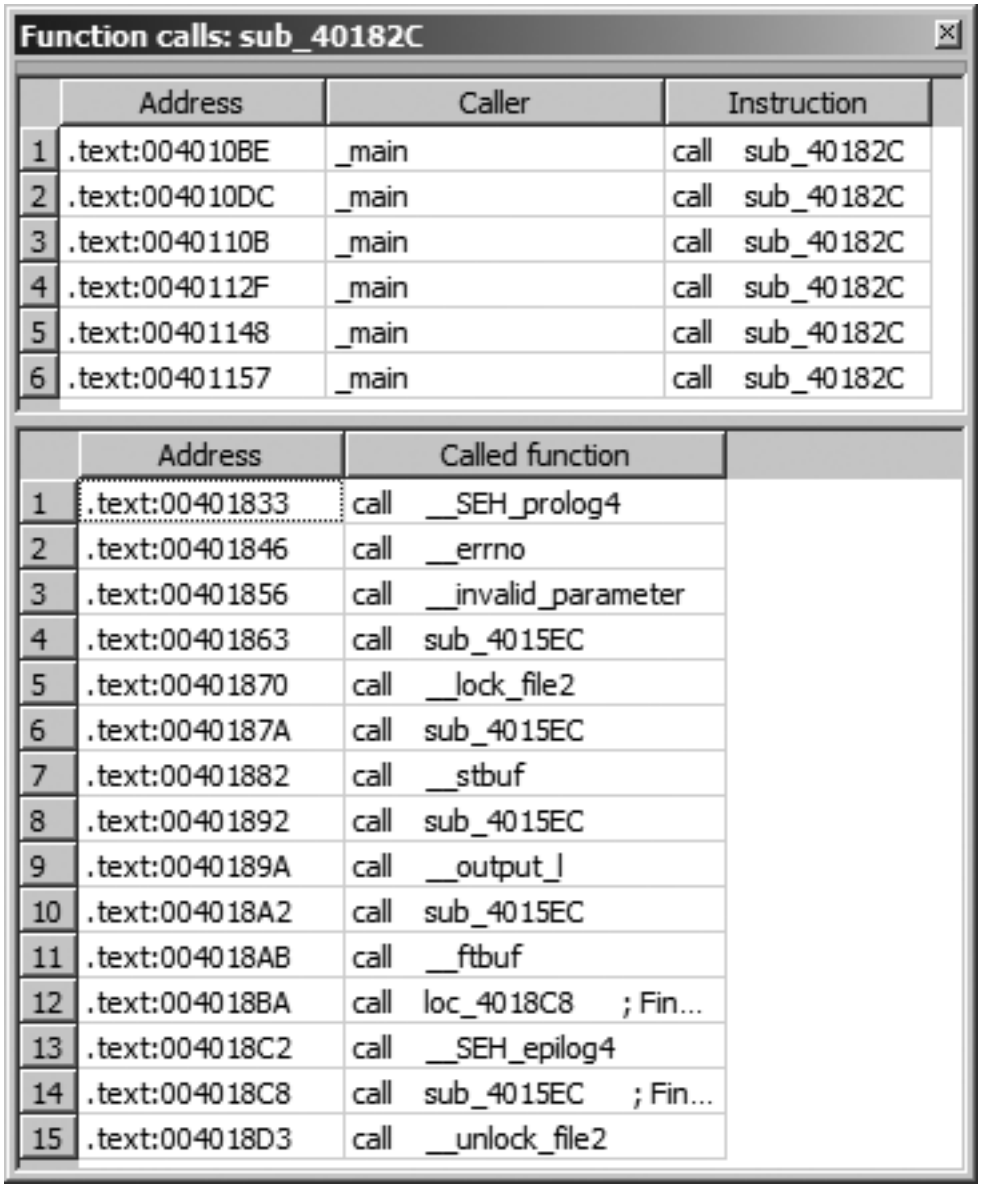

Рис. 5-10. Окно Function Calls

В примере `sub_40182C` вызывается из шести мест `_main`, а сама `_main` выполняет ещё 15 вызовов. Двойной щелчок переходит к выбранной инструкции или функции. Окно строится на перекрёстных ссылках, которые подробно рассматриваются в главе 9.

Problems

Problems сообщает о трудностях дизассемблирования и решениях IDA. Иногда пользователь может исправить результат, иногда проблему лучше оставить без изменений. Даже простая программа способна породить записи:

```
Address          Type      Instruction
.text:0040104C   BOUNDS    call eax
.text:004010B0   BOUNDS    call eax
.text:00401108   BOUNDS    call eax
.text:00401350   BOUNDS    call dword ptr [eax]
.text:004012A0   DECISION  push ebp
.text:004012D0   DECISION  push ebp
.text:00401560   DECISION  jmp ds:__set_app_type
.text:004015F8   DECISION  dd 0FFFFFFFFh
```

```
.text:004015FC   DECISION  dd 0
```

Для каждой проблемы показаны адрес, тип и инструкция. `BOUNDS` возникает, когда цель перехода или вызова неизвестна либо лежит вне виртуального диапазона программы. Например, IDA не знает статическое значение EAX в `call eax`.

`DECISION` чаще не является настоящей ошибкой. Это адрес, байты которого IDA решила считать инструкциями, хотя рекурсивный проход не нашёл явной ссылки на него. Полный список и рекомендации находятся в разделе Problems List встроенной справки.

Итоги

Количество окон сначала кажется избыточным. Удобнее начать с IDA View, Functions и Output, а остальные подключать по мере необходимости. Не каждое представление полезно в каждой задаче.

Кроме окон, IDA содержит множество диалогов, которые будут вводиться далее по мере применения. Графы, кроме стандартного графа функции, сознательно оставлены до главы 9: меню IDA выделяет их в отдельную категорию.

Следующая глава посвящена навигации по дизассемблированному коду, а затем мы перейдём к изменению представления, чтобы сделать поведение программы понятнее.

Глава 6. Навигация по дизассемблированному коду

Интерактивность IDA прежде всего означает удобную навигацию и возможность изменять базу. Эта глава посвящена перемещению по дизассемблированному коду, следующая - его редактированию.

Обычный статический листинг допускает главным образом прокрутку. Даже хороший текстовый редактор обычно предлагает лишь поиск наподобие `grep`. База данных IDA связывает адреса, имена и ссылки, поэтому позволяет перемещаться по программе логически.

Основы навигации

IDA сочетает привычный текстовый поиск с исчерпывающим набором перекрёстных ссылок, похожих на гиперссылки. В большинстве случаев достаточно двойного щелчка.

Навигация двойным щелчком

Каждому месту программы назначен виртуальный адрес. Теоретически можно перейти куда угодно, если помнить число, но символические имена гораздо удобнее. Они играют ту же роль, что мнемоники инструкций вместо кодов операций.

IDA получает имена из таблицы символов или создаёт их по способу использования адреса. Любое имя в листинге является целью навигации. В отличие от веб-ссылки оно не выделено специальным оформлением и открывается двойным, а не одиночным щелчком.

```
.text:0040132B loc_40132B:      ; CODE XREF: sub_4012E4+B
.text:0040132B     cmp edx, 0CDh
.text:00401331     jg  short loc_40134E
.text:00401333     jz  loc_4013BF
.text:00401339     sub edx, 0Ah
.text:0040133C     jz  short loc_4013A7
.text:0040133E     sub edx, 0C1h
.text:00401344     jz  short loc_4013AF
.text:00401346     dec edx
.text:00401347     jz  short loc_4013B7
.text:00401349     jmp loc_4013DD
```

Двойной щелчок по любому `loc_...` переносит к цели.

Перекрёстная ссылка тоже является целью. Запись `sub_4012E4+4D` означает место на `0x4D`, или 77 десятичных байтов, дальше начала `sub_4012E4`. Двойной щелчок по ней переносит к инструкции, которая ссылается на текущий адрес. Xref подробно рассматриваются в главе 9.

Третья возможность - шестнадцатеричное число, которое является допустимым виртуальным адресом текущего файла:

```
.data:00409013  db 4
.data:00409014  dd 4037B0h  ; переход возможен
.data:00409018  db 0
.data:00409019  db 0Ah
.data:0040901A  dd 404590h  ; переход возможен
.data:0040901E  db 0
.data:0040901F  db 0Ah
.data:00409020  dd 404DA8h  ; переход возможен
```

Числа `4`, `0` и `0Ah` адресами не являются, поэтому двойной щелчок ничего не делает.

В Output навигация работает, только если допустимый адрес или имя стоит первым элементом сообщения. Например, строка `40134e is an interesting location` открывает адрес, а `Testing: 40134e` нет.

Переход к адресу

Когда адрес известен, но на экране нет подходящего имени, можно прокручивать листинг, искать функцию по имени или использовать Search. Самый прямой способ - Jump > Jump to Address либо горячая клавиша `G`.

Рис. 6-1. Диалог Jump to Address

В поле принимается имя или шестнадцатеричный адрес. Предыдущие значения сохраняются в раскрывающемся списке.

История навигации

Как веб-браузер запоминает посещённые страницы, IDA добавляет каждое новое место в историю.

Jump > Jump to Previous Position и клавиша `Esc` возвращают назад. Это особенно полезно после перехода по цепочке вызовов на несколько уровней. Нужно помнить, что `Esc` в другом активном окне не возвращает адрес, а закрывает это окно.

Jump > Jump to Next Position и `Ctrl+Enter` идут вперёд по истории. На панели доступны две знакомые кнопки со списками, позволяющими сразу выбрать любую сохранённую позицию.

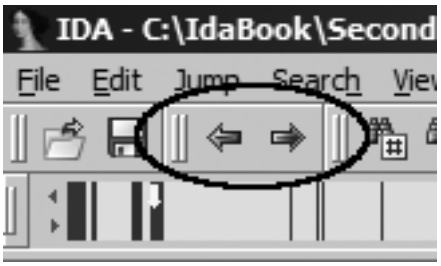

Рис. 6-2. Кнопки истории навигации

Кадры стека

Для понимания некоторых представлений IDA необходимы низкоуровневые сведения о скомпилированных программах. Один из ключевых терминов - кадр стека, или запись активации.

Кадр стека - блок памяти в стеке времени выполнения, принадлежащий одному конкретному вызову функции. Пока функция не выполняется, ей обычно почти не нужна память. Во время вызова требуется хранить переданные аргументы, временные локальные переменные и адрес возврата.

Компилятор автоматически создаёт и удаляет кадры, скрывая эту работу от программиста. Благодаря отдельному кадру для каждого вызова возможна рекурсия: данные вложенного вызова не смешиваются с данными предыдущего.

Типичная последовательность такова:

1. Вызывающий код размещает аргументы в регистрах или стеке согласно соглашению о вызове.
2. Управление передаётся функции через инструкцию вроде x86 `call` или MIPS `jal`; адрес возврата сохраняется в стеке или регистре.
3. Функция при необходимости настраивает указатель кадра и сохраняет регистры, которые обязана оставить неизменными.
4. Выделяется место для локальных переменных, обычно изменением указателя стека.
5. Выполняется тело функции. Оно читает аргументы и помещает результат в согласованный регистр или регистры.
6. Освобождается область локальных переменных.
7. Восстанавливаются сохранённые регистры, включая указатель кадра вызывающей функции.
8. Инструкция `ret` или MIPS `jr` возвращает управление; в некоторых соглашениях она одновременно удаляет аргументы.
9. Если аргументы должен удалять вызывающий код, он восстанавливает свой указатель стека.

Шаги 3 и 4 образуют пролог функции, шаги 6-8 - эпилог. Всё, кроме тела на шаге 5, является накладными расходами вызова.

Соглашения о вызовах

Соглашение определяет, где находятся аргументы, кто удаляет их из стека и где возвращается результат. Вызывающий и вызываемый код обязаны следовать одинаковым правилам, иначе стек будет повреждён.

Ниже используются 32-разрядная x86 и типичное поведение Microsoft Visual C/C++ и GNU gcc/g++.

cdecl

Стандартное соглашение большинства компиляторов C для x86 обозначается `_cdecl`. Аргументы помещаются в стек справа налево, а после возврата их удаляет вызывающий код.

Правый порядок означает, что первый, самый левый параметр всегда находится на вершине при входе в функцию. Поэтому `cdecl` подходит для функций с переменным числом аргументов, например `printf`: вызывающий код знает точное число фактически переданных значений и способен правильно очистить стек.

```
void demo_cdecl(int w, int x, int y, int z);
```

Обычный вызов выглядит так:

```
; demo_cdecl(1, 2, 3, 4)
push 4              ; z
push 3              ; y
push 2              ; x
push 1              ; w
```

```
call demo_cdecl
add  esp, 16       ; вызывающий код удаляет четыре int
```

Четыре `push` изменяют ESP на 16 байт, а `add` возвращает прежнее значение. При множестве вызовов очистка повторяется после каждого.

gcc часто заранее резервирует область аргументов в прологе и записывает значения без изменения ESP:

```
mov [esp+12], 4
mov [esp+8],  3
mov [esp+4],  2
mov [esp],    1
call demo_cdecl
```

В обоих вариантах при вызове ESP указывает на первый параметр.

stdcall

Название Standard Calling Convention было введено Microsoft и обозначается `_stdcall`. Аргументы также передаются через стек справа налево, но удаляет их вызываемая функция.

```
void _stdcall demo_stdcall(int w, int x, int y);
```

Функция обязана знать точное число параметров, поэтому соглашение не подходит для `printf` и других функций переменной арности. Три `int` занимают 12 байт, и специальная форма `ret` одновременно извлекает адрес возврата и очищает аргументы:

```
ret 12
```

Преимущество - отсутствие отдельной инструкции очистки после каждого вызова, что слегка уменьшает и ускоряет код. Microsoft традиционно применяла `stdcall` для экспортируемых из DLL функций с фиксированным числом параметров. Это важно при восстановлении прототипа или создании двоично совместимой замены.

fastcall для x86

`fastcall` является вариантом `stdcall`: первые два аргумента передаются в ECX и EDX, остальные через стек справа налево. Стековые значения удаляет вызываемая функция.

```
void fastcall demo_fastcall(int w, int x, int y, int z);
```

```
; demo_fastcall(1, 2, 3, 4)
push 4             ; z
push 3             ; y
mov  edx, 2        ; x
mov  ecx, 1        ; w
call demo_fastcall
```

После возврата вызывающему коду не нужен `add esp`. Хотя аргументов четыре, функция очищает только восемь байтов, потому что два значения были в регистрах.

Соглашения C++

Нестатической функции класса нужен скрытый указатель `this` на объект. Стандарт C++ не задаёт способ передачи, поэтому компиляторы различаются.

Microsoft Visual C++ использовала `thiscall`: `this` передавался в ECX, а функция очищала стек подобно `stdcall`. GNU g++ считал `this` неявным первым параметром `cdecl`, помещал его на вершину стека и оставлял очистку вызывающему коду.

Другие особенности скомпилированного C++ рассматриваются в главе 8.

Другие соглашения

Соглашения зависят от языка, компилятора и архитектуры. Особенно внимательно следует анализировать три случая.

Оптимизированный внутренний код. Экспортируемая функция должна следовать известному ABI, но внутренний код программы может использовать частное соглашение, выбранное оптимизатором. Примеры - параметр `/GL` Visual C++ и `regparm` в gcc/g++.

Ручной ассемблер. Автор полностью контролирует передачу значений и может выбрать любую схему, если функция не предназначена для внешнего вызова. Такой код часто встречается в обфускации и шелл-коде.

Системные вызовы. Они переводят выполнение из пользовательского режима в режим ядра. Механизм зависит от ОС и процессора. Linux x86 использовал `int 0x80` или `sysenter`. Во многих системах номер вызова помещается в EAX, а параметры в стек. В Linux x86 аргументы в основном передаются через определённые регистры, а при их нехватке через память.

Размещение локальных переменных

В отличие от аргументов, для локальных переменных нет внешнего соглашения. Компилятор определяет общий размер, решает, какие значения оставить в регистрах, и размещает остальные в стеке. Это решение не видно вызывающей стороне, и точную разметку обычно нельзя восстановить только из исходного текста.

Пример кадра стека

Рассмотрим функцию x86:

```
void bar(int j, int k);

void demo_stackframe(int a, int b, int c) {
    int x;
    char buffer[64];
    int y;
    int z;
    bar(z, y);
}
```

Минимальный объём локальных данных равен 76 байтам: три `int` по 4 байта и массив на 64 байта.

Кадр относительно ESP

Если отдельного регистра кадра нет, его роль выполняет ESP. Пролог:

```
sub esp, 76
```

Положение	Объект	Адрес относительно ESP	Категория
ESP	z	[esp]	локальная переменная
	y	[esp+4]	локальная переменная
	buffer	[esp+8]	локальная переменная
	x	[esp+72]	локальная переменная
	сохранённый EIP	[esp+76]	адрес возврата
	a	[esp+80]	параметр
	b	[esp+84]	параметр
	c	[esp+88]	параметр

Рис. 6-3. Кадр стека относительно ESP

Вызов `bar(z, y)` показывает сложность переменного основания:

```
push dword [esp+4]  ; y
push dword [esp+4]  ; z после изменения ESP
call bar
add  esp, 8
```

Первая инструкция действительно берёт `y`. Затем `push` сдвигает ESP, и смещение `z` становится `[esp+4]`. При анализе такого кадра каждое изменение указателя стека требует пересчитывать все смещения. Преимущество - остальные регистры остаются свободными.

Перед возвратом локальные данные удаляются, чтобы ESP снова указывал на сохранённый EIP:

```
add esp, 76
ret
```

Кадр относительно EBP

Чаще x86-компилятор выделяет EBP как постоянный указатель кадра. Опция gcc `-fomit-frame-pointer` отключает это поведение. Типичный пролог:

```
push ebp       ; сохранить EBP вызывающей функции
mov  ebp, esp  ; установить основание кадра
sub  esp, 76   ; выделить локальные данные
```

По ABI System V для 32-разрядной Intel функция может свободно менять EAX, ECX и EDX, но обязана сохранять другие регистры. Поэтому старый EBP записывается в стек и восстанавливается перед возвратом. Для ESI, EDI и других сохраняемых регистров единого места в кадре нет.

Положение	Объект	Адрес относительно EBP	Категория
ESP	z	[ebp-76]	локальная переменная
	y	[ebp-72]	локальная переменная
	buffer	[ebp-68]	локальная переменная
	x	[ebp-4]	локальная переменная
EBP	сохранённый EBP	[ebp]	сохранённый регистр
	сохранённый EIP	[ebp+4]	адрес возврата
	a	[ebp+8]	параметр
	b	[ebp+12]	параметр
	c	[ebp+16]	параметр

Рис. 6-4. Кадр стека относительно EBP

Положительные смещения обычно относятся к аргументам, отрицательные - к локальным данным. ESP можно свободно менять, не влияя на адресацию:

```
push dword [ebp-72]  ; y
push dword [ebp-76]  ; z
call bar
add  esp, 8
```

Эпилог очищает локальную область и возвращает старый EBP:

```
mov esp, ebp
pop ebp
ret
```

Инструкция `leave` объединяет первые две операции:

```
leave
ret
```

Другие архитектуры используют иные регистры и инструкции, но общий принцип сохраняется. Умение узнавать типичные прологи и эпилоги позволяет быстрее переходить к содержательному телу функции.

Представления стека в IDA

Хотя кадр существует только во время исполнения, весь код его создания присутствует в файле. IDA тщательно отслеживает `push`, `pop` и арифметику указателя стека, определяет размер локальной области, наличие отдельного указателя кадра и обращения к переменным.

Например, при кадре EBP инструкция `mov eax, [ebp+8]` читает первый аргумент. IDA различает параметры ниже адреса возврата и локальные значения выше него, а также отмечает только непосредственно используемые места.

Рассмотрим вариант функции:

```
void demo_stackframe(int a, int b, int c) {
    int x = c;
    char buffer[64];
    int y = b;
    int z = 10;
    buffer[0] = 'A';
    bar(z, y);
}
```

Фрагмент вывода IDA:

```
.text:00401090 demo_stackframe proc near
.text:00401090 var_60 = dword ptr -60h
.text:00401090 var_5C = dword ptr -5Ch
.text:00401090 var_58 = byte ptr  -58h
.text:00401090 var_C  = dword ptr -0Ch
.text:00401090 arg_4  = dword ptr  0Ch
.text:00401090 arg_8  = dword ptr  10h
.text:00401090        push ebp
.text:00401091        mov  ebp, esp
.text:00401093        sub  esp, 78h
.text:00401096        mov  eax, [ebp+arg_8]
.text:00401099        mov  [ebp+var_C], eax
.text:0040109C        mov  eax, [ebp+arg_4]
.text:0040109F        mov  [ebp+var_5C], eax
.text:004010A2        mov  [ebp+var_60], 0Ah
.text:004010A9        mov  [ebp+var_58], 41h
.text:004010AD        mov  eax, [ebp+var_5C]
.text:004010B0        mov  [esp+4], eax
.text:004010B4        mov  eax, [ebp+var_60]
.text:004010B7        mov  [esp], eax
.text:004010BA        call bar
.text:004010BF        leave
.text:004010C0        retn
```

IDA узнала кадр на основе EBP. gcc выделил `0x78`, то есть 120 байт: сюда входят восемь байтов для двух аргументов `bar`, но всё равно больше ожидаемых 76. Компиляторы добавляют выравнивающий запас.

Краткое представление стека перечисляет только непосредственно использованные переменные, их размер и смещение. Локальные имена имеют префикс `var_` и величину отрицательного смещения: `var_C` означает `[ebp-0Ch]`. Аргументы называются `arg_0`, `arg_4`, `arg_8` и так далее. Неиспользуемый a отсутствует, поскольку в коде нет обращения `[ebp+8]`.

IDA заменяет числовые смещения символическими именами. `[ebp+arg_8]` равнозначно `[ebp+10h]`, но читается лучше. Контекстное меню позволяет вернуть числовой формат или выбрать другое представление.

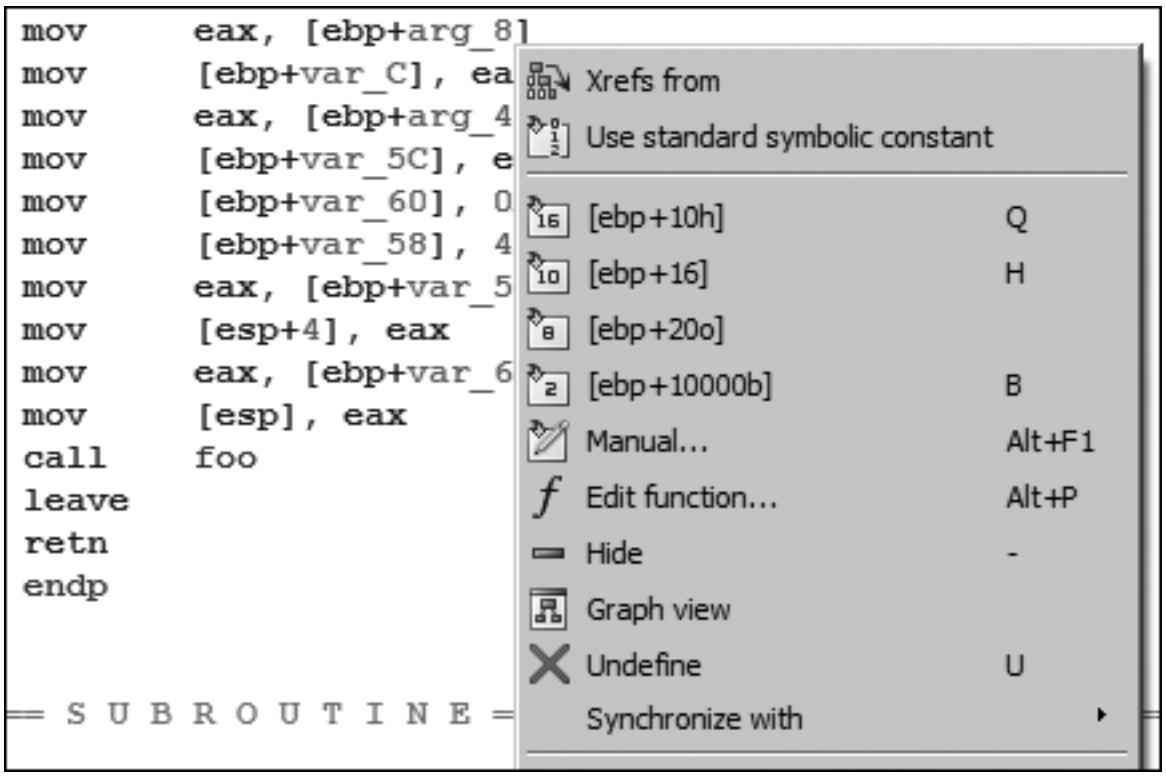

Рис. 6-5. Выбор альтернативного формата

Сопоставляя исходный код и инструкции, можно восстановить переменные:

1. a, b, c соответствуют `arg_0`, `arg_4`, `arg_8`, но `arg_0` не показан из-за отсутствия обращений.
2. x получает c, следовательно это `var_C`.
3. y получает b, следовательно это `var_5C`.
4. z инициализируется числом 10, значит это `var_60`.
5. Первый байт `buffer` получает `0x41`, значит массив начинается с `var_58`.
6. gcc записывает аргументы `bar` в заранее выделенную верхнюю область, а не выполняет `push`; IDA не создаёт для этих мест отдельные локальные имена.

Двойной щелчок по имени переменной открывает подробный кадр, где учтён каждый байт.

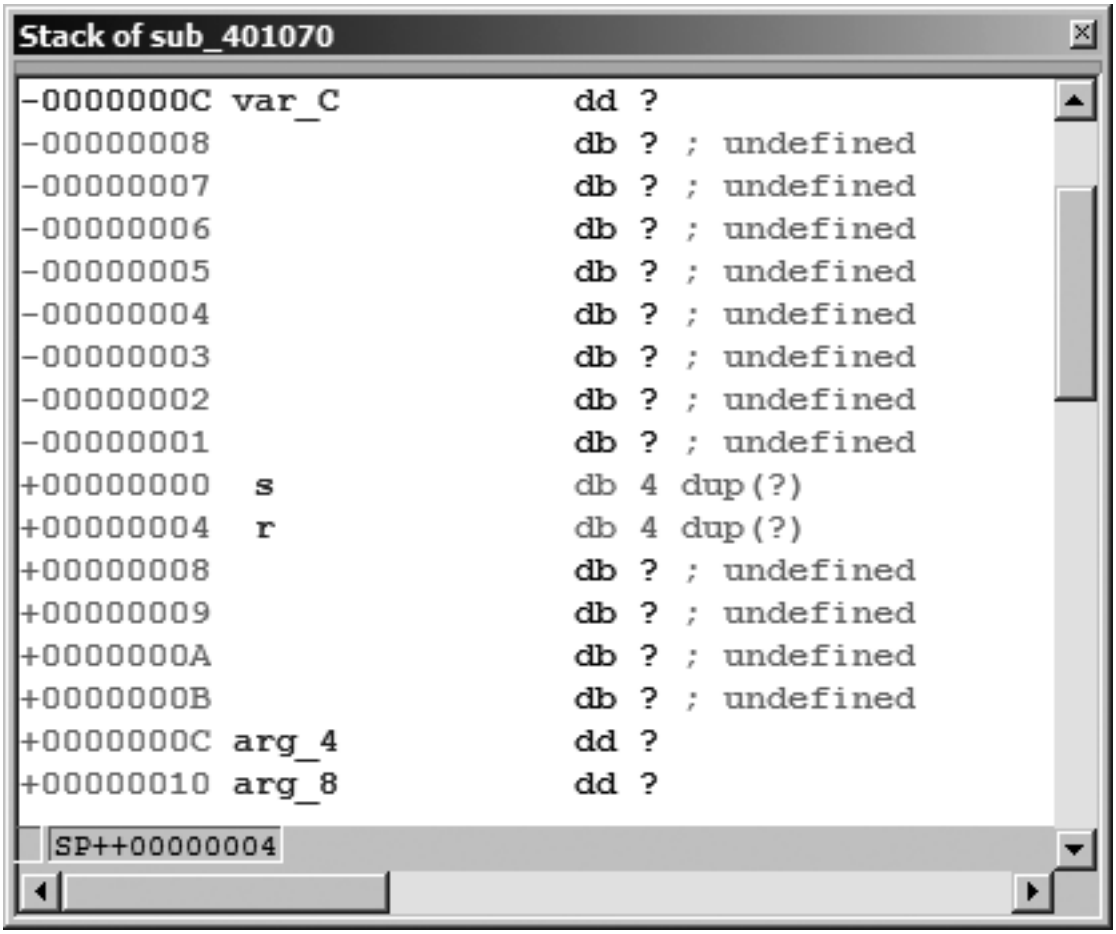

Рис. 6-6. Кадр стека в IDA

Без прямой ссылки байт остаётся без имени. `arg_4` прочитан в 32-разрядный EAX, поэтому IDA определяет его как `dd`, то есть четыре байта. `db` задаёт один байт, `dw` два, `dd` четыре.

Псевдопеременная `r` обозначает сохранённый адрес возврата, `s` - сохранённые регистры, здесь EBP. Подробный вид показывает и заполнение: между сохранённым EBP и `var_C` вставлено восемь байтов, а под массив фактически отведено 76 вместо 64.

Обычно такой запас объясняется выравниванием и не влияет на корректную программу. Для разработчика эксплойта он важен: чтобы переполнение дошло до следующего значимого объекта, может потребоваться не 64, а как минимум 76 байтов.

Поиск по базе

Специализированные окна уже группируют имена, строки и импорты. В Search есть команды перехода к следующему элементу определённого типа, например Next Code. В Jump можно выбрать функцию или другую категорию из списка. Для общего поиска особенно важны текстовый и двоичный варианты.

Текстовый поиск

Search > Text или `Alt+T` ищет подстроку в текстовом листинге.

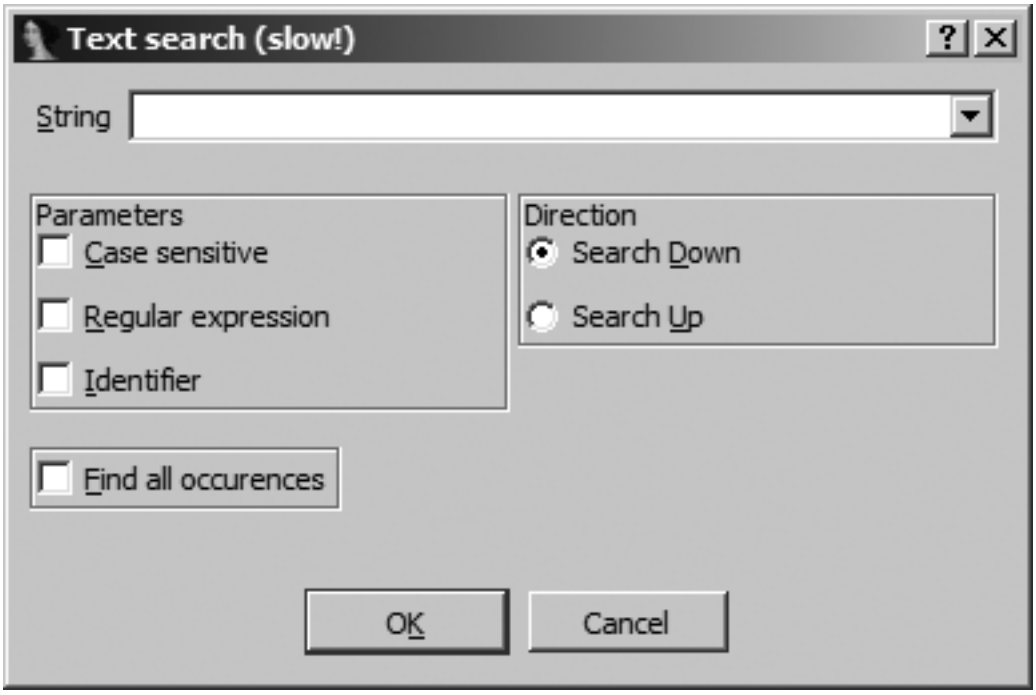

Рис. 6-7. Диалог Text Search

Поддерживаются регулярные выражения POSIX. Параметр Identifier фактически требует совпадения целого слова и способен находить не только имена, но и мнемоники или константы. Например, поиск Identifier по `401116` не найдёт `loc_401116`, потому что число является лишь частью слова.

Find all occurrences открывает отдельное окно всех результатов. `Ctrl+T` или Search > Next Text повторяет поиск и переходит к следующему совпадению.

Двоичный поиск

Для известной последовательности байтов нужен Search > Sequence of Bytes или `Alt+B`. Поиск идёт по содержимому Hex View, даже если команда вызвана из IDA View.

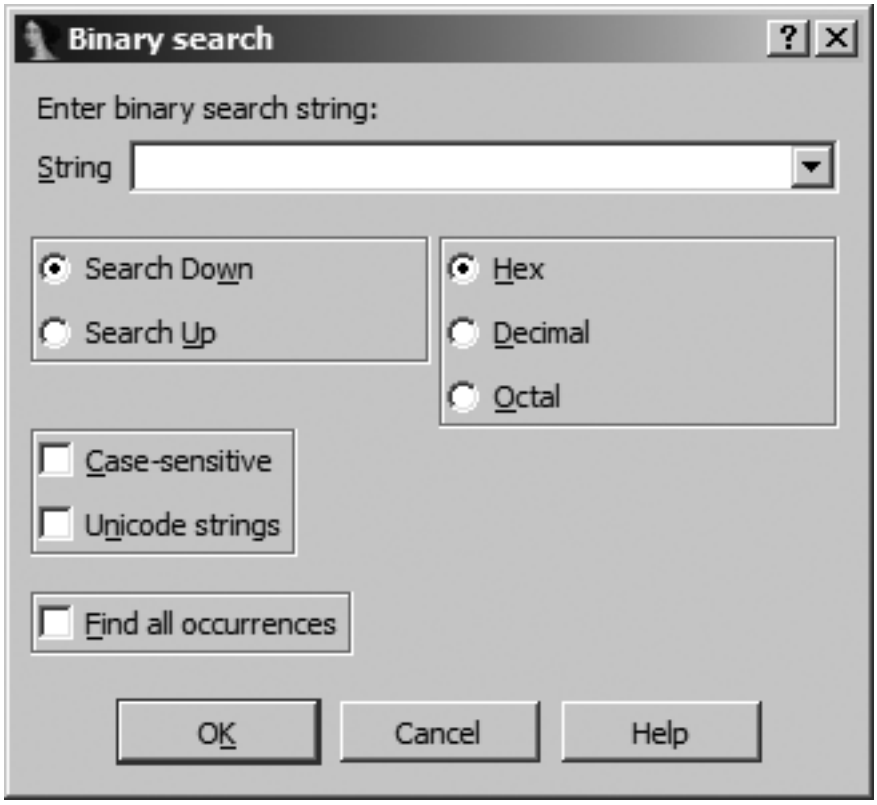

Рис. 6-8. Диалог Binary Search

Байты задаются парами шестнадцатеричных цифр через пробел, например `CA FE BA BE`. Регистр самих букв A-F значения не имеет. Чтобы искать ASCII-текст, строку заключают в кавычки. Unicode strings включает широкий вариант.

Case-sensitive для текста работает ожидаемо: без него `"hello"` совпадёт с `"HELLO"`. Для шестнадцатеричного шаблона эффект менее очевиден. Поиск `E9 41 C3` без учёта регистра может совпасть с `E9 61 C3`, поскольку `0x41` и `0x61` являются ASCII-буквами `A` и `a`.

При поиске точных опкодов обязательно включайте Case-sensitive. `Ctrl+B` или Search > Next Sequence of Bytes переходит к следующему результату.

Итоги

Описанные операции составляют основную часть повседневной навигации в IDA. Имена, перекрёстные ссылки, история, представления стека и поиск позволяют быстро перемещаться по логике программы, а не по строкам статического текста.

Следующая глава показывает, как изменять базу, добавляя знания, полученные при анализе содержимого и поведения файла.

Глава 7. Изменение дизассемблированного кода

После навигации важнейшая возможность IDA - изменение представления под задачу аналитика. Благодаря общей базе правка распространяется во все окна и сохраняет согласованную картину программы. IDA автоматически выполняет глобальную замену имён, преобразует инструкции в данные и обратно и позволяет менять формат операндов.

В рассматриваемой версии отмены нет. Следует часто сохранять базу, чтобы при необходимости вернуться к недавнему состоянию.

Имена и именование

До сих пор встречались имена виртуальных адресов и переменных кадра стека. Большинство из них IDA создаёт автоматически и называет dummy names. Такие обозначения вроде `sub_401000` или `var_5C` мало говорят о назначении объекта.

Одна из первых операций анализа - замена автоматических имён осмысленными. Обычно нужно выделить имя и нажать `N` либо выбрать Rename в контекстном меню. IDA сама заменит все связанные употребления.

Аргументы и локальные переменные

Имя стековой переменной действует только внутри одной функции и не связано с отдельным виртуальным адресом, поэтому его нет в Names. В разных функциях может существовать `arg_0`, но внутри одного кадра имена должны быть уникальны.

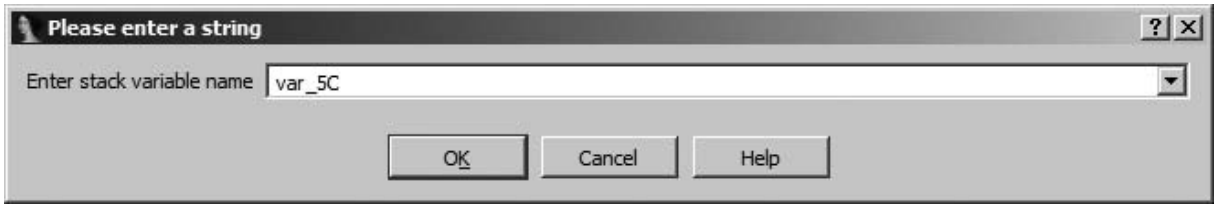

Рис. 7-1. Переименование переменной стека

После замены `var_5C` на y новое имя появляется в объявлении и во всех инструкциях функции:

```
.text:00401090 demo_stackframe proc near
.text:00401090 var_60 = dword ptr -60h
.text:00401090 y      = dword ptr -5Ch
.text:00401090 var_58 = byte ptr  -58h
.text:00401090 var_C  = dword ptr -0Ch
.text:00401090 arg_4  = dword ptr  0Ch
.text:00401090 arg_8  = dword ptr  10h
...
.text:0040109F        mov [ebp+y], eax
...
.text:004010AD        mov eax, [ebp+y]
```

Пустая строка в диалоге удаляет пользовательское имя и заставляет IDA восстановить стандартное.

Именованные адреса

Для адреса диалог N содержит больше параметров.

Рис. 7-2. Переименование адреса

Он показывает адрес и рекомендуемую максимальную длину из `<IDADIR>/cfg/ida.cfg`. Более длинное имя допустимо: IDA предупредит и предложит увеличить предел для текущей базы. Новые базы продолжат использовать значение конфигурационного файла.

Доступны следующие атрибуты.

Local name. Область действия ограничена функцией. Разные функции могут иметь одинаковые локальные имена, одна функция - нет. Адрес вне функции, имя функции и глобальная переменная локальными быть не могут. Чаще всего так именуют цели ветвлений внутри функции.

Include in names list. Добавляет объект в Names для быстрого поиска. Автоматические dummy names по умолчанию там отсутствуют.

Public name. Обозначает экспортируемый символ, обычно найденный загрузчиком в заголовке общей библиотеки. Ручная установка главным образом добавляет аннотацию `public` в листинг и Names.

Autogenerated name. Несмотря на название, сам по себе не заставляет IDA генерировать имя и почти не влияет на листинг.

Weak name. Слабый публичный символ используется, только если его не переопределил обычный публичный символ того же имени. Атрибут важен ассемблеру и почти не меняет анализ IDA.

Create name anyway. Глобальные адреса должны иметь уникальные имена во всей базе, локальные - внутри функции. При глобальном конфликте IDA в любом случае предлагает добавить числовой суффикс. Для локального имени без этого флажка конфликт просто отклоняется, а с ним

также создаётся уникальный суффикс.

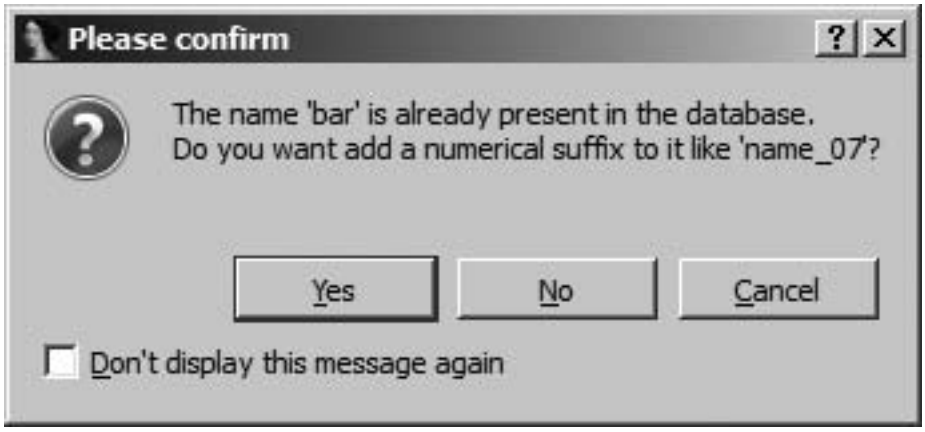

Рис. 7-3. Диалог конфликта имён

Проще всего выбрать действительно уникальное и содержательное имя.

Имена регистров

В пределах функции регистру можно назначить псевдоним. Это удобно, когда компилятор держит переменную не в стеке, а, например, в EDX. Команда `N` или Rename заменяет все употребления в функции и добавляет в начале запись вида `alias = register`.

Переименовать регистр в коде, который не принадлежит функции, нельзя.

Комментарии в IDA

Комментарии сохраняют ход исследования и объясняют последовательность инструкций на более высоком уровне. Например, C-подобная запись быстрее напомнит смысл функции, чем повторный анализ ассемблера.

Операции находятся в Edit > Comments, имеют горячие клавиши и контекстные меню. В большинстве случаев комментарий начинается с точки с запятой.

Обычные комментарии

Обычный комментарий стоит справа от существующей строки. Диалог открывается через правую часть листинга или клавишей `:`. Многострочный текст переносится на следующие строки с тем же отступом. Для изменения или удаления диалог открывают снова. Стандартный цвет - синий.

IDA сама добавляет такие комментарии к передаваемым аргументам, если знает прототип функции из библиотеки типов или введённых пользователем сведений.

Повторяемые комментарии

Repeatable comment вводится один раз, но автоматически показывается во всех местах, ссылающихся на адрес. Горячая клавиша - `;`, что легко перепутать с `:` обычного комментария.

У исходного адреса он отображается как обычный синий текст, у ссылок обычно серым. Если в месте ссылки создать обычный комментарий, он скроет повторяемый. После удаления обычного повторяемый снова станет виден.

Строковые переменные имеют виртуальный повторяемый комментарий с их содержимым. Его нельзя редактировать напрямую: каждое обращение к строке автоматически сопровождается текстом, например `"The first parameter is larger"`.

Строки до и после инструкции

Anterior и posterior - полноширинные комментарии непосредственно перед или после выбранной строки. Только они не получают префикс `;`. Отличить их можно по адресу относительно соседней инструкции.

Комментарии функции

Комментарий функции группируется в её заголовке. Нужно выделить имя в начале функции и добавить обычный либо повторяемый комментарий. Повторяемый вариант виден также у каждого вызова. Команда Set Function Type из главы 8 автоматически создаёт комментарий с прототипом.

Базовые преобразования кода

Для обычной программы автоматического результата часто достаточно. Чем сильнее файл отличается от стандартного вывода известного компилятора, тем больше требуется ручной работы. Это особенно верно для обфускации и неизвестных форматов.

IDA позволяет:

• преобразовывать данные в код и код в данные;
• объявлять последовательность инструкций функцией;
• исправлять начало и конец функции;
• менять формат операндов.

Настройка строк листинга

Каждая строка состоит из метки, мнемоники, операндов и дополнительных частей. Постоянные элементы скрыть нельзя, а остальные выбираются на вкладке Disassembly в Options > General.

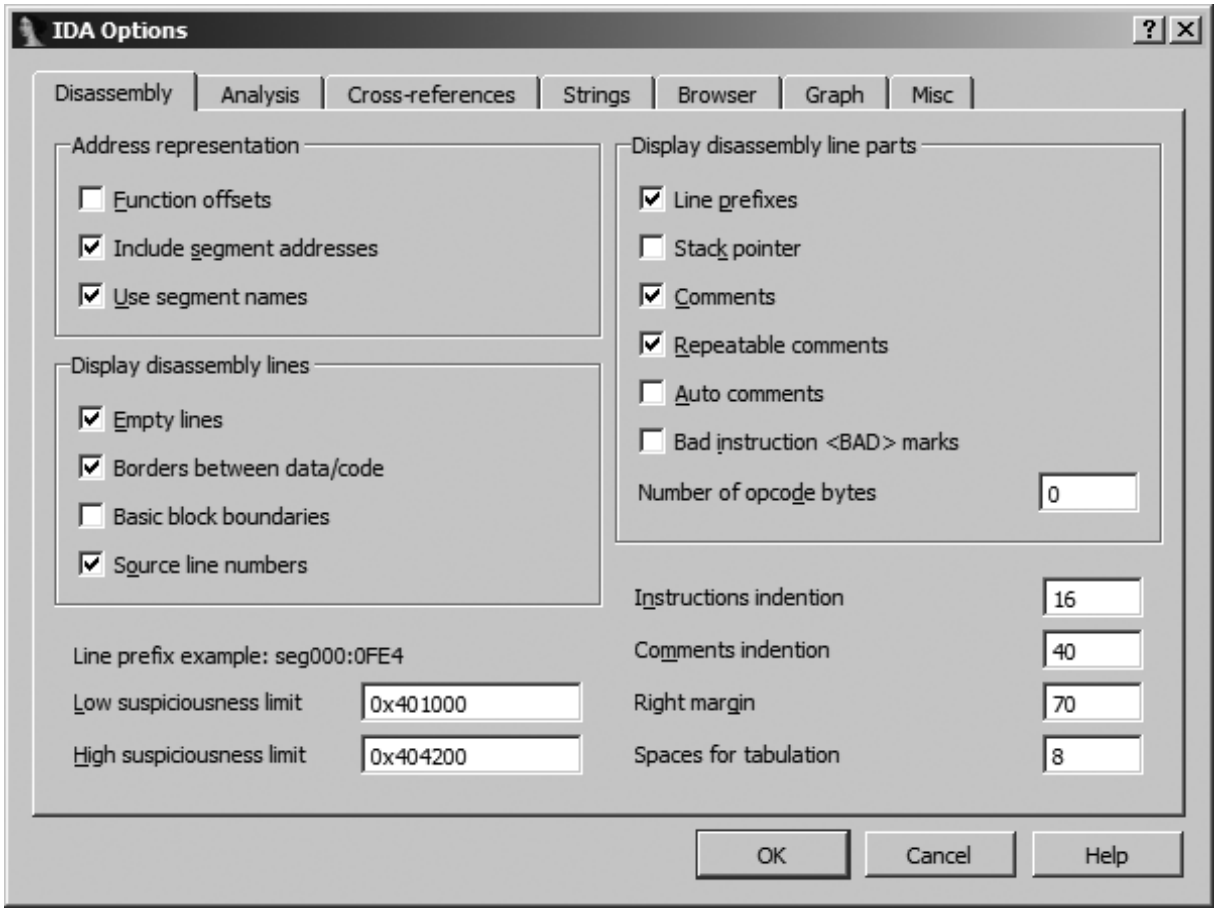

Рис. 7-4. Настройка строк дизассемблирования

Line prefixes показывает `секция:адрес`. В текстовом виде включено по умолчанию, в графе обычно выключено.

Stack pointer добавляет относительное изменение указателя стека в каждой точке функции. Это помогает обнаружить неверно распознанное соглашение о вызове или необычную работу со стеком. Если у `ret` значение не вернулось к нулю, IDA отмечает строку красным. Причиной может быть обфускация либо непонятный IDA пролог или эпилог.

Comments и repeatable comments управляют показом соответствующих комментариев и позволяют освободить место на экране.

Auto comments добавляет пояснения к некоторым нетривиальным инструкциям, например `Integer Subtraction`, `Call Procedure` и `High Level Procedure Exit`. Пользовательский комментарий имеет приоритет.

Bad instruction `<BAD>` **marks** показывает допустимую для процессора, но неизвестную некоторым ассемблерам инструкцию как байты с комментарием `<BAD>`. Так итоговый листинг легче повторно собрать.

Number of opcode bytes помещает машинные байты рядом с ассемблером. Для архитектуры фиксированной длины настройка проста. В x86 IDA резервирует одинаковую ширину, а слишком длинную инструкцию сокращает знаком +.

```
000 55                push  ebp
004 89 E5             mov   ebp, esp
004 83 EC 78          sub   esp, 78h  ; Integer Subtraction
07C 8B 45 10          mov   eax, [ebp+arg_8]
07C C7 45 A0 0A 00+   mov   [ebp+var_60], 0Ah
07C E8 91 FF FF FF     call  bar       ; Call Procedure
07C C9                leave           ; High Level Procedure Exit
```

```
000 C3                retn            ; Return Near from Procedure
```

Отступы инструкций, комментариев, правое поле и ширина табуляции также задаются в диалоге. Изменения действуют в текущей базе; глобальные значения хранятся в `ida.cfg`.

Форматирование операндов

Константа может быть смещением, адресом, арифметическим значением или именованной константой исходного кода. IDA старается использовать символы, когда контекст ясен, иначе обычно выбирает шестнадцатеричный формат.

Контекстное меню числа предлагает десятичное, восьмеричное, двоичное и, для печатного диапазона, символьное представление. Рядом сразу показан будущий текст.

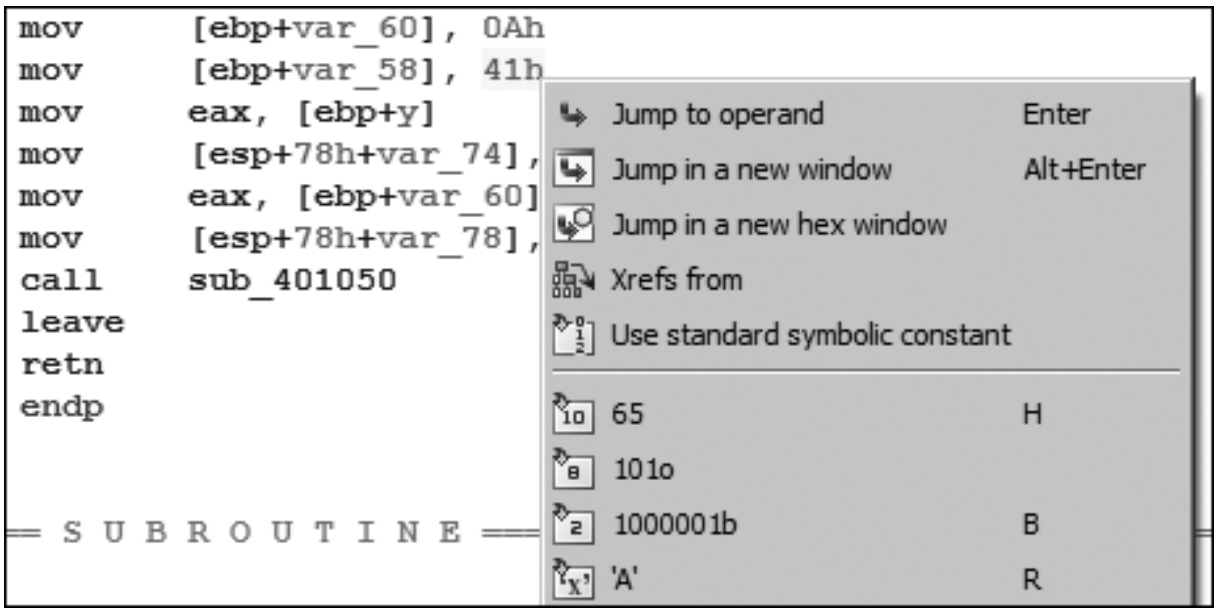

Рис. 7-5. Форматы константы

После компиляции невозможно понять, использовал ли автор литерал, `#define` или элемент enum. IDA содержит каталог стандартных символических констант C и Windows API. Use standard symbolic constant фильтрует его по текущему значению.

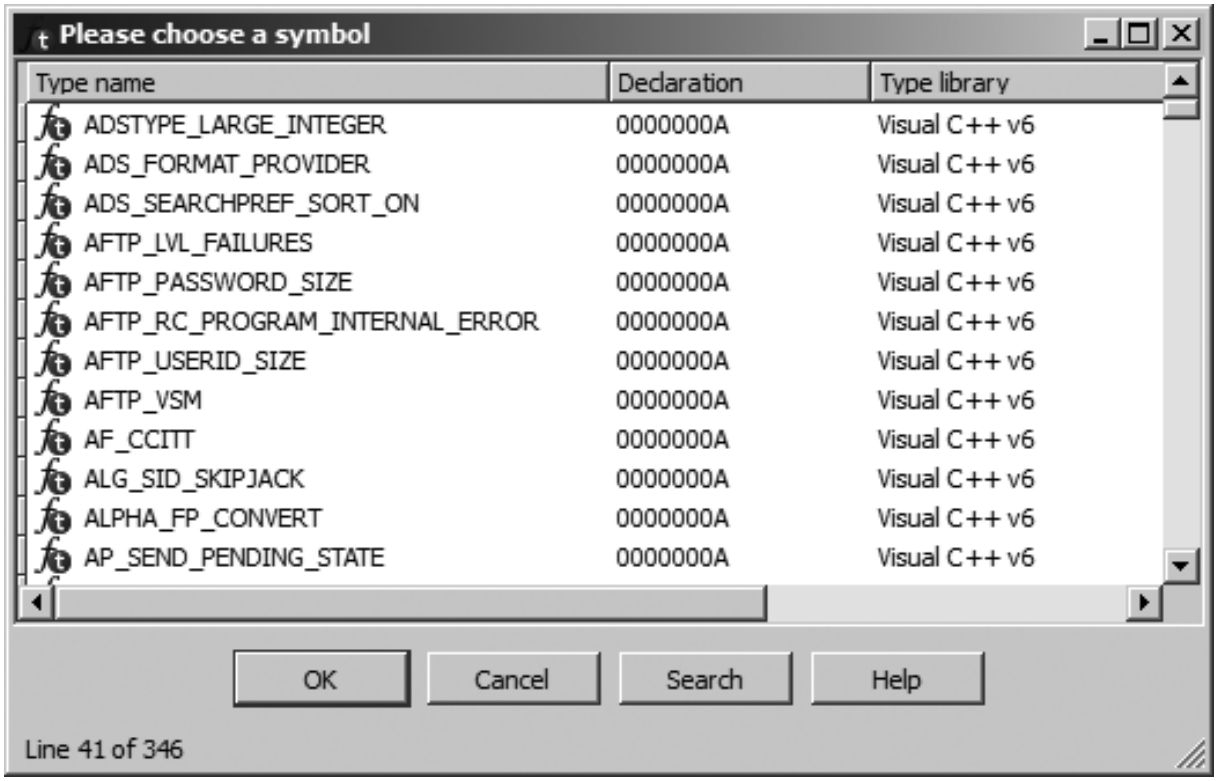

Рис. 7-6. Выбор символической константы

Если `0Ah` обозначает семейство сетевых адресов X.25, можно выбрать `AF_CCITT`:

```
.text:004010A2  mov [ebp+var_60], AF_CCITT
```

Такой список экономит поиск возможного имени в документации API.

Изменение функций

IDA может пропустить функцию без очевидного вызова, неверно определить её конец или столкнуться с оптимизацией, когда общие завершающие блоки объединены.

Создание и удаление

Чтобы создать функцию, курсор ставят на первый байт или инструкцию и выбирают Edit > Functions > Create Function. IDA при необходимости превращает данные в код, ищет `ret`, создаёт имя и анализирует кадр. Если конец найти нельзя или встречена недопустимая инструкция, операция завершается неудачно.

Edit > Functions > Delete Function удаляет ошибочно созданную функцию.

Фрагменты функций

Microsoft Visual C++ может переносить редко выполняемые блоки подальше, чтобы часто исполняемая часть лучше помещалась на страницах памяти. IDA называет отдельные диапазоны function chunks, а меню - function tails.

```
.text:004037AE ChunkedFunc proc near
.text:004037AE ; FUNCTION CHUNK AT .text:004040D7 SIZE 00000011 BYTES
```

```
.text:004037AE ; FUNCTION CHUNK AT .text:004129ED SIZE 0000000A BYTES
.text:004037AE ; FUNCTION CHUNK AT .text:00413DBC SIZE 00000019 BYTES
```

Двойной щелчок по адресу переходит к фрагменту:

```
.text:004040D7 ; START OF FUNCTION CHUNK FOR ChunkedFunc
.text:004040D7 loc_0040C0D7:
.text:004040D7     dec  eax
.text:004040D8     jnz  loc_403836
.text:004040DE     call sub_4040ED
.text:004040E3     jmp  loc_403836
.text:004040E3 ; END OF FUNCTION CHUNK FOR ChunkedFunc
```

Если IDA пропустила часть, нужно выделить свободный диапазон и выбрать Edit > Functions > Append Function Tail, затем указать родительскую функцию. Remove Function Tail удаляет фрагмент под курсором.

Создание tails можно выключить в Kernel Options при загрузке. Тогда функция содержит красные переходы за свои границы, а цели не показываются в её графе.

Атрибуты функции

Edit function позволяет исправить основные свойства.

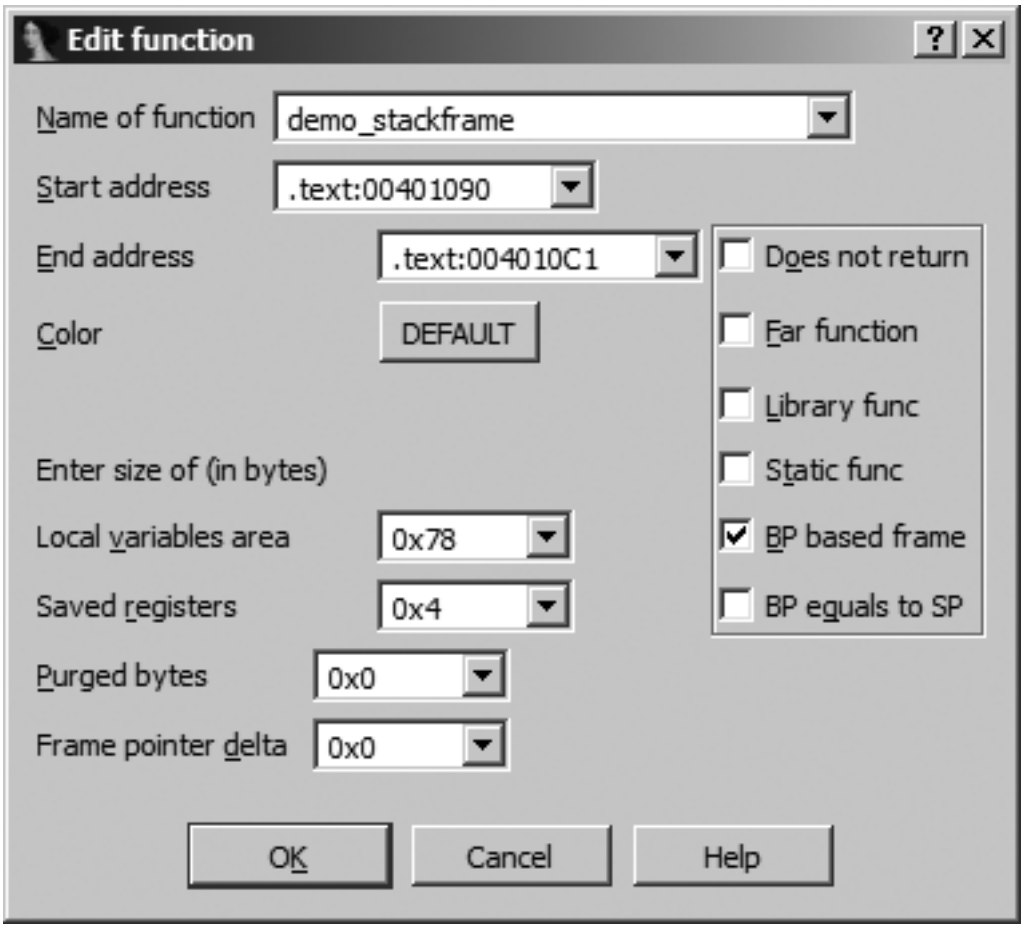

Рис. 7-7. Диалог изменения функции

• **Name of function** меняет имя.
• **Start address** задаёт первую инструкцию.
• **End address** задаёт адрес сразу после последней инструкции, а не саму последнюю инструкцию.
• **Local variables area** определяет число байтов локальных данных.

• **Saved registers** определяет область регистров над адресом возврата. Если компилятор сохраняет их выше локальных данных, IDA может включить размер в local variables.
• **Purged bytes** показывает объём аргументов, удаляемых функцией. Для `cdecl` это ноль; для `stdcall` IDA часто выводит число из `ret N`.
• **Frame pointer delta** задаёт расстояние от изменённого указателя кадра до сохранённого. Оптимизатор может сдвинуть основание внутрь локальной области, чтобы больше переменных попадало в диапазон однобайтового смещения от -128 до +127.

Флажки уточняют характер функции:

• **Does not return** сообщает, что после вызова нет обычного продолжения;
• **Far function** указывает дальний вызов с сегментом и смещением;
• **Library func** применяет цвет библиотечного кода;
• **Static func** только добавляет атрибут `static`;
• **BP based frame** включает отдельный указатель кадра и символические стековые переменные. При ручной установке нужно согласовать размеры сохранённых регистров и локальной области;
• **BP equals to SP** означает, что указатель кадра при входе указывает на вершину локальной области, то есть delta равна её размеру.

Ручная поправка указателя стека

Точность кадра зависит от правильного отслеживания SP. Если IDA не знает, что импортированная `stdcall`-функция удаляет аргументы, после вызова все последующие смещения будут неверны:

```
.text:004010EB  01C push eax
.text:004010F3  020 push 2
.text:004010FB  024 push 1
.text:00401102  028 call some_imported_func
.text:00401107  028 mov  ebx, eax  ; должно быть 01C
```

На адресе вызова Edit > Functions > Change Stack Pointer или `Alt+K` позволяет указать изменение 12 байтов.

Если функция вызывается много раз, лучше перейти к её записи импорта и установить Purged bytes. IDA распространит это знание на все вызовы. Встроенный анализ также решает систему линейных уравнений для устранения некоторых расхождений автоматически.

Преобразование данных и кода

Автоанализ иногда принимает встроенные данные за инструкции или пропускает неочевидный код. Обфускация намеренно размывает границу.

Сначала существующее определение можно снять командой Undefine, Edit > Undefine или клавишей `U`. Подлежащие байты становятся необработанными `db`. Для диапазона его предварительно выделяют.

```
; исходная функция
.text:004013E0 sub_4013E0 proc near
.text:004013E0     push ebp
.text:004013E1     mov  ebp, esp
.text:004013E3     pop  ebp
.text:004013E4     retn

; после Undefine
.text:004013E0 unk_4013E0 db 55h
.text:004013E1             db 89h
.text:004013E2             db 0E5h
```

```
.text:004013E3             db 5Dh
.text:004013E4             db 0C3h
```

Клавиша `C` или Edit > Code дизассемблирует неопределённые байты до уже размеченного объекта или недопустимой инструкции. Для большого диапазона также можно использовать выделение.

Код в данные через контекстное меню не преобразуется. Нужно использовать Edit > Data или `D`; для массового преобразования удобнее сначала выполнить Undefine.

Базовые преобразования данных

Корректные типы данных столь же важны, как корректные инструкции. IDA выводит размер несколькими способами:

1. Загрузка памяти в 32-разрядный регистр подразумевает объект размером четыре байта, хотя это может быть и число, и указатель.
2. Прототип функции задаёт тип параметра. Если известная функция принимает `CRITICAL_SECTION*`, IDA может разметить переданный адрес как структуру Windows `CRITICAL_SECTION`.
3. Содержимое само подсказывает тип: длинная последовательность печатных байтов похожа на строку.

Размеры данных

Наиболее частые директивы: `db` для 1 байта, `dw` для 2, `dd` для 4. Options > Setup Data Types содержит кнопки немедленного преобразования слева и список data carousel справа.

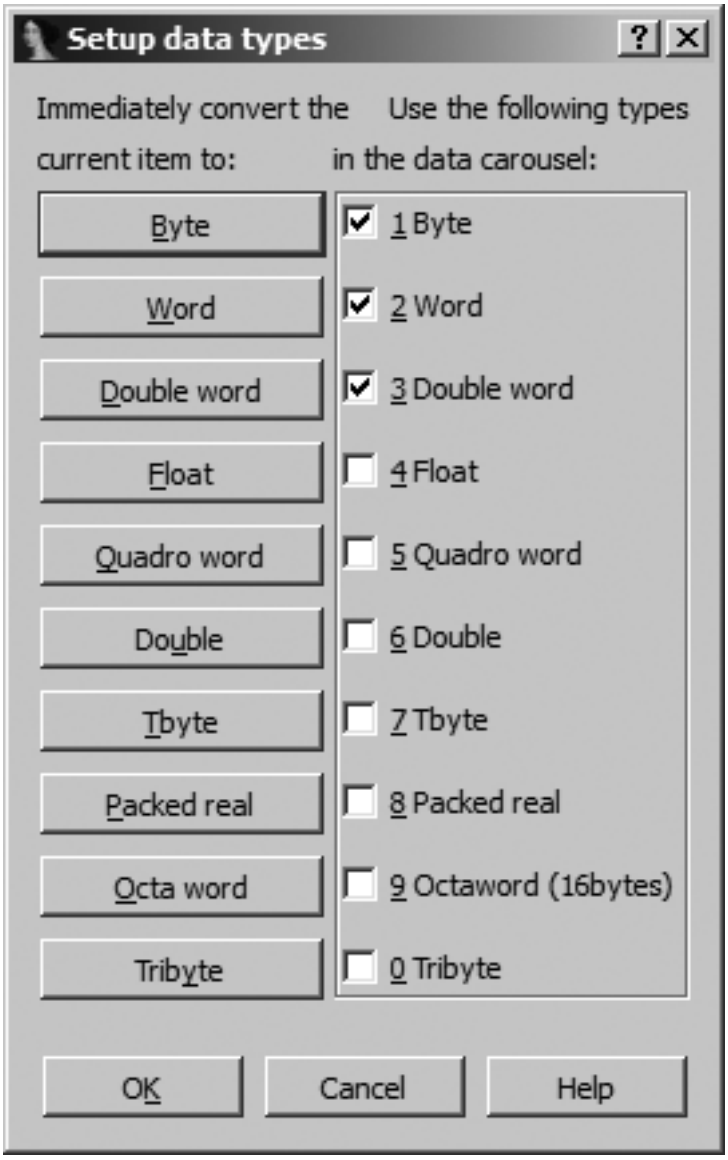

Рис. 7-8. Диалог Setup Data Types

Отмеченные типы появляются в контекстном меню. Клавиша `D` циклически переключает текущий объект между ними. Для списка byte, word, double word последовательность будет `db -> dw -> dd -> db`. Код при первом нажатии превращается в первый тип карусели.

При уменьшении лишние байты становятся неопределёнными. При увеличении IDA спрашивает `Directly convert to data?`, то есть разрешить ли снять разметку со следующих объектов и поглотить их. Те же операции применимы к переменным стека через подробный вид кадра.

Работа со строками

IDA распознаёт много форматов и по умолчанию создаёт C-строки с нулём. Edit > Strings выбирает конкретный стиль, а `A` использует текущий стандартный. Преобразование выполняется, только если байты действительно соответствуют формату.

Options > ASCII String Style, несмотря на название, содержит C, DOS, Pascal, Delphi, Unicode и варианты Pascal Unicode. Кнопка слева немедленно создаёт строку, переключатель справа назначает формат для `A`. Для строк с завершающим символом можно задать до двух терминаторов.

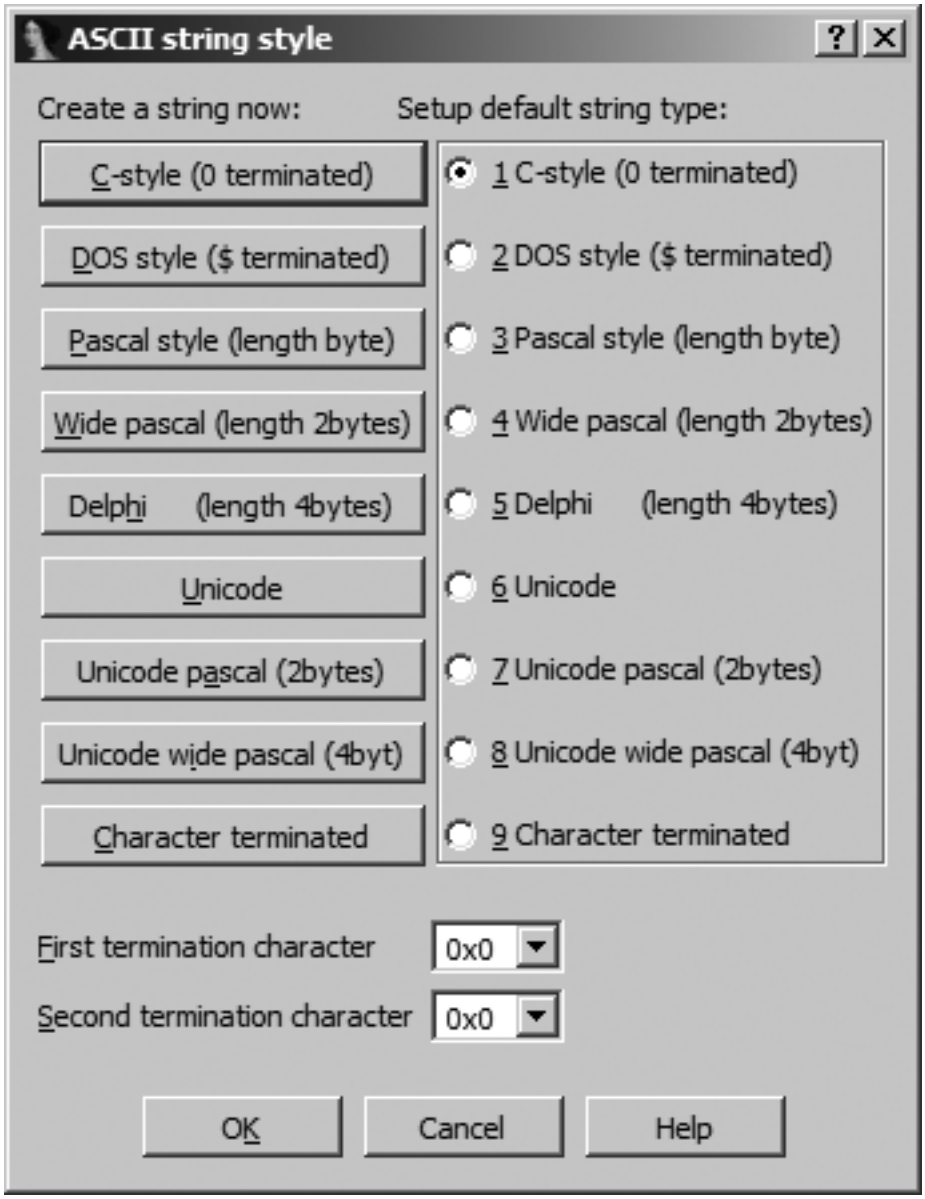

Рис. 7-9. Настройка строковых данных

Вторая группа находится на вкладке Strings в Options > General.

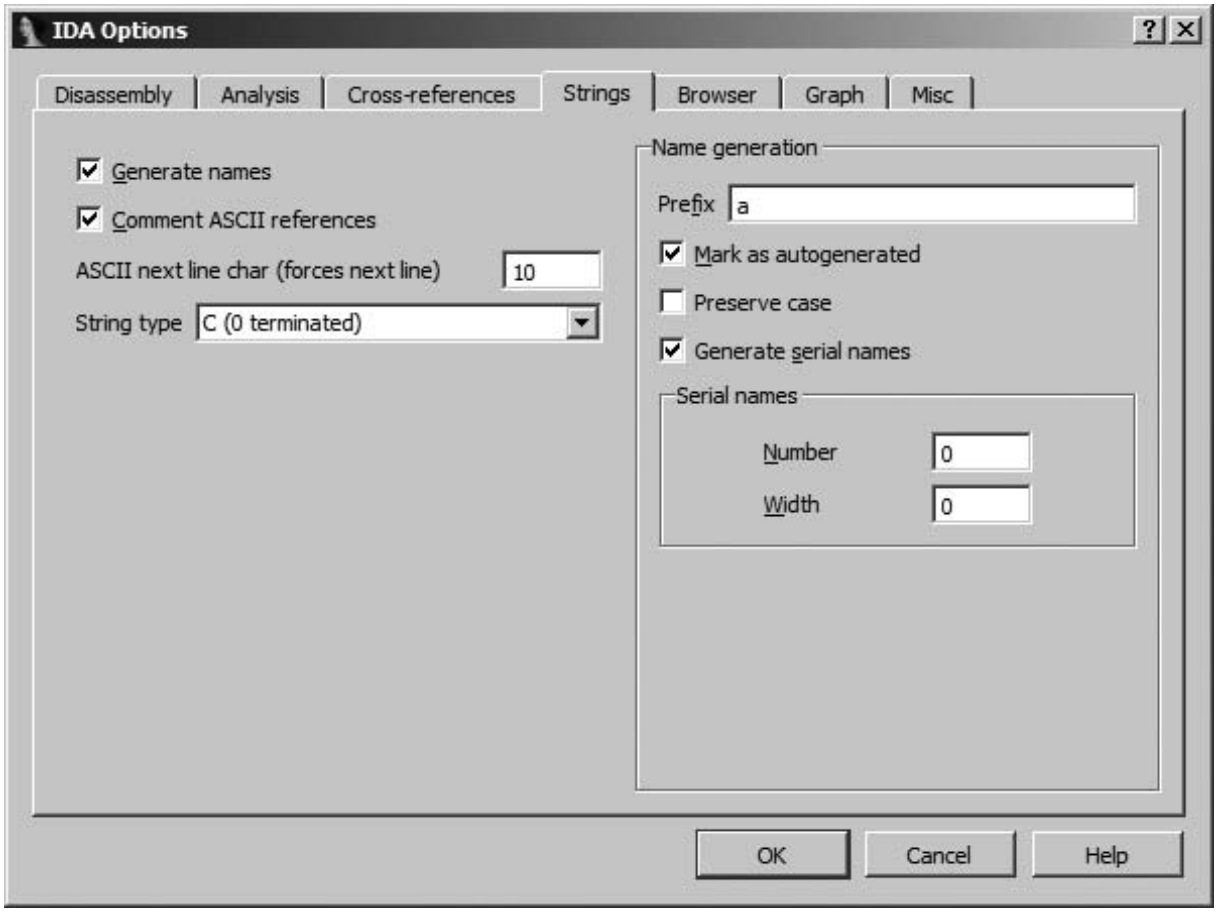

Рис. 7-10. Параметры Strings

Без Generate names строка получает dummy name с `asc_`. При включённой генерации префикс соединяется с допустимыми символами содержимого до максимальной длины:

```
.rdata:00402069 aThisIsACharact db 'This is a Character array',0
```

Пробелы и недопустимые знаки пропускаются, обычно применяется Title Case. Mark as autogenerated выделяет такое имя отдельным цветом. Preserve case сохраняет регистр исходной строки. Generate serial names создаёт числовые суффиксы; при Number `0` и Width `3` получаются `a000`, `a001`, `a002`.

Массивы

В листинге трудно определить размер исходного массива, а один элемент на строку занимает много места. Если инструкции ссылаются только на первый адрес, а остальные элементы вычисляют индексом, последовательность, вероятно, является массивом.

Сначала первый объект должен иметь правильный размер элемента. Затем Edit > Array или Array в контекстном меню открывает диалог.

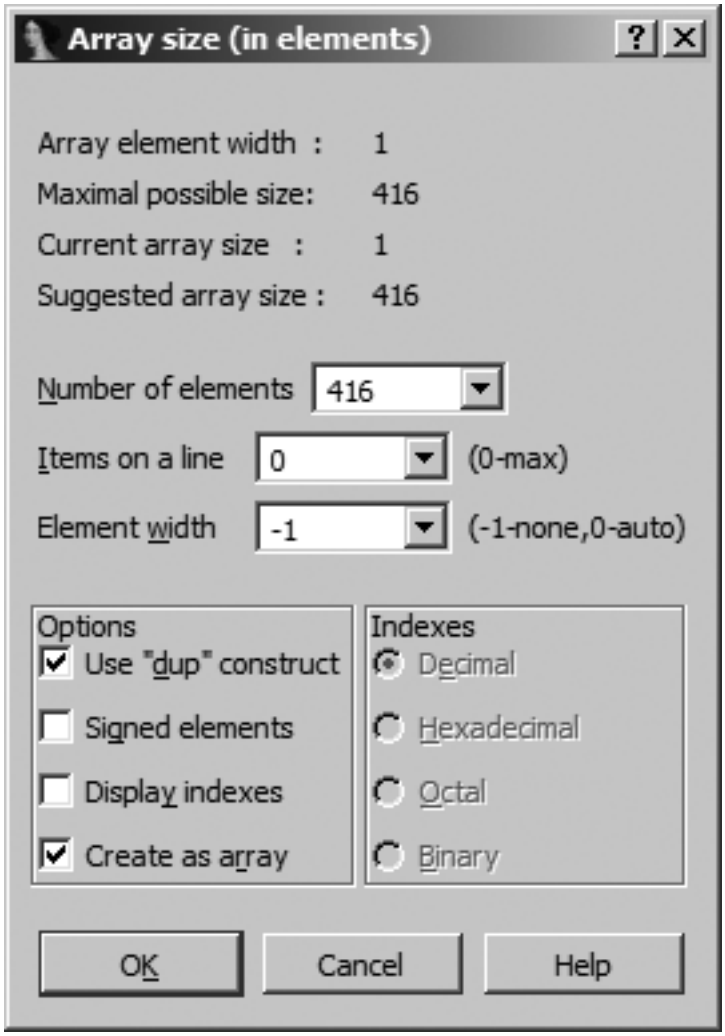

Рис. 7-11. Диалог создания массива

Поля имеют следующий смысл:

• **Array element width** - размер элемента, унаследованный от текущего `db`, `dw`, `dd` и так далее;
• **Maximum possible size** - число элементов до следующего уже определённого объекта;
• **Number of elements** - точный размер массива; общий объём равен числу элементов, умноженному на ширину;
• **Items on a line** - число элементов в строке листинга;
• **Element width** - только ширина столбца при форматировании;
• **Use** `dup` **construct** объединяет одинаковые значения;
• **Signed elements** выбирает знаковое представление;
• **Display indexes** добавляет индексы комментариями и позволяет выбрать систему счисления;
• **Create as array** группирует элементы в массив. Если снять флажок, будет создано указанное число отдельных объектов.

Настройки на рисунке превращают 416 строк с нулями в одну:

```
.rdata:00402060 byte_402060 db 1A0h dup(0)
```

Массивам внутри кадров стека посвящена следующая глава.

Итоги

Навигация и преобразования базы составляют основную часть повседневной работы в IDA. Осмысленные имена, точные типы и подробные комментарии объединяют вывод автоматического анализа со знаниями пользователя. Они помогают и самому автору исследования, и тем, кто продолжит работу позднее.

Следующая глава переходит к структурам C, сложным типам и низкоуровневым особенностям скомпилированного C++.

Глава 8. Типы и структуры данных

Проще всего начать понимание двоичной программы с перечня вызываемых библиотечных функций. Вызов `connect` указывает на сетевое соединение, `RegOpenKey` - на обращение к реестру Windows. Но чтобы понять, как и зачем они вызываются, необходимо исследовать аргументы.

Для `connect` важно знать конкретный сетевой адрес. Число, тип и порядок аргументов образуют сигнатуру функции. Поэтому аналитику нужно понимать, как типы и структуры представлены на уровне машинных инструкций.

IDA выводит типы из известных прототипов. Вызов `connect` может быть оформлен так:

```
.text:004010F3  push 10h          ; namelen
.text:004010F5  lea  ecx, [ebp+name]
.text:004010F8  push ecx          ; name
.text:004010F9  mov  edx, [ebp+s]
.text:004010FF  push edx          ; s
.text:00401100  call connect
```

Названия параметров взяты из прототипа, а локальные значения автоматически переименованы в `name` и `s`.

Распространение типов работает не только для библиотек. Если пользователь задаст прототип любой функции через Edit > Functions > Set Function Type, контекстное меню или `Y`, IDA применит сведения к ней и всем вызовам.

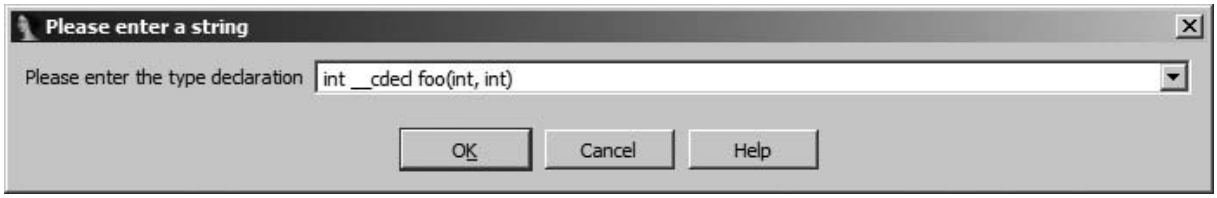

Рис. 8-1. Установка типа функции

Для неизвестной функции IDA обычно предполагает возврат `int`, аргументы `int` и выводит соглашение по эпилогу. После ввода

```
int __cdecl foo(float f, char *ptr)
```

листинг получает комментарий-прототип и новые имена:

```
.text:00401050 ; int __cdecl foo(float f, char *ptr)
.text:00401050 foo proc near
.text:00401050 f   = dword ptr 8
.text:00401050 ptr = dword ptr 0Ch
```

У вызывающей функции IDA подписывает аргументы и способна переименовать связанные локальные переменные:

```
.text:004010AD  mov eax, [ebp+ptr]
.text:004010B0  mov [esp+4], eax  ; ptr
.text:004010B4  mov eax, [ebp+f]
.text:004010B7  mov [esp], eax    ; f
.text:004010BA  call foo
```

Прототип импортированной функции часто виден во всплывающей подсказке. Если IDA его не знает, следует обратиться к документации API и справочным страницам.

Распознавание сложных данных

Примитивный тип часто совпадает с размером регистра или операнда. Массивы и структуры требуют вычислять адрес отдельного элемента, и характер этих вычислений даёт подсказки.

Доступ к элементам массива

Массив - непрерывная последовательность однотипных элементов. Размер равен числу элементов, умноженному на размер каждого:

```
int array_demo[100];
size_t bytes = 100 * sizeof(int);
```

При размере `int` четыре байта `array_demo[20]` находится на смещении 80. Константу компилятор вычисляет заранее. Для `array_demo[i]` значение `i * 4` обычно вычисляется во время исполнения.

Глобальные массивы

Адрес глобального массива известен при компиляции:

```
int global_array[3];

int main() {
    int idx = 2;
    global_array[0] = 10;
    global_array[1] = 20;
    global_array[2] = 30;
    global_array[idx] = 40;
}
```

```
.text:0040100B  mov dword_40B720, 10
.text:00401015  mov dword_40B724, 20
.text:0040101F  mov dword_40B728, 30
.text:00401029  mov eax, [ebp+idx]
.text:0040102C  mov dword_40B720[eax*4], 40
```

Три константных обращения выглядят как три глобальные переменные. Только масштаб `eax*4` указывает на массив, его начало и четырёхбайтовый элемент. После Edit > Array IDA показывает:

```
mov dword_40B720,   10
mov dword_40B720+4, 20
mov dword_40B720+8, 30
```

Константные индексы скрывают массив, переменный индекс раскрывает базу и размер элемента.

Массивы в стеке

Абсолютный адрес неизвестен, но смещение от EBP постоянно:

```
int stack_array[3];
int idx = 2;
stack_array[0] = 10;
stack_array[1] = 20;
stack_array[2] = 30;
stack_array[idx] = 40;
```

```
.text:00401000 var_10 = dword ptr -10h
.text:00401000 var_C  = dword ptr -0Ch
.text:00401000 var_8  = dword ptr -8
.text:00401000 idx    = dword ptr -4
...
.text:0040100D  mov [ebp+var_10], 10
.text:00401014  mov [ebp+var_C],  20
```

```
.text:0040101B  mov [ebp+var_8],  30
.text:00401022  mov eax, [ebp+idx]
.text:00401025  mov [ebp+eax*4+var_10], 40
```

Опять видны три отдельные переменные. Компилятор упростил `ebp - 0x10 + 4` до `ebp - 0x0C`. Лишь индексированная строка показывает начало и ширину массива. Положение следующей переменной `idx` позволяет оценить максимум в три элемента, что особенно полезно при анализе переполнения.

Массивы в куче

Динамический массив адресуется через указатель, возвращённый `malloc` или `new`:

```
int *heap_array = (int*)malloc(3 * sizeof(int));

push 0Ch                    ; общий размер
call _malloc
mov  [ebp+heap_array], eax
mov  eax, [ebp+heap_array]
mov  dword ptr [eax],   10
mov  ecx, [ebp+heap_array]
mov  dword ptr [ecx+4], 20
mov  edx, [ebp+heap_array]
mov  dword ptr [edx+8], 30
mov  eax, [ebp+idx]
mov  ecx, [ebp+heap_array]
mov  dword ptr [ecx+eax*4], 40
```

Каждый доступ сначала читает базовый адрес. Параметр `malloc` даёт общий объём 12 байт, смещения и масштаб показывают ширину 4, следовательно выделено три элемента.

Надёжнее всего массив узнаётся по переменному индексу, масштабированному на размер элемента. Константные обращения почти неотличимы от полей структуры.

Доступ к полям структуры

Структура C объединяет разнотипные поля, выбираемые по имени. Компилятор заменяет имена числовыми смещениями, поэтому результат похож на массив с постоянными индексами.

```
struct ch8_struct {     // размер   min offset   default offset
    int field1;         // 4        0            0
    short field2;       // 2        4            4
    char field3;        // 1        6            6
    int field4;         // 4        7            8
    double field5;      // 8        11           16
};                      // минимум 19, обычно 24 байта
```

Сумма полей равна 19, но компилятор по умолчанию выравнивает 4-байтовые значения на смещения, кратные 4, и 8-байтовые на кратные 8. Для этого вставляется padding, и размер становится 24.

`#pragma pack` в Microsoft Visual C/C++ и gcc/g++ управляет выравниванием; gcc также поддерживает атрибут `packed`. Выравнивание в один байт даёт минимальные смещения. На некоторых процессорах невыровненный доступ медленнее или вызывает исключение.

Глобальные структуры

Для глобального объекта все адреса вычисляются заранее:

```
mov  dword_40EA60, 10
mov  word_40EA64,  20
mov  byte_40EA66,  30
mov  dword_40EA68, 40
fld  ds:dbl_40B128
fstp dbl_40EA70
```

Листинг выглядит как пять глобальных переменных. Без исходника уверенно доказать наличие структуры нельзя.

Структуры в стеке

Смещения от EBP также известны заранее:

```
.text:00401000 var_18 = dword ptr -18h
.text:00401000 var_14 = word ptr  -14h
.text:00401000 var_12 = byte ptr  -12h
.text:00401000 var_10 = dword ptr -10h
.text:00401000 var_8  = qword ptr -8
...
mov  [ebp+var_18], 10
mov  [ebp+var_14], 20
mov  [ebp+var_12], 30
mov  [ebp+var_10], 40
fld  ds:dbl_40B128
fstp [ebp+var_8]
```

IDA снова показывает пять локальных переменных, хотя `var_18` является началом единого 24-байтового объекта.

Структуры в куче

Динамический объект наиболее информативен, потому что каждое поле адресуется относительно неизвестного заранее указателя:

```
push 24                 ; sizeof(ch8_struct)
call _malloc
mov  [ebp+heap_struct], eax
mov  eax, [ebp+heap_struct]
mov  dword ptr [eax],    10  ; offset 0, size 4
mov  ecx, [ebp+heap_struct]
mov  word ptr [ecx+4],   20  ; offset 4, size 2
mov  edx, [ebp+heap_struct]
mov  byte ptr [edx+6],   30  ; offset 6, size 1
mov  eax, [ebp+heap_struct]
mov  dword ptr [eax+8],  40  ; offset 8, size 4
mov  ecx, [ebp+heap_struct]
fld  ds:dbl_40B128
fstp qword ptr [ecx+10h]     ; offset 16, double
```

Параметр `malloc` даёт размер 24, а инструкции раскрывают поля. `fld` и `fstp` позволяют уточнить, что восьмибайтовое поле является `double`.

В упакованном варианте `malloc` получает 19, `field4` имеет смещение 7, `field5` - `0x0B`. На практике отдельные функции редко обращаются сразу ко всем полям. Нужно отслеживать указатель по программе и собирать смещения до восстановления полной схемы.

Массивы структур

Сложные типы можно вкладывать. Для массива из пяти `ch8_struct`:

```
int idx = 1;
struct ch8_struct *heap_struct;
heap_struct = (struct ch8_struct*)malloc(sizeof(struct ch8_struct) * 5);
heap_struct[idx].field1 = 10;

push 120
call _malloc
mov  [ebp+heap_struct], eax
mov  eax, [ebp+idx]
imul eax, 24
mov  ecx, [ebp+heap_struct]
mov  dword ptr [ecx+eax], 10
```

Общий размер 120 и масштаб 24 дают пять элементов. Отсутствие дополнительного смещения означает четырёхбайтовое поле в начале каждого. О прочих 20 байтах этот фрагмент ничего не говорит.

Создание структур в IDA

Цель ручной разметки - заменить `[edx+10h]` на `[edx+ch8_struct.field5]`. Известные стандартные структуры IDA иногда применяет сама, пользовательские нужно описать.

Новая структура или union

Все применимые типы должны присутствовать в Structures.

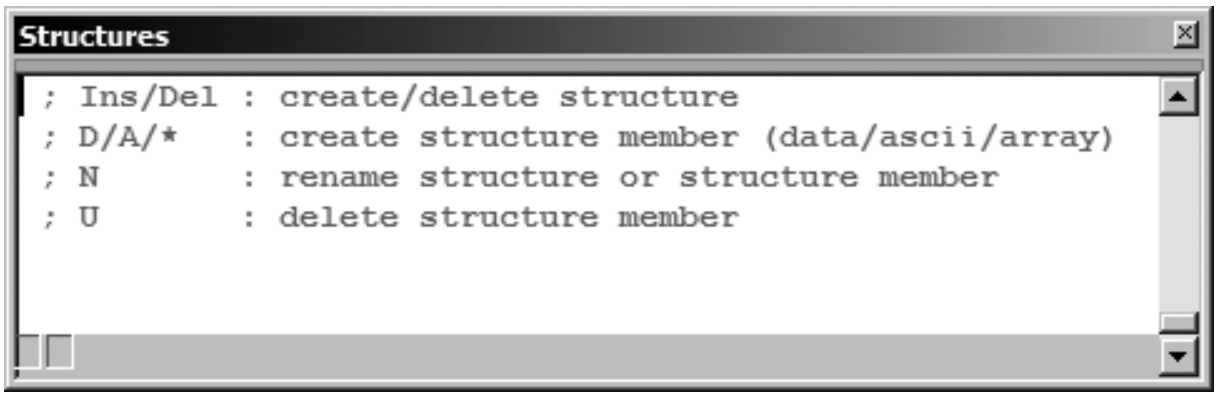

Рис. 8-2. Окно Structures

IDA могла не добавить тип, потому что не распознала его применение или вообще не знает нестандартную схему. В обоих случаях работа начинается здесь. `Insert` открывает диалог создания.

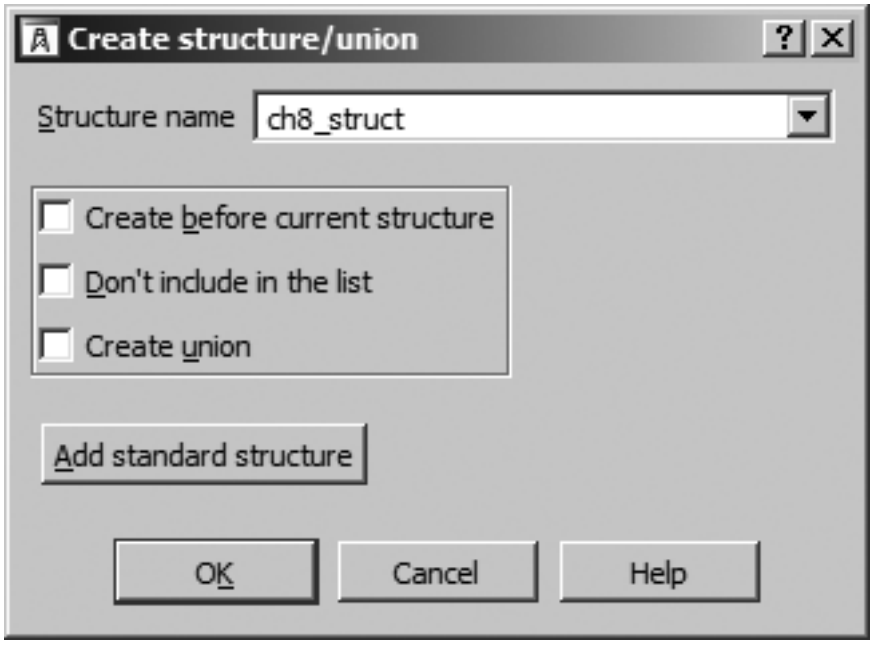

Рис. 8-3. Диалог Create Structure/Union

Нужно указать имя. Первые флажки определяют расположение в списке, Create union создаёт C union. Размер структуры равен сумме полей с заполнением, размер union - максимальному полю, поскольку все они перекрываются. Add standard structure выбирает известный тип.

После подтверждения появляется пустое определение.

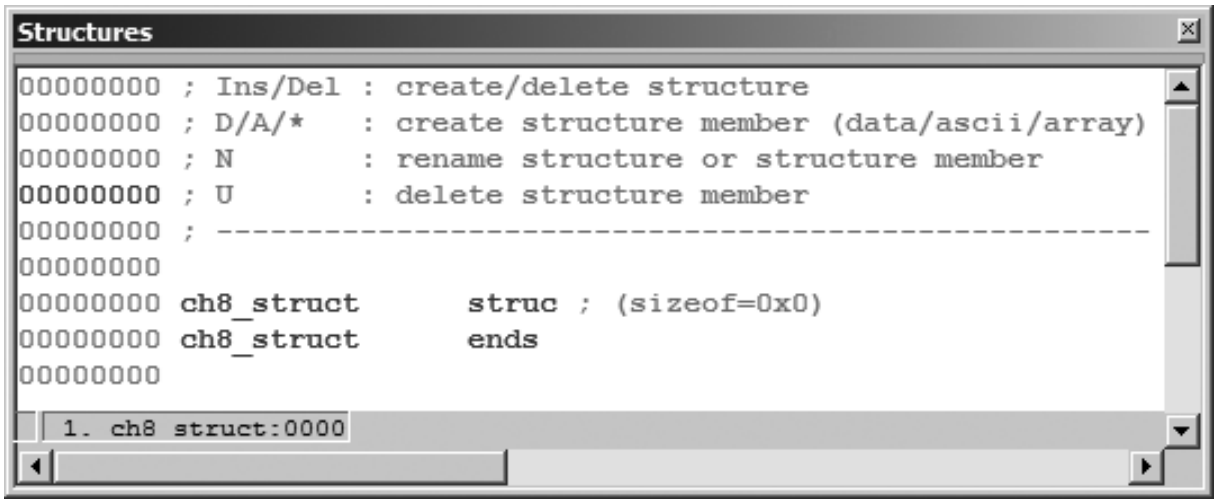

Рис. 8-4. Пустая структура

Редактирование полей

Рекомендуемая последовательность:

1. Поставить курсор на строку `ends` и нажать `D`. Добавится поле типа, стоящего первым в data carousel, с именем `field_N`, где `N` - смещение.
2. На имени поля повторять `D` для смены размера или выбрать тип через Options > Setup Data Types. Для массива использовать Array.
3. Нажать `N` и дать полю осмысленное имя.

Слева всегда показано восьмизначное смещение, а в заголовке обновляется `sizeof`. К полям можно добавлять комментарии.

Клавиша `U` удаляет только последнее поле. В середине она лишь снимает определение, оставляя байты. Затем Edit > Shrink Struct Type удаляет свободную область. Edit > Expand Struct Type вставляет байты перед выбранным полем.

IDA не различает packed и обычную структуру и не добавляет padding сама. Аналитик обязан вручную создать нужные байты. Если известен только общий размер, можно создать массив однобайтовых элементов и последний отдельный байт, затем снять определение с массива: размер сохранится, а поля будут добавляться по мере исследования.

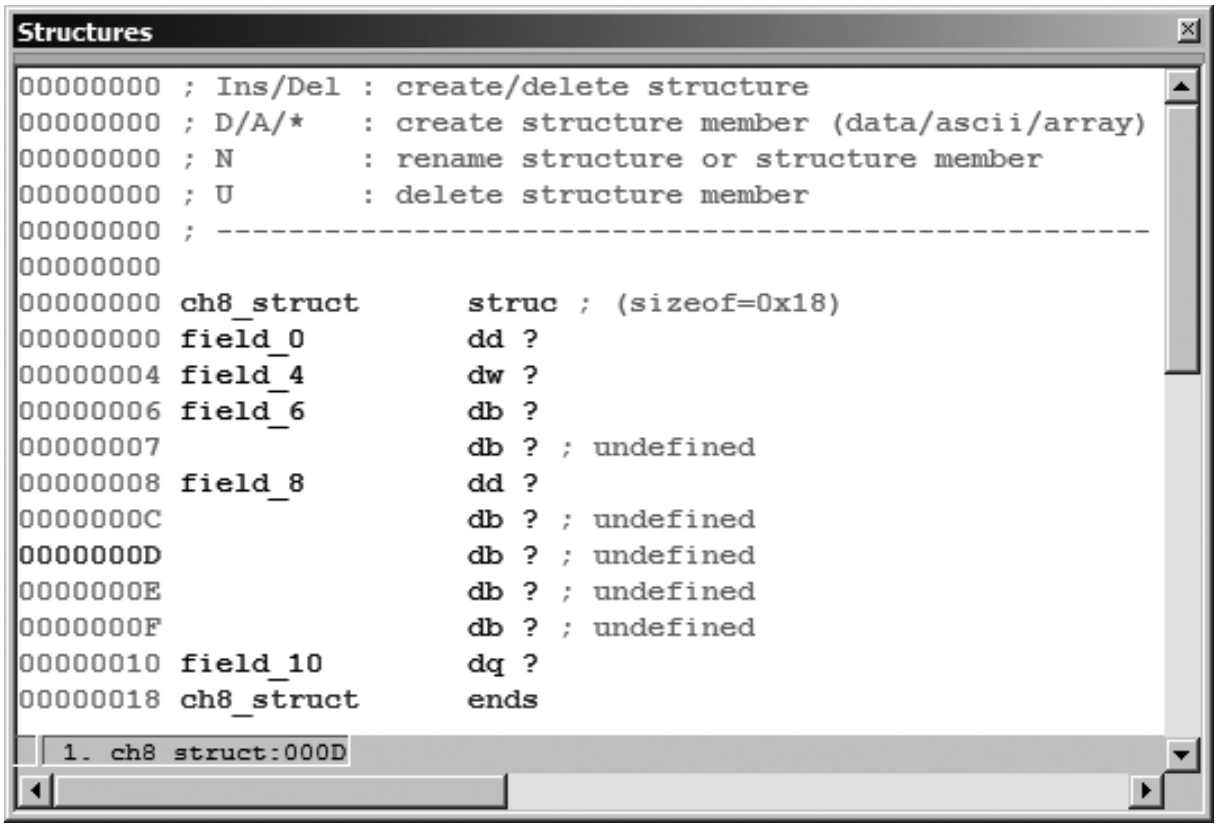

Рис. 8-5. Ручная разметка ch8_struct

На рисунке присутствуют байты заполнения, смещения совпадают с 24-байтовой невыровненной схемой. Минус цифровой клавиатуры сворачивает готовое определение, плюс или двойной щелчок раскрывает.

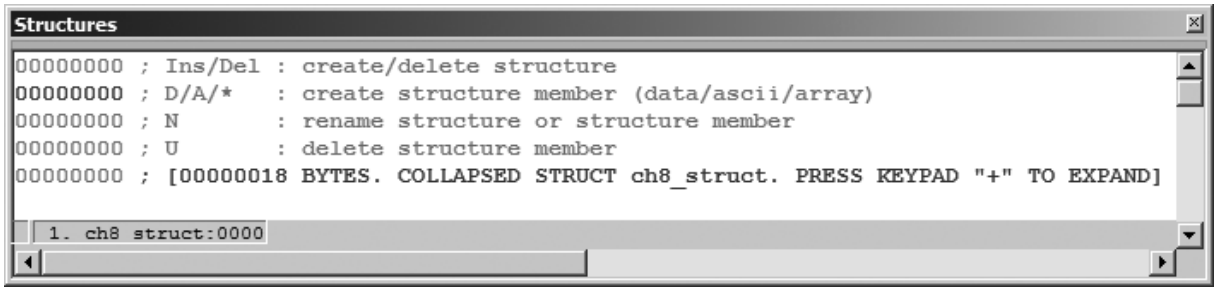

Рис. 8-6. Свёрнутое определение

Кадр стека как структура

Внутренне IDA хранит кадр и структуру сходным образом: непрерывный блок байтов, разбитый на именованные поля со смещениями. Отличие в том, что у обычной структуры смещения неотрицательны от начала, а кадр использует положительные и отрицательные относительно EBP или адреса возврата.

Применение шаблонов

Определение можно использовать двумя способами:

• заменить числовое смещение символическим полем при доступе через указатель;
• назначить структуру целой глобальной или стековой переменной.

Удобно мысленно представить поля как перечисление:

```
enum {
    ch8_struct.field1 = 0,
    ch8_struct.field2 = 4,
    ch8_struct.field3 = 6,
    ch8_struct.field4 = 8,
    ch8_struct.field5 = 16
};
```

Контекстное меню константы `0x10` предлагает Structure offset и все известные структуры с полем на этом смещении.

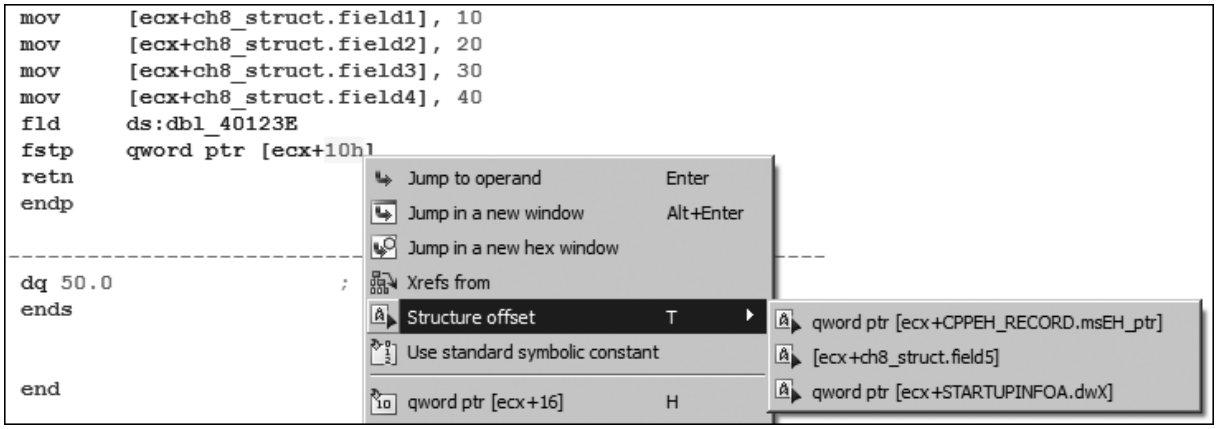

Рис. 8-7. Преобразование смещения в поле

Для целой стековой переменной нужно открыть подробный кадр и выбрать Edit > Struct Var или `Alt+Q`. Появляется выбор известного типа.

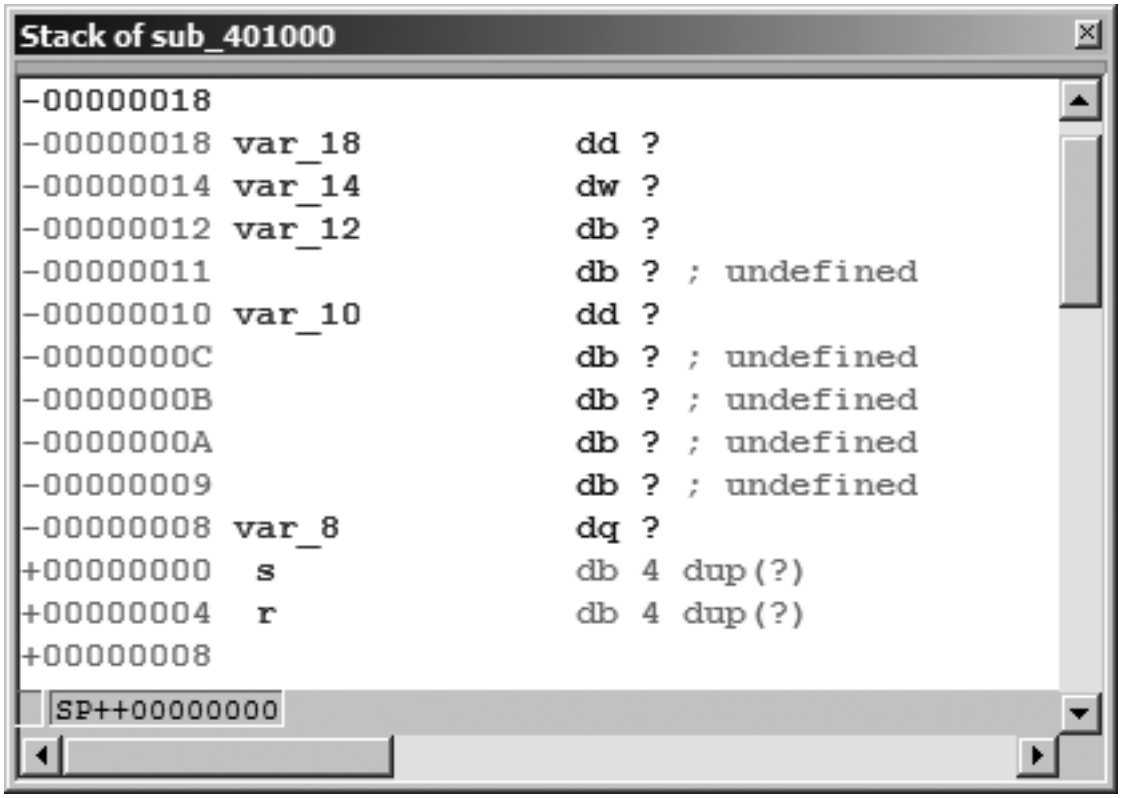

Рис. 8-8. Выбор структуры

До преобразования 24 байта выглядят как несколько значений:

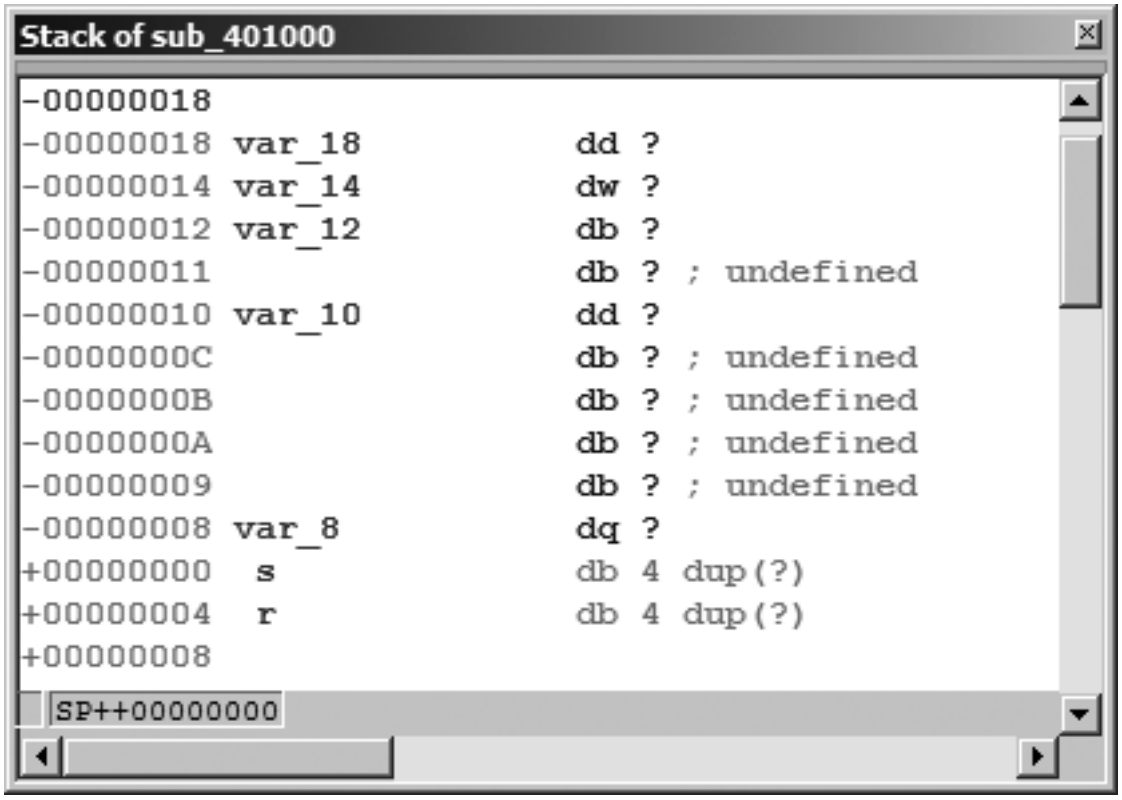

Рис. 8-9. Структура в стеке до форматирования

После назначения `var_18` типа `ch8_struct` отдельные определения объединяются. Предупреждение о разрушении прежних переменных ожидаемо.

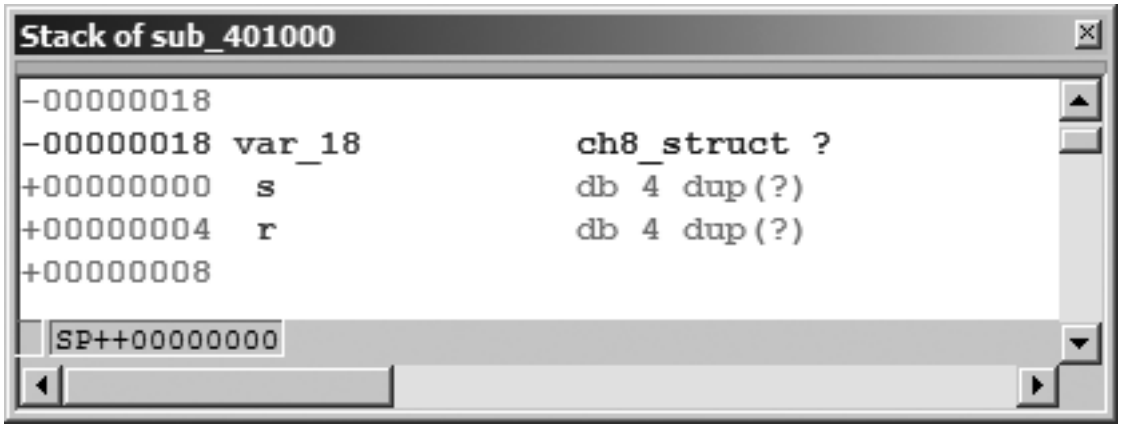

Рис. 8-10. Структура в стеке после форматирования

IDA теперь понимает каждую ссылку внутри 24-байтового диапазона:

```
mov  [ebp+var_18.field1], 10
mov  [ebp+var_18.field2], 20
mov  [ebp+var_18.field3], 30
mov  [ebp+var_18.field4], 40
fld  ds:dbl_40B128
fstp [ebp+var_18.field5]
```

Для глобального объекта процедура почти та же: выбрать начальный адрес и Edit > Struct Var. Для неопределённых глобальных данных тип доступен и через контекстное меню.

Импорт определений

Ручной редактор неудобен для больших типов. IDA умеет разбирать отдельные объявления C и целые заголовочные файлы. C++ этим парсером не поддерживается.

Объявление C

View > Open Subviews > Local Types показывает типы текущей базы. `Insert` открывает ввод объявления.

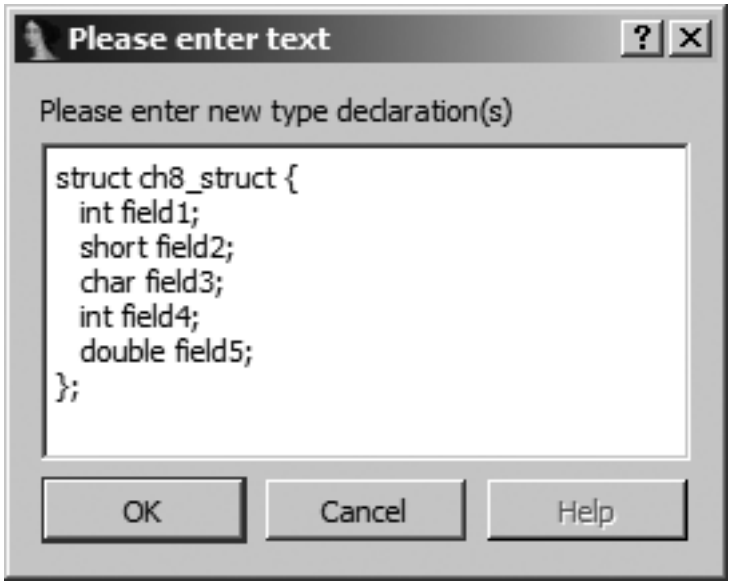

Рис. 8-11. Ввод Local Types

```
struct ch8_struct {
    int field1;
    short field2;
    char field3;
    int field4;
    double field5;
};
```

Ошибки появляются в Output. Успешный тип добавляется в список.

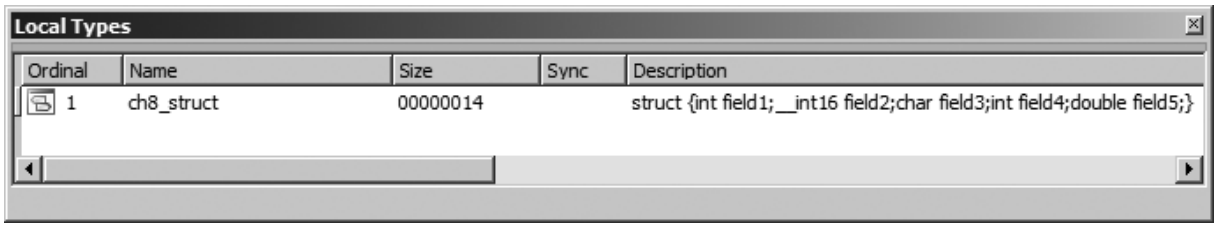

Рис. 8-12. Окно Local Types

Парсер по умолчанию выравнивает поля на четыре байта и понимает `#pragma pack`. Чтобы тип появился в Structures, нужно Synchronize to idb либо добавить его как standard structure.

Заголовочные файлы C

File > Load File > Parse C Header File разбирает файл. Успешный результат сопровождается `Compilation successful`, ошибки печатаются в Output.

Все распознанные структуры попадают в Local Types и конец общего списка, но не в Structures до явного добавления. Новое определение с прежним именем заменяет старое.

Нужно учитывать:

• стандартное выравнивание парсера равно четырём байтам и может отличаться от компилятора;
• `#include` ищется рядом с исходным файлом и в Include directories из Options > Compiler;
• понимаются типы C, `#define` и `typedef`, поэтому `uint32_t` допустим после подходящего typedef;
• без исходника иногда быстрее написать упрощённое объявление в текстовом редакторе, чем работать ручными командами;
• новые типы принадлежат текущей базе, если отдельно не организовать их перенос.

Для обратной разработки правильные размеры и смещения важнее точного имени типа. Если ради разбора нужно заменить `uint32_t` на четырёхбайтовый `int`, это разумный компромисс.

Стандартные структуры

IDA содержит множество типов библиотек и API. При создании базы она определяет компилятор и платформу и загружает соответствующие шаблоны. Structures показывает только применённое подмножество.

В Structures нужно нажать `Insert`, затем Add standard structure. Поиск работает по подстроке и по вводимому префиксу. Вместе с выбранным типом добавляются вложенные.

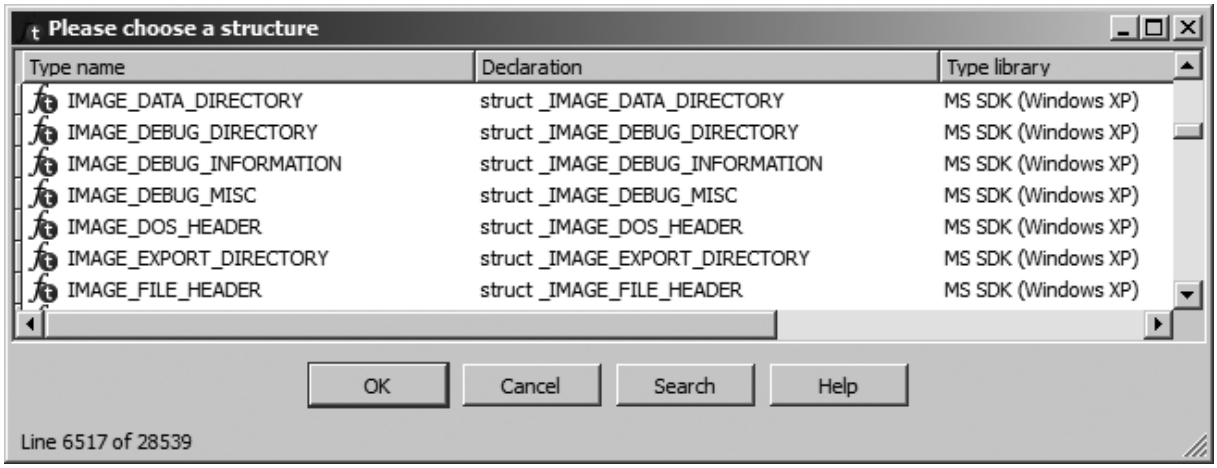

Рис. 8-13. Список стандартных структур

Например, при Manual load в базе присутствуют заголовки PE, но не размечены. Первые байты `4D 5A` являются DOS-сигнатурой. После добавления `IMAGE_DOS_HEADER` и применения `Alt+Q` 64 байта объединяются. Плюс цифровой клавиатуры раскрывает поля:

```
HEADER:00400000 __ImageBase dw 5A4Dh      ; e_magic
HEADER:00400000             dw 90h        ; e_cblp
HEADER:00400000             dw 3          ; e_cp
HEADER:00400000             dw 0          ; e_crlc
HEADER:00400000             dw 4          ; e_cparhdr
...
HEADER:00400000             dd 80h        ; e_lfanew
```

`e_lfanew` указывает файловое смещение `IMAGE_NT_HEADERS`. Применение второго типа по адресу `00400080` сразу раскрывает сведения:

```
HEADER:00400080 dd 4550h     ; Signature
HEADER:00400080 dw 14Ch      ; FileHeader.Machine
HEADER:00400080 dw 5         ; FileHeader.NumberOfSections
HEADER:00400080 dd 4789ADF1h ; FileHeader.TimeDateStamp
...
HEADER:00400080 dd 1000h     ; OptionalHeader.AddressOfEntryPoint
HEADER:00400080 dd 400000h   ; OptionalHeader.ImageBase
```

Видно пять секций и предпочтительную базу `00400000`. Минус сворачивает объект обратно.

Файлы TIL

Все типы и прототипы IDA хранит в type libraries. Поставляемые `.til` находятся в `<IDADIR>/til`, а View > Open Subviews > Type Libraries показывает загруженные наборы.

Загрузка

Если из-за обфускации IDA не определила компилятор, в Types можно нажать `Insert` и выбрать дополнительный TIL. Его структуры добавятся к стандартным, а совпадающие функции немедленно получат прототипы и распространённые типы.

Обмен пользовательскими типами

Ручные и импортированные определения хранятся в компоненте базы с тем же базовым именем: для `some_file.idb` это `some_file.til`. Пока база открыта, файл можно скопировать в каталог TIL и затем загрузить из другой базы.

Официальный альтернативный способ - File > Produce File > Dump Typeinfo to IDC File. Сценарий воссоздаёт структуры в другой базе, но включает только типы, фактически перечисленные в Structures, а не весь результат разбора заголовков.

Отдельная утилита `tilib` перечисляет содержимое и создаёт TIL из заголовков C:

```
tilib -l til\pc\vc6win.til
tilib -c -hch8_struct.h ch8.til
```

Полученный файл нужно поместить в `<IDADIR>/til`. До IDA 6.1 `tilib` поставлялась только как программа Windows, но её результат был кроссплатформенным.

Основы обратной разработки C++

Класс C++ расширяет структуру C, но полное рассмотрение языка выходит за рамки книги. Для анализа необходимо заранее понимать наследование и полиморфизм на уровне исходного кода.

Указатель this

Каждая нестатическая функция-член получает указатель на объект:

```
object1.member_func();   // this = &object1
object2.member_func();   // this = &object2
p_obj->member_func();    // this = p_obj
```

Его удобно считать скрытым первым параметром. Visual C++ передаёт `this` в ECX по `thiscall`. g++ помещает его первым аргументом `cdecl` на вершину стека.

Загрузка адреса в ECX перед вызовом может указывать на Visual C++ и функцию-член. Если один адрес передаётся нескольким функциям, они, вероятно, принадлежат одной иерархии классов. Использование ECX до инициализации внутри функции также является признаком `this`, хотя возможно и `fastcall`.

У g++ признак менее заметен, но функция без указателя первым аргументом определённо не является обычным нестатическим методом.

Виртуальные функции и vtable

Для класса с виртуальными функциями компилятор создаёт таблицу указателей. Каждый объект содержит дополнительное первое поле - указатель на подходящую vtable. Виртуальный вызов выбирает функцию по этой таблице во время исполнения.

```
class BaseClass {
public:
    BaseClass();
    virtual void vfunc1() = 0;
    virtual void vfunc2();
    virtual void vfunc3();
    virtual void vfunc4();
private:
    int x;
    int y;
```

```
};

class SubClass : public BaseClass {
public:
    SubClass();
    virtual void vfunc1();
    virtual void vfunc3();
    virtual void vfunc5();
private:
    int z;
};
```

`SubClass` наследует четыре виртуальных метода и добавляет пятый. `BaseClass::vfunc1` чисто виртуальна и не имеет реализации, поэтому объект базового класса создать нельзя. В слоте обычно находится обработчик `purecall`, аварийно завершающий программу при невозможном вызове.

Хотя явно у BaseClass два поля, фактически есть ещё `p_vftable`; у SubClass четыре поля с унаследованным указателем.

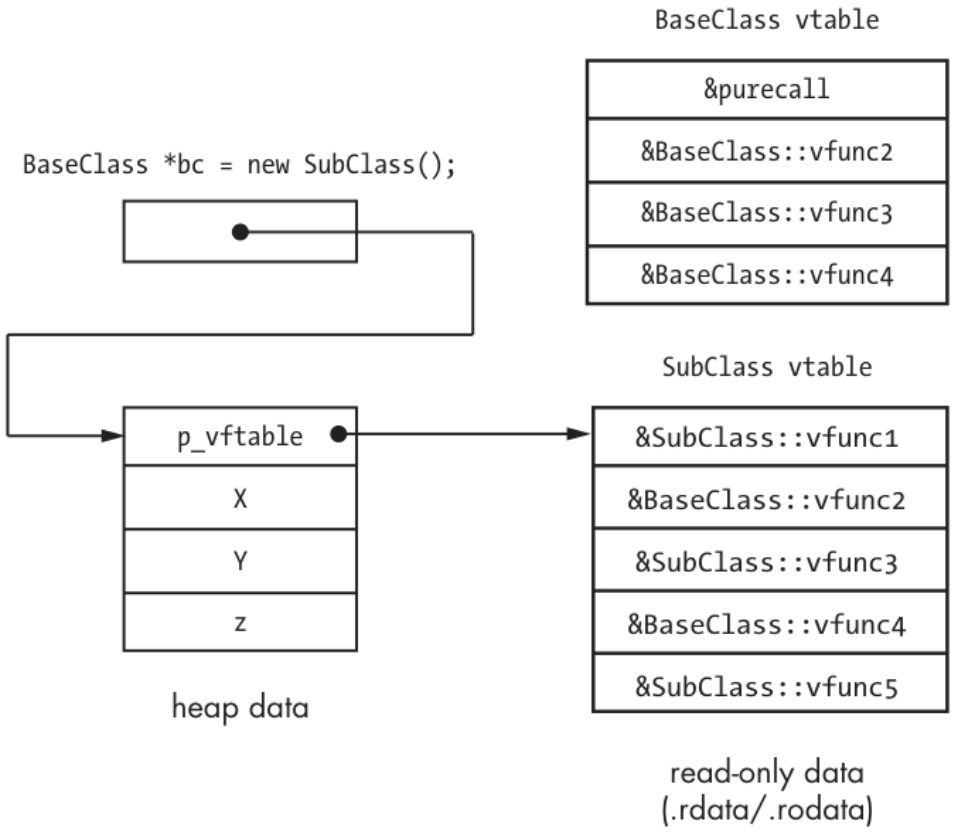

Рис. 8-14. Простая схема vtable

SubClass не переопределяет `vfunc2` и `vfunc4`, поэтому её таблица содержит адреса методов BaseClass. При описании класса через Structures нужно всегда учитывать первый vtable pointer и включать его в размер, видимый в аргументе `new`.

```
void call_vfunc(BaseClass *b) {
    b->vfunc3();
}
```

```
.text:004010A3  mov eax, [ebp+b]
.text:004010A6  mov edx, [eax]      ; адрес vtable
.text:004010A8  mov ecx, [ebp+b]    ; this
.text:004010AB  mov eax, [edx+8]    ; третий слот vfunc3
.text:004010AE  call eax
```

Таблицу также можно описать структурой:

```
00000000 SubClass_vtable struc
00000000 vfunc1 dd ?
00000004 vfunc2 dd ?
00000008 vfunc3 dd ?
0000000C vfunc4 dd ?
00000010 vfunc5 dd ?
00000014 SubClass_vtable ends
```

Тогда доступ становится `[edx+SubClass_vtable.vfunc3]`.

Жизненный цикл объекта

Конструкторы глобальных и статических объектов вызываются до `main`. Стековый объект создаётся при входе в область видимости. Динамическое создание состоит из выделения `new` и вызова конструктора. Visual C++ дополнительно проверяет результат `new` на null.

Конструктор выполняет действия в порядке:

1. вызывает конструктор базового класса;
2. устанавливает vtable pointer текущего класса, при необходимости заменяя значение базового;
3. вызывает конструкторы полей-объектов;
4. выполняет код конструктора, написанный программистом.

Конструктор не имеет объявленного возвращаемого типа, но Visual C++ технически возвращает `this` в EAX.

Деструкторы глобальных объектов вызываются после `main`, стековых - при выходе из области, динамических - оператором `delete` перед освобождением памяти. Порядок примерно обратный:

1. vtable pointer восстанавливается на таблицу текущего класса;
2. выполняется пользовательский код деструктора;
3. вызываются деструкторы полей-объектов;
4. вызывается деструктор базового класса.

Цепочка конструкторов и деструкторов раскрывает наследование. Прямая ссылка на vtable обычно встречается только в них, поэтому xref на найденную таблицу быстро приводит ко всем конструкторам и деструкторам класса.

Декорирование имён

Перегруженные функции имеют одинаковое исходное имя, поэтому компилятор кодирует в символе тип возврата, класс и список параметров. Схема не стандартизована и различается между Visual C++ и g++.

IDA понимает основные варианты. Options > Demangled Names выбирает отображение читаемой формы в комментариях, вместо символа или полное отключение.

Рис. 8-15. Отображение декорированных имён

Комментарий сохраняет исходный символ:

```
.text:00401050 ; protected: __thiscall SubClass::SubClass(void)
.text:00401050 ??0SubClass@@IAE@XZ proc near
...
.text:004010DC call ??0SubClass@@IAE@XZ ; SubClass::SubClass(void)
```

Режим Names заменяет его читаемым именем. Assume GCC v3.x names различает старую схему g++ 2.9 и новую 3.x. Setup short names и Setup long names уточняют формат.

Манглинг мгновенно раскрывает сигнатуру. Без него параметры и результат приходится восстанавливать по потоку данных.

RTTI

Операторы `typeid` и `dynamic_cast` требуют Runtime Type Identification. Реализация зависит от компилятора.

Полиморфный объект уже содержит указатель vtable, поэтому информация о типе размещается рядом с таблицей. Непосредственно перед vtable находится указатель:

• в g++ на `type_info`, содержащую имя класса;
• в Visual C++ на `RTTICompleteObjectLocator`, откуда доступен `TypeDescriptor` со строкой имени.

RTTI нужна только при фактическом использовании `typeid` или `dynamic_cast` и может быть отключена, поэтому её отсутствие нормально.

Определение наследования

При наличии RTTI можно пройти структуру конкретного компилятора. Без неё полезны цепочки конструкторов и сравнение vtable. Встроенный inline-конструктор способен скрыть явный вызов базового класса.

Для таблиц действуют эвристики:

• одинаковое число слотов может указывать на связь;
• если у X больше слотов, X может быть наследником Y;
• общие адреса означают, что X наследует Y, Y наследует X либо оба наследуют общий Z;
• если в X есть `purecall`, а в соответствующем слоте Y уже реализация, Y является наследником X.

В примере SubClass содержит больше методов, разделяет реализации BaseClass и заменяет её `purecall`, поэтому наследует BaseClass.

Итоги

Сложные типы встречаются почти в любой нетривиальной программе. Масштаб индекса, смещения от указателя, размер динамического выделения и инструкции доступа позволяют восстановить массивы, структуры и классы.

Шаблоны IDA заменяют числа именами полей, библиотеки типов распространяют прототипы, а анализ `this`, vtable, жизненного цикла и RTTI раскрывает C++. В следующей главе базовый набор возможностей завершается перекрёстными ссылками и графами.

Глава 9. Перекрёстные ссылки и графы

При обратной разработке постоянно возникают вопросы: откуда вызывается эта функция, какие функции обращаются к данным, как выполнение достигает уязвимого места?

Найденное переполнение бесполезно для эксплойта, если атакующий не может заставить уязвимую функцию выполниться. Необходимо пройти вверх по цепочкам вызовов и понять, какие данные передаются на каждом уровне.

Строка `Executing Denial of Service attack!` сама по себе не доказывает наличие атаки. Нужно найти код, который на неё ссылается, и проверить поведение. IDA решает такие задачи через xref и графическое представление связей между кодом и данными.

Перекрёстные ссылки

В IDA их часто называют xrefs. Есть две основные категории: ссылки кода и ссылки данных. Каждая идёт от одного адреса к другому. В терминах теории графов адреса являются вершинами, ссылки - направленными рёбрами.

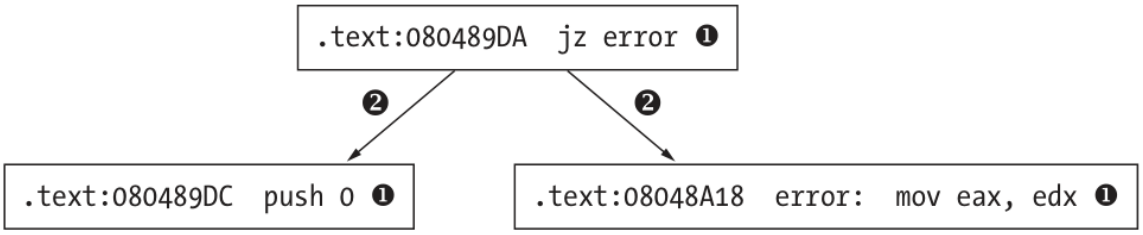

Рис. 9-1. Основные компоненты графа

По стрелке можно пройти от верхней инструкции к каждому нижнему блоку, но не обратно. Code xref лежат в основе графов потока управления и вызовов.

В заголовке функции ссылка отображается обычным комментарием.

```
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 sub_401000      proc near               ; CODE XREF: _main+2A↓p
.text:00401000
```

Рис. 9-2. Базовая перекрёстная ссылка

`CODE XREF` отличает её от `DATA XREF`. Адрес `_main+2A` информативнее абсолютного `.text:0040154A`: источник находится на `0x2A`, то есть 42 байта после начала `_main`. Стрелка вверх или вниз показывает, лежит источник ниже или выше текущего адреса. Однобуквенный суффикс задаёт тип.

Ссылки кода

Code xref означает, что инструкция передаёт или может передать управление другой инструкции. IDA различает обычный поток, переход и вызов. Для сегментированной адресации переходы и вызовы дополнительно бывают near и far.

Рассмотрим программу:

```
int read_it;
int write_it;
int ref_it;

void callflow() {}

int main() {
    int *p = &ref_it;
    *p = read_it;
    write_it = *p;
    callflow();
    if (read_it == 3)
        write_it = 2;
    else
        write_it = 1;
    callflow();
}
```

Обычный поток - последовательное выполнение следующей инструкции. Отдельный индикатор не нужен: B просто следует за A в листинге. Исключения - ветвления и возврат.

Поток вызова создаётся инструкцией `call`. Обычно у неё есть и обычное продолжение после возврата. Если функция отмечена Does not return, продолжения нет.

```
.text:00401000 callflow proc near ; CODE XREF: _main+20 p
.text:00401000                    ; _main:loc_401054 p
...
.text:00401004 retn
```

Суффикс `p`, Procedure, обозначает вызов. Если источник имеет имя, используется оно; иначе показывается смещение внутри функции.

Поток перехода создаётся условным или безусловным branch. У условной инструкции сохраняется и обычный путь, у безусловной нет. Разрыв пунктирной линией между соседними строками показывает отсутствие последовательного потока. В цели xref получает суффикс `j`, Jump.

Ссылки данных

Data xref отслеживает чтение, запись и получение адреса. Она применима к байтам с виртуальным адресом, но не к локальным стековым переменным.

```
.data:0040B720 read_it  dd ? ; DATA XREF: _main+E r
.data:0040B720               ; _main+25 r
.data:0040B724 write_it dd ? ; DATA XREF: _main+1B w
.data:0040B724               ; _main+2E w ...
.data:0040B728 ref_it   db ? ; DATA XREF: _main+4 o
```

Read, суффикс `r`, означает чтение содержимого. Источник всегда является инструкцией, цель может быть любым адресом. Загрузка `read_it` в 32-разрядный ECX помогает IDA определить `dd`.

Write, суффикс `w`, отмечает изменение. Многоточие означает, что ссылок больше настроенного предела Number of displayed xrefs на вкладке Cross-references в Options > General. Запись в байты инструкции часто указывает на самомодифицирующийся код и встречается в распаковщиках вредоносных программ.

Offset, суффикс `o`, означает использование самого адреса, а не содержимого. Это типично для указателя. База глобального массива или строки часто является целью offset xref.

В отличие от чтения и записи, offset может исходить и из данных. Таблица указателей, например vtable, создаёт ссылку от каждого слота к функции:

```
.rdata:00408148 off_408148 dd offset SubClass::vfunc1(void)
.rdata:0040814C            dd offset BaseClass::vfunc2(void)
.rdata:00408150            dd offset SubClass::vfunc3(void)
```

```
.rdata:00408154           dd offset BaseClass::vfunc4(void)
.rdata:00408158           dd offset SubClass::vfunc5(void)
```

Виртуальная функция не вызывается напрямую и обычно не имеет call xref. Зато её адрес присутствует хотя бы в одной vtable, поэтому есть offset xref. Переход назад от таких ссылок помогает находить таблицы классов.

Полные списки xref

Если число ссылок не превышает настроенный предел, их можно просмотреть во всплывающей подсказке и открыть двойным щелчком. Полный список доступен двумя способами.

Поставив курсор на адрес назначения, View > Open Subviews > Cross-References создаёт постоянную вкладку.

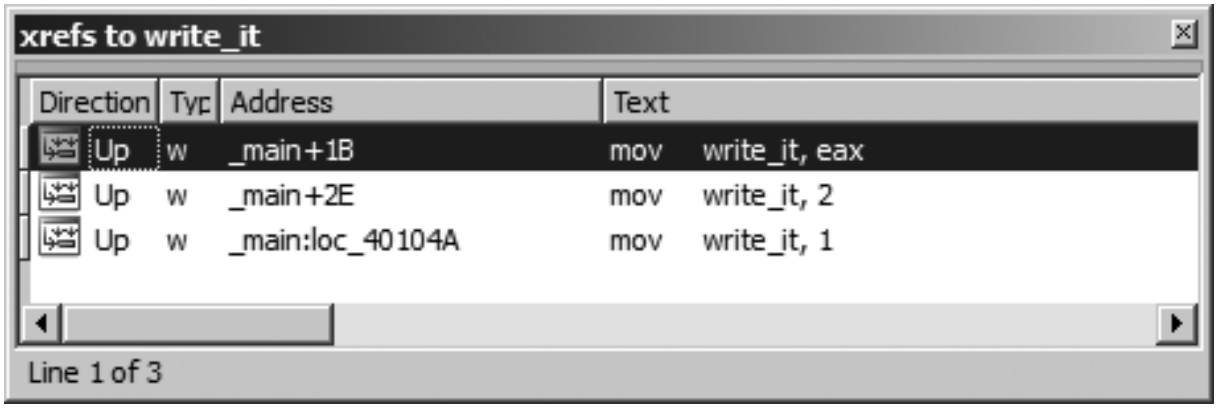

Рис. 9-3. Окно перекрёстных ссылок

Столбцы показывают направление Up или Down, тип, исходный адрес и текст инструкции. Двойной щелчок переходит к источнику. Вкладка остаётся открытой.

Второй способ - выделить любое употребление символа и выбрать Jump > Jump to xref либо `Ctrl+X`.

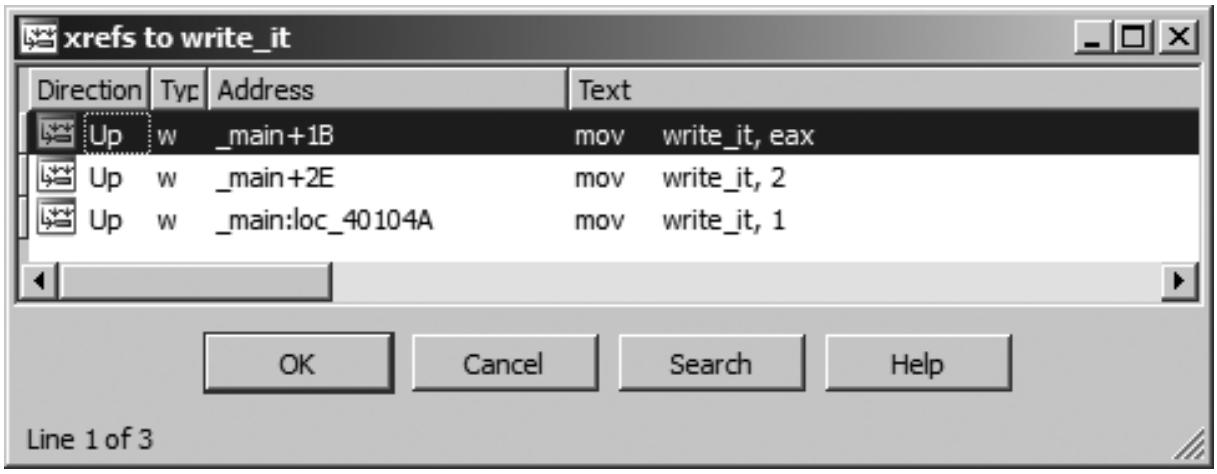

Рис. 9-4. Диалог Jump to xref

Этот список является модальным: выбор закрывает его и переносит листинг. Он доступен из любого употребления символа, тогда как subview открывается только на адресе назначения.

Например, чтобы исследовать все небезопасные вызовы `strcpy`, достаточно найти одно употребление, нажать `Ctrl+X` и пройти по call xref. Имя, написанное в комментарии, IDA также

воспринимает как символ и позволяет вызвать для него те же операции.

Вызовы функций

View > Open Subviews > Function Calls показывает ближайшее окружение текущей функции. Верхняя таблица содержит вызывающие места, нижняя - сделанные вызовы.

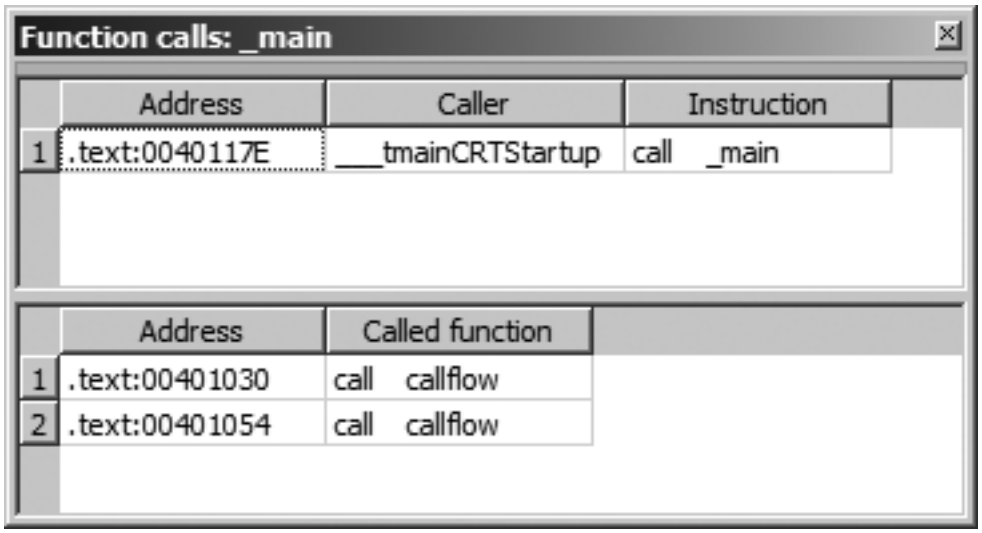

Рис. 9-5. Окно Function Calls

Это уже не просто связь адресов, а отношение между функциями, которое естественно представить графом.

Графы IDA

В графе xref становятся рёбрами. Вершиной может быть инструкция, базовый блок или целая функция. IDA предоставляет внешние графы через отдельную программу и встроенное интерактивное представление.

Внешняя программа

Старые версии Windows поставлялись с `wingraph32`, IDA 6.0 в других системах использовала `dotty`, а с 6.1 все платформы получили Qt-программу `qwingraph`. Настройка находится в переменной `GRAPH_VISUALIZER` файла `<IDADIR>/cfg/ida.cfg`.

IDA записывает временное описание, затем запускает просмотрщик. Поддерживаются GDL и DOT, выбираемые `GRAPH_FORMAT`; язык должен соответствовать программе просмотра.

View > Graphs создаёт пять видов:

• flowchart текущей функции;
• call graph всего файла;
• xref к символу;
• xref от символа;
• настраиваемый граф xref.

Для flowchart и call graph File > Produce File сохраняет GDL. Внешние графы не интерактивны, обычно доступны только масштаб и прокрутка.

Базовый блок имеет один вход в первую инструкцию и один выход из последней. После исполнения первой инструкции оставшаяся часть блока гарантированно выполняется. При динамическом покрытии достаточно поставить breakpoint в начале каждого блока вместо каждой инструкции.

Внешний flowchart

View > Graphs > Flow Chart или `F12` строит CFG функции из обычных и jump flow. Он похож на встроенный граф, но содержит мало адресной информации, поэтому его труднее сопоставлять с листингом.

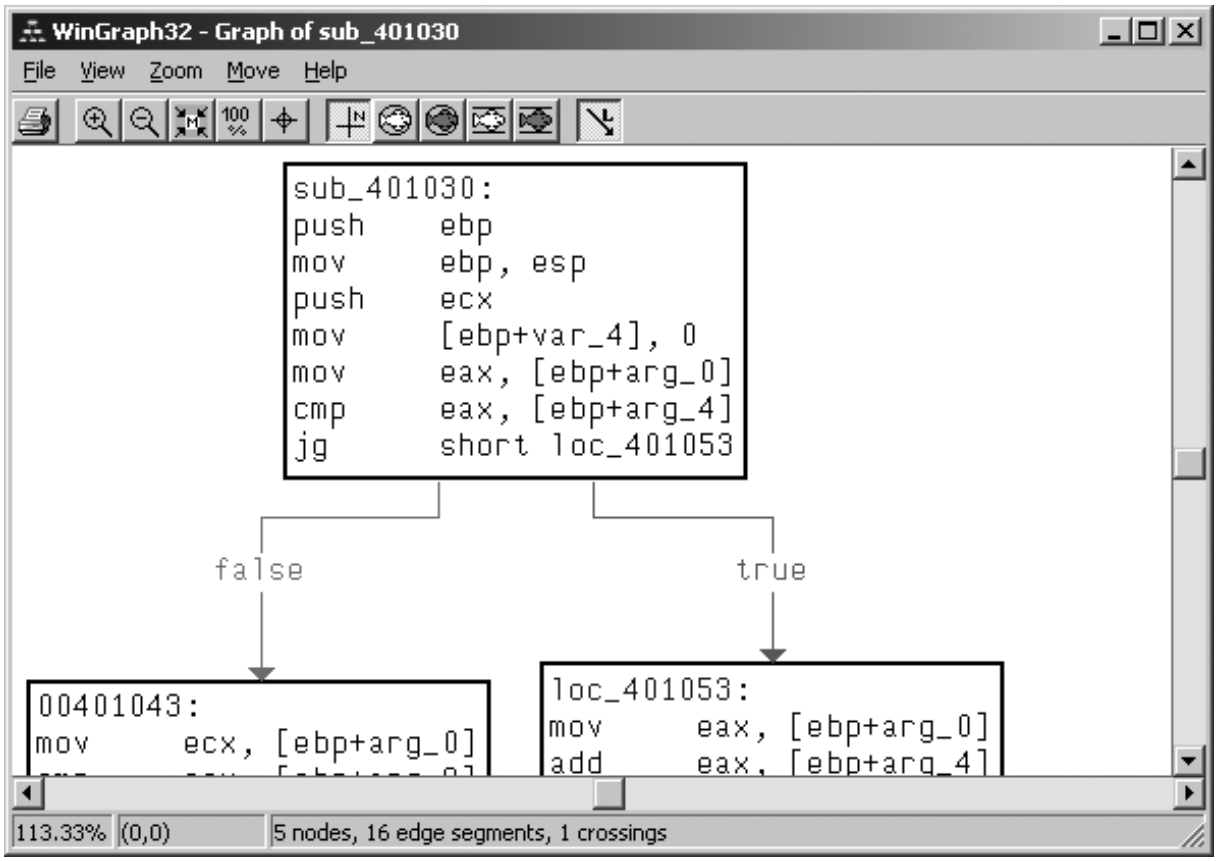

Рис. 9-6. Внешний flowchart

Граф вызовов

Для каждой функции создаётся вершина, call xref становятся рёбрами. Рекурсивный обход можно прекращать на библиотеке: её поведение проще прочитать в документации. В динамическом файле тела библиотеки вообще нет. В статическом оно присутствует и резко увеличивает граф.

Примерная иерархия:

```
void depth_2_1() { printf("inside depth_2_1\n"); }
void depth_2_2() { fprintf(stderr, "inside depth_2_2\n"); }
void depth_1() {
    depth_2_1();
    depth_2_2();
    printf("inside depth_1\n");
}
```

```
int main() { depth_1(); }
```

View > Graphs > Function Calls для динамической сборки даёт читаемую схему. Компилятор может заменить `printf` и `fprintf` на более эффективные `puts` и `fwrite`. В граф попадает также служебный код запуска и завершения до `main`.

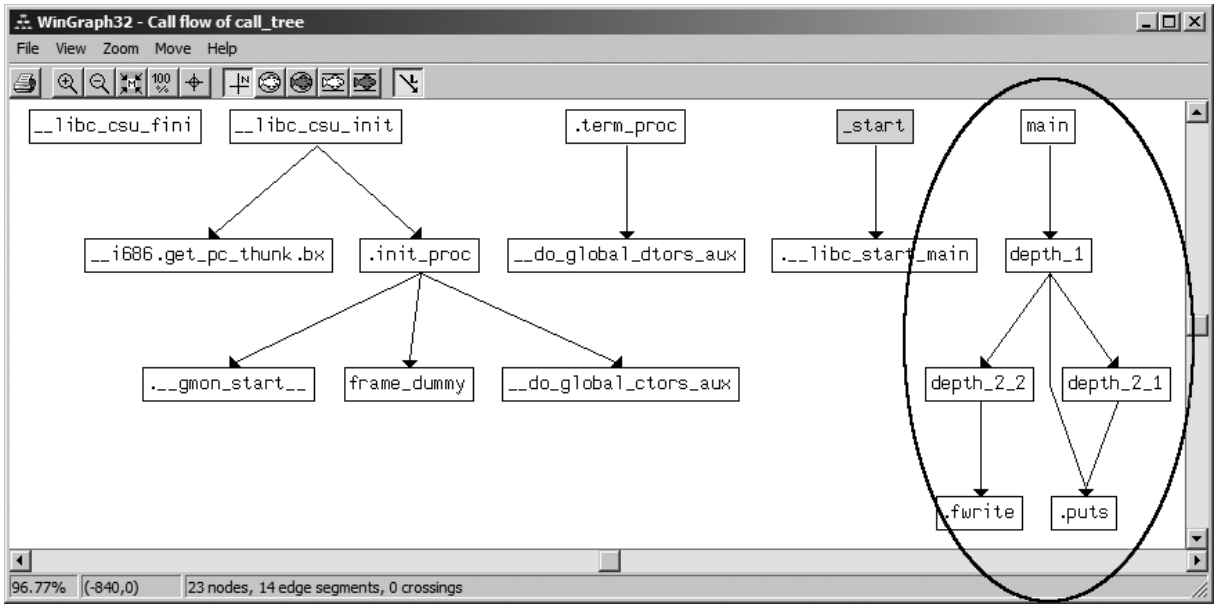

Рис. 9-7. Внешний граф вызовов

Статическая сборка примера породила 946 вершин, 10 125 участков рёбер и 100 182 пересечения. Просмотрщик изначально вписывает всё в окно, и схема практически бесполезна.

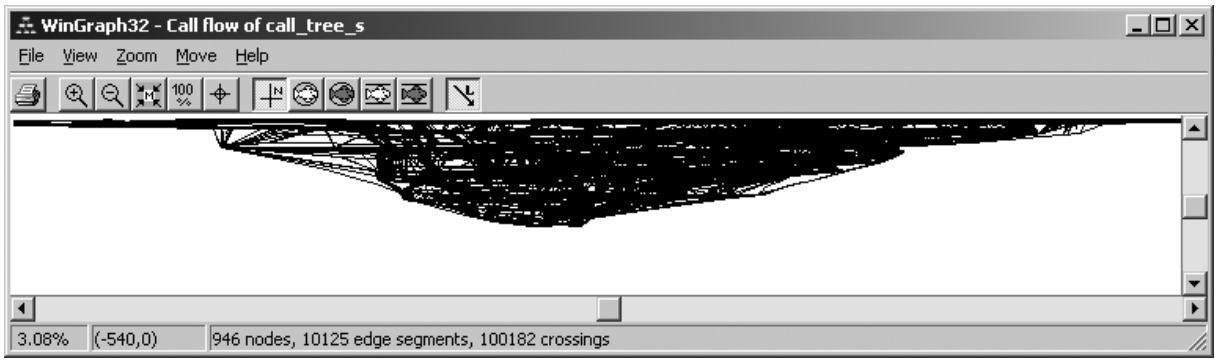

Рис. 9-8. Call graph статически связанной программы

Xrefs To и Xrefs From

View > Graphs > Xrefs To рекурсивно идёт назад по ссылкам к выбранному глобальному символу. Он отвечает на вопрос, какая цепочка вызовов достигает функции.

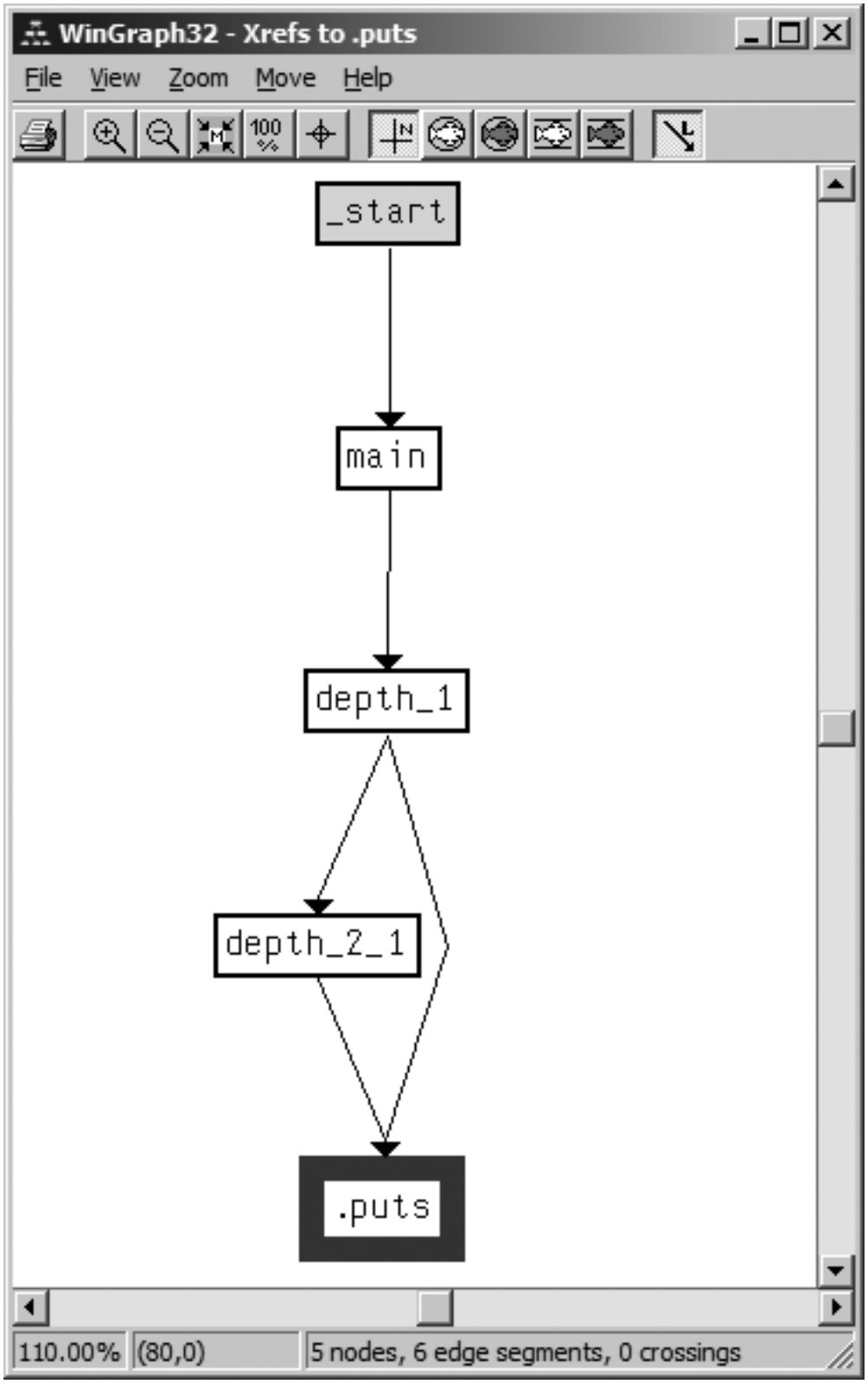

Рис. 9-9. Граф Xrefs To

Этот тип способен включать ссылки данных и показывать функции, достигающие обращений к глобальной переменной.

Xrefs From идёт вперёд. Для функции отслеживаются только вызовы, а не глобальные данные. Для инициализированного глобального указателя прослеживается offset xref. В результате для функции получается rooted call graph.

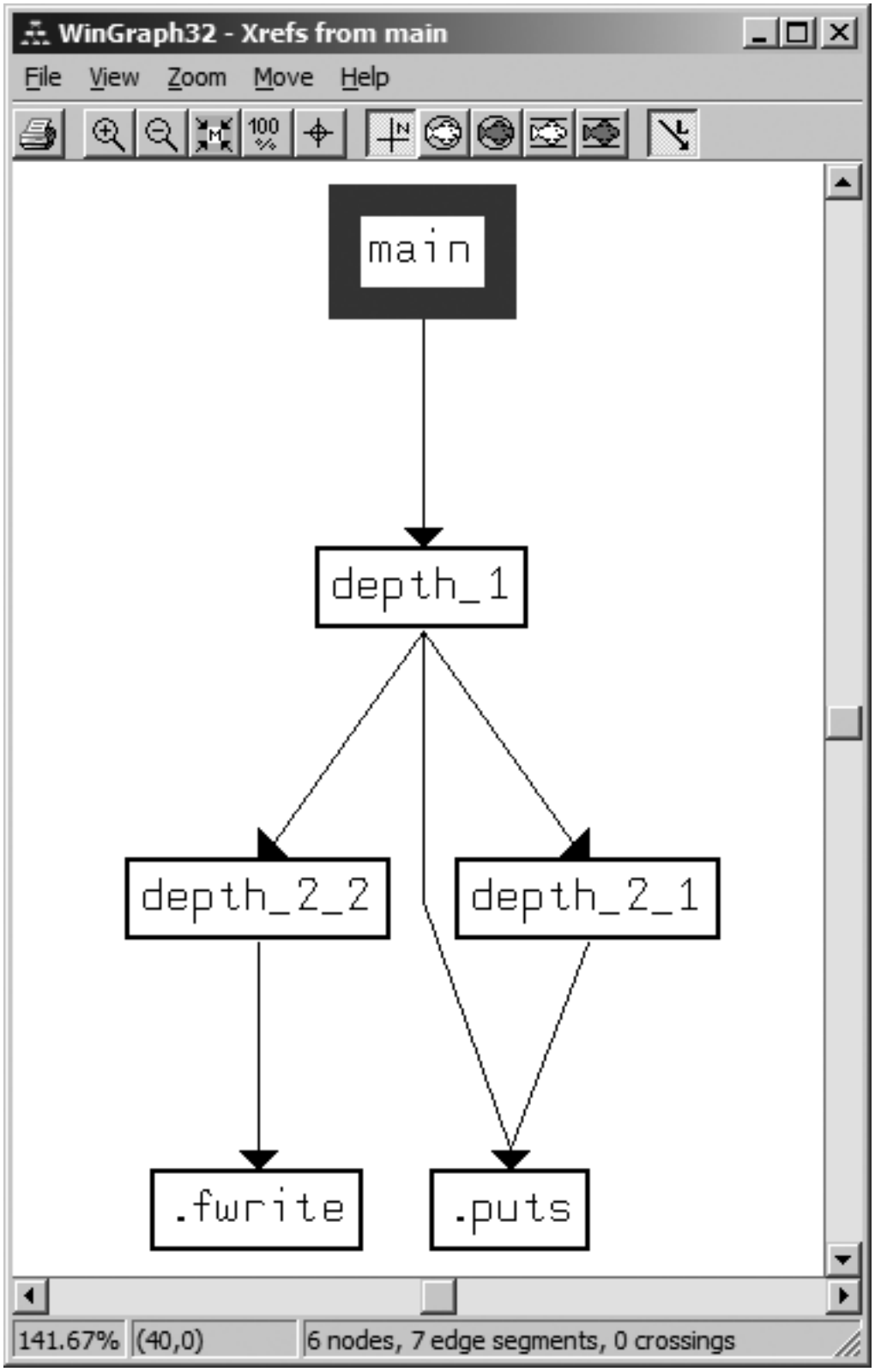

Рис. 9-10. Граф Xrefs From

Для сложного кода здесь также возникает перегруженность.

Пользовательский граф xref

View > Graphs > User Xrefs Chart сочетает оба направления и позволяет ограничить глубину и категории вершин.

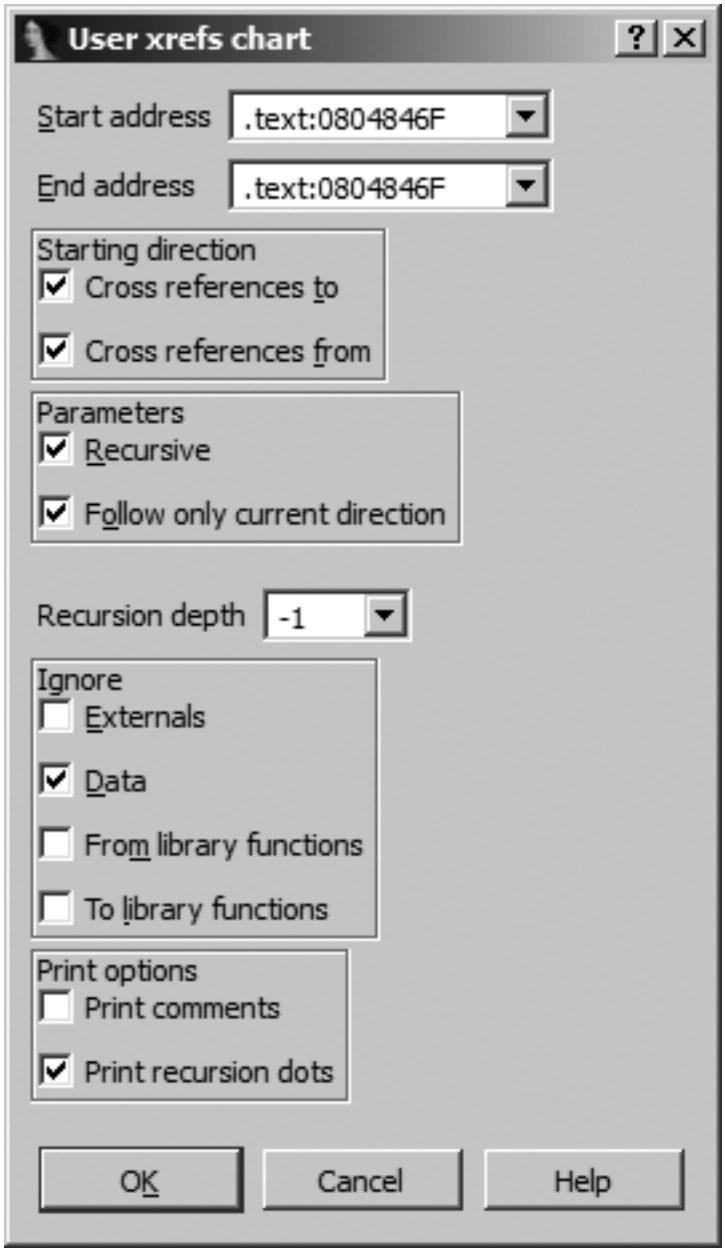

Рис. 9-11. Диалог User Xrefs Chart

Все глобальные символы в диапазоне становятся начальными вершинами. Для одного символа start и end одинаковы; диапазон всей базы вырождается в полный call graph.

Starting direction включает ссылки к символу, от него или оба направления.

Recursive продолжает подъём или спуск. Follow only current direction не позволяет найденной вершине переключать направление. Если снять его и выбрать оба исходных направления, каждая новая вершина исследуется и назад, и вперёд.

Recursion depth ограничивает глубину; `-1` означает полный обход.

Ignore исключает externals, data, переходы из или в library functions. Игнорирование библиотечного кода особенно упрощает статические файлы, если IDA правильно распознала библиотеки.

Print comments включает комментарии функций. Print recursion dots добавляет `...`, если за пределом глубины существуют продолжения.

Для `depth_1` при глубине 1 видны `main`, непосредственные вызываемые функции и точки дальнейшего продолжения.

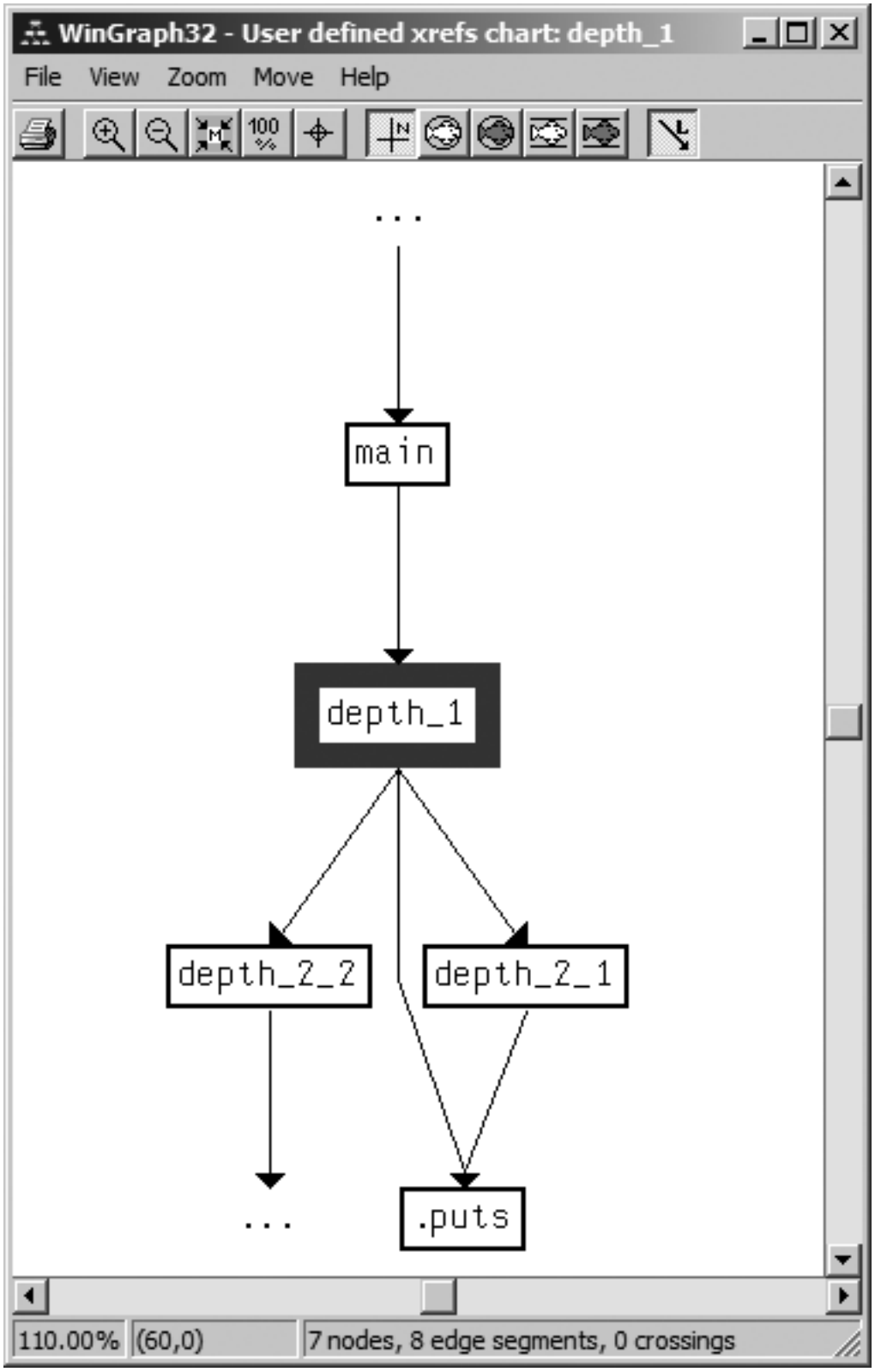

Рис. 9-12. User xref graph для depth_1

Это самый гибкий внешний режим. Flowchart в основном заменён встроенным графом, а остальные готовые варианты являются частными конфигурациями User Xrefs.

Встроенный Graph View

С IDA 5.0 интерактивный CFG стал альтернативой текстовому листингу. Он показывает одну функцию и не работает для инструкций вне функции. Для нескольких функций или глобальных данных нужен текстовый вид.

Пробел и контекстное меню переключают режимы. Граф перетаскивается за фон, большой удобнее перемещать рамкой Graph Overview. Двойной щелчок, история и операции над операндами работают как в тексте. Переход к данным автоматически включает текстовый вид, возвращение в функцию восстанавливает граф. Краткий кадр находится в корневом блоке, подробный открывается двойным щелчком.

Главное различие относится к панели отдельной вершины.

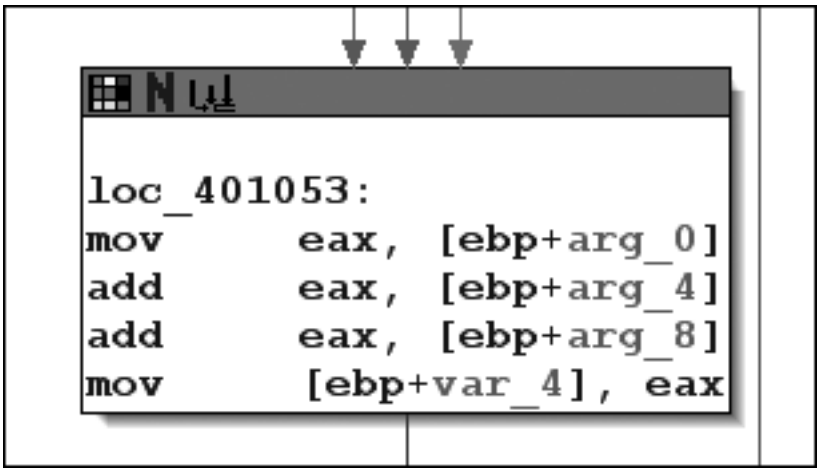

Рис. 9-13. Развёрнутая вершина графа

Слева направо кнопки меняют цвет, назначают имя первому адресу блока и открывают входящие xref.

Цвет помогает отмечать исследованные или интересные блоки и переносится в фон тех же инструкций текстового вида. Set node color to default удаляет его.

Не каждый базовый блок уже имеет имя. После условной инструкции один следующий блок является целью jump, второй начинается обычным продолжением и может остаться без символа. Средняя кнопка позволяет назвать его.

Правая кнопка показывает xref к блоку. В список включён и обычный входящий поток с типом ^, поскольку на графе не всегда очевиден линейный предшественник. Чтобы обычные комментарии xref были видны прямо в узлах, Number of displayed xrefs должен быть больше нуля.

Несколько блоков выбираются `Ctrl`-щелчками по заголовкам, затем Group nodes. Пользователь задаёт текст свёрнутой вершины.

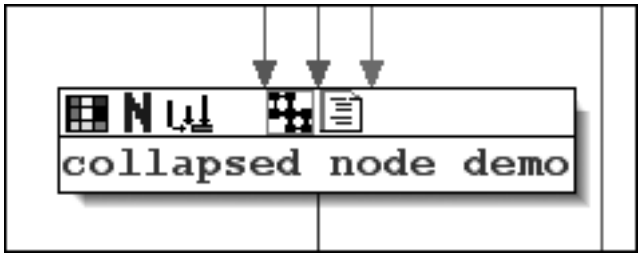

Рис. 9-14. Свёрнутая вершина

Появляются кнопки раскрытия и редактирования текста. Раскрытие показывает исходные блоки, но не удаляет группу. Hide Group снова сворачивает, Ungroup Nodes полностью отменяет группировку и при необходимости раскрывает её.

Итоги

Xref и графы помогают видеть отношения, скрытые в линейном листинге. Для вопроса «как достигается функция?» подходит call graph или Xrefs To, для пути к конкретной инструкции - control flow graph.

Встроенный режим сохраняет почти все данные текстового представления, но оформляет ссылки как рёбра между блоками. Внешние графы полезны для общего обзора и настраиваемого обхода, хотя большие статические программы быстро делают их нечитаемыми.

Глава 10. Разные режимы IDA

Много лет главной версией IDA считался графический интерфейс Windows. С IDA 6.0 полноценный GUI появился также в Linux и OS X, но консольный вариант, происходящий от первой программы для MS-DOS, сохранился на всех платформах.

Встроенная удалённая отладка делает IDA многоплатформенным средством анализа. Кроме интерактивной работы, все версии поддерживают пакетный режим для автоматической обработки множества файлов. Важно понимать ограничения каждого варианта и выбирать подходящий.

Консольная IDA

Текстовые версии использовали перенесённую на Windows, Linux и OS X библиотеку TVision компании Borland. Общая библиотека обеспечивала одинаковый интерфейс, хотя поддержка мыши, изменения размера и передачи горячих клавиш зависела от терминала.

Общие возможности

Консольная IDA работает внутри терминала или оболочки. Верхняя строка содержит меню и состояние, нижняя - панель часто используемых команд. Операции запускаются горячими клавишами или мышью. Почти все команды GUI присутствуют и здесь с теми же сочетаниями.

Между панелями находятся окна данных. В типичном терминале 80 на 25 символов мало места, поэтому по умолчанию открыты только дизассемблирование и сообщения. TVision показывает окна внахлёст, а `F6` циклически переключает их вместо вкладок. Номер виден в левом верхнем углу.

При поддержке мыши окно перемещают за верхнюю границу и меняют размер за правый нижний угол. Без мыши служат Window > Resize/Move, `Ctrl+F5`, стрелки для движения и `Shift` со стрелками для изменения размера. Если сам терминал можно растянуть, IDA подстраивается.

Графический режим дизассемблирования и стрелки потока недоступны, но почти все subviews сохранены. Отдельной вкладки Hex View нет. Options > Dump/Normal View или `Ctrl+F4` переключает текущее окно между листингом и hex dump. Чтобы видеть оба, открывают второе Disassembly и переводят его в Dump, но синхронизировать их нельзя.

Двойной щелчок по имени или `Enter` переходит к адресу. `Enter` на стековой переменной открывает подробный кадр. Без мыши меню вызываются через `Alt` и выделенную букву.

Особенности Windows

Консольная программа называлась `idaw.exe`, графическая `idag.exe`; для 64-разрядного анализа - `idaw64.exe` и `idag64.exe`.

Для мыши в `cmd.exe` необходимо заранее отключить QuickEdit mode в Properties > Options. Изменение после запуска IDA не подхватывается.

Обычный `cmd.exe` нельзя произвольно расширить мышью, поэтому Window > Set Video Mode предлагает шесть фиксированных размеров до 255 на 100 символов.

Встроенного Graph View нет, но View > Graphs запускает внешний `qwingraph`. В Windows можно держать несколько графов открытыми и одновременно продолжать работу в IDA.

Особенности Linux

Программы назывались `idal` и `idal64`. До IDA 6.0 консольные версии копировались из общей поставки вместе с `ida.key`; ключ можно было поместить в `$HOME/.idapro/ida.key`. IDA 6.x устанавливалась отдельно для выбранной платформы и сама размещала GUI, console и ключ.

Файл `tvtuning.txt` содержал рекомендации для разных терминалов, в том числе SecureCRT и PuTTY. Главная проблема - конфликт горячих клавиш. Например, `Alt+F` может перехватить File меню терминала. Решение - выбрать настраиваемый терминал либо переназначить команды IDA. Разные раскладки на разных компьютерах усложняют привычки и совместную работу.

В обычной текстовой консоли размер фиксирован, мышь зависит от GPM. Без GPM следует запускать:

```
TVOPT=noGPM ./idal file_to_disassemble
```

Цветов доступно только 16, поэтому иногда нужно выбрать одну из четырёх палитр или вручную настроить Options > Colors, чтобы текст не сливался с фоном.

Под X можно использовать `konsole`, `gnome-terminal`, `xterm` и другие. `xterm` имеет мало конфликтов, KDE `konsole` авторы книги предпочитали за внешний вид, мышь и удобные клавиши.

Существовал нативный X11-порт TVision Джереми Купера. После замены `libtvision.so` IDA получала собственное X-окно и могла перенаправляться на другой компьютер через SSH X forwarding. При ошибке загрузки VGA-шрифта требовалось отдельно установить подходящий шрифт и добавить путь в X server.

Для удалённого запуска со стандартной TVision терминал должен эмулировать xterm. Например, для мыши SecureCRT:

```
TVOPT=xtrack ./idal
```

Переменную можно экспортировать постоянно. Внешние графы доступны только в оконной среде и при корректном `GRAPH_VISUALIZER` в `ida.cfg`:

```
GRAPH_VISUALIZER = "qwingraph.exe -remove -timelimit 10"
```

`-remove` удаляет временное описание, `-timelimit 10` ограничивает красивую раскладку десятью секундами, после чего применяется быстрый упрощённый алгоритм. Для `aiSee` можно было указать абсолютный путь:

```
GRAPH_VISUALIZER = "/usr/local/bin/aisee"
```

Начиная с 6.0 настройки для Windows и остальных систем находились в разных условных блоках файла.

Особенности OS X

Имена также `idal` и `idal64`. Клавиатура Mac создаёт проблему: Option не всегда ведёт себя как PC Alt.

В Terminal необходимо включить Use option key as meta key, тогда `Option+F` достигает меню IDA. Без этого вместо Alt используется `Esc`, но в IDA он уже отвечает за возврат и закрытие окна.

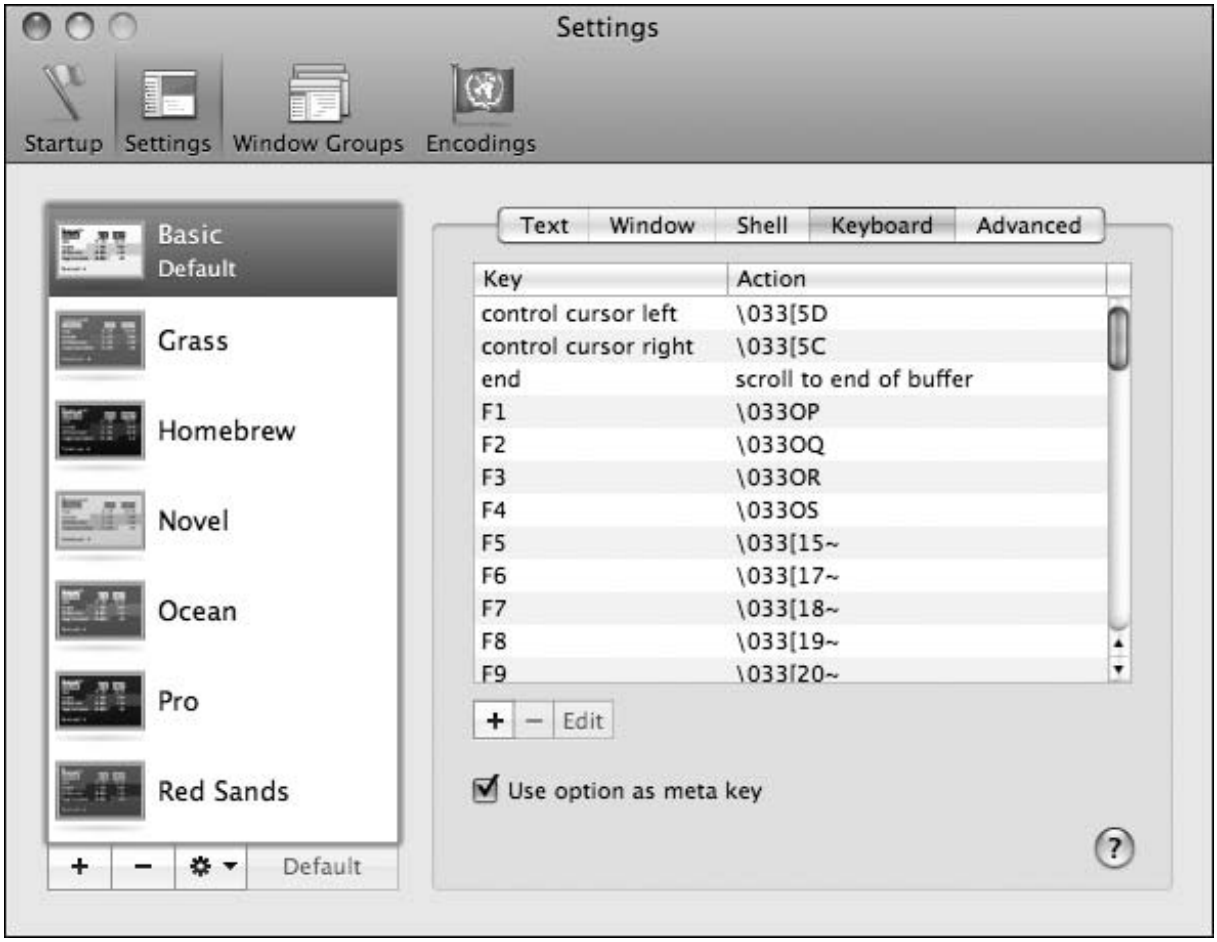

Рис. 10-1. Настройка клавиатуры Terminal в OS X

Альтернативой являлся iTerm с поддержкой Alt и мыши. Также существовал перенос gnome-terminal под X11.

При установке X11 и изменённой `libtvision.dylib` IDA могла работать в отдельном X-окне. Но X11 воспринимал Option как `Mode_switch`, поэтому клавиши переназначались через `~/.Xmodmap`:

```
clear Mod1
keycode 66 = Alt_L
keycode 69 = Alt_R
add Mod1 = Alt_L
add Mod1 = Alt_R
```

Стандартный `/etc/X11/xinit/xinitrc` читает этот файл. При пользовательском `.xinitrc` нужно явно добавить:

```
xmodmap $HOME/.Xmodmap
```

В X11 Preferences следует отключить Follow system keyboard layout, иначе системная раскладка перезапишет изменения.

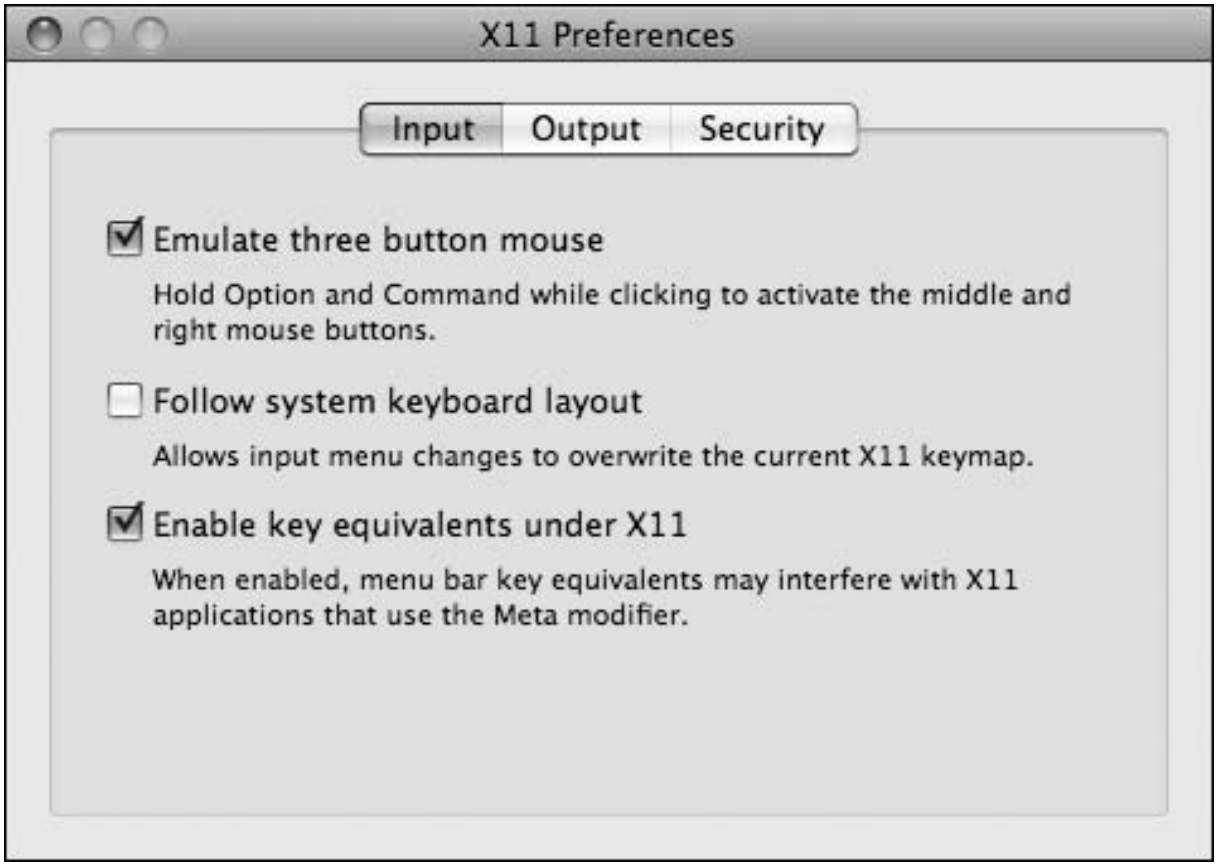

Рис. 10-2. Настройки X11 в OS X

Проверка:

```
$ xmodmap
shift    Shift_L (0x40), Shift_R (0x44)
lock     Caps_Lock (0x41)
control  Control_L (0x43), Control_R (0x46)
mod1     Alt_L (0x42), Alt_R (0x45)
mod2     Meta_L (0x3f)
```

Если в `mod1` нет `Alt_L` и `Alt_R`, команду загрузки `.Xmodmap` нужно повторить.

Пакетный режим

Все версии IDA могут автоматически запуститься, выполнить IDC-скрипт и завершиться. GUI при batch не показывает окон и удобно встраивается в другие сценарии.

Windows console открывает полноценный экран, который закрывается в конце. Вывод можно подавить:

```
idaw -B some_program.exe > NUL
```

Основные параметры:

• `-A` включает автономный режим без интерактивных диалогов. Непринятое ранее лицензионное соглашение всё равно может появиться.
• `-c` удаляет прежнюю базу для указанного файла и создаёт новую.
• `-Sscript.idc` запускает сценарий при старте, без пробела после `S`. Поиск идёт в `<IDADIR>/idc`. При установленном IDAPython допускается `.py`.

• `-B` эквивалентен `-A -c -Sanalysis.idc`. Стандартный `analysis.idc` ждёт завершения анализа, создаёт `.asm`, сохраняет базу и закрывает IDA.

Ключевым является `-S`: IDA завершится только в том случае, если сам сценарий вызовет завершение. Иначе параметры автоматизируют лишь запуск. IDC рассматривается в главе 15.

Из-за ограничений TVision пакетный `idal` Linux и OS X должен был работать в TTY. Новая TVision распознавала `TVHEADLESS`, позволяя перенаправить stdout:

```
TVHEADLESS=1 ./idal -B input_file.exe > /dev/null
```

Для полного фонового отделения дополнительно перенаправляются stdin и stderr. Пакетный режим можно расширить последовательностью сценариев и запуском плагина.

Итоги

GUI остаётся наиболее полным интерфейсом, но консольный и пакетный режимы дают гибкость для удалённой работы и автоматизированных систем анализа.

На этом базовые возможности IDA рассмотрены. Следующие главы переходят к расширенной настройке и вспомогательным средствам анализа.

Глава 11. Настройка IDA

Со временем у пользователя появляются предпочтения, которые хочется применять к каждой новой базе. Одни параметры сохраняются между сеансами, другие приходится задавать заново. IDA управляет поведением через конфигурационные файлы, команды меню и параметры, сохранённые либо глобально, либо только в текущей базе.

Конфигурационные файлы

Большинство файлов находится в `<IDADIR>/cfg`. Исключение составляет конфигурация плагинов `<IDADIR>/plugins/plugins.cfg`, рассматриваемая в главе 17. Многие файлы из `cfg` принадлежат процессорным модулям и используются только для определённых архитектур.

Три главных файла:

• `ida.cfg` - общие параметры всех версий IDA;
• `idagui.cfg` - параметры графического интерфейса;
• `idatui.cfg` - параметры консольного интерфейса.

Главный файл ida.cfg

IDA читает `ida.cfg` в начале запуска, чтобы назначить процессоры по расширениям файлов и настроить память. После выбора процессора файл читается повторно для остальных параметров.

Среди общих настроек особенно полезны:

• `VPAGESIZE` - настройка виртуальной памяти. При ошибке создания базы для очень большого файла обычно помогает увеличение значения и повторное открытие входного файла;
• `CREATE_BACKUPS` - создание резервных копий;
• `GRAPH_VISUALIZER` - имя внешней программы построения графов.

Здесь же находятся значения по умолчанию для форматирования листинга: `OPCODE_BYTES` задаёт число показываемых байтов кода, `INDENTATION` - отступ инструкций, `SHOW_SP` - показ смещения указателя стека, `SHOW_XREFS` - максимальное число перекрёстных ссылок в строке. Отдельные параметры отвечают за графический режим.

`MAX_NAMES_LENGTH` ограничивает длину имён программных адресов, но не стековых переменных. Значение по умолчанию равно 15, поскольку некоторые ассемблеры не принимают более длинные символы. Если листинг не планируется повторно собирать, предел можно безопасно увеличить.

Набор допустимых символов в пользовательских именах задаёт `NameChars`. По умолчанию разрешены буквы, цифры и `_$?@`. Например, для точки её нужно добавить в этот набор. Точку с запятой, двоеточие, запятую и пробел лучше не разрешать: они служат разделителями элементов строки дизассемблирования.

Для разбора заголовков C используются:

• `C_HEADER_PATH` - каталоги поиска зависимостей `#include`. Путь по умолчанию ориентирован на распространённую установку Visual Studio, поэтому для другого компилятора его следует изменить;
• `C_PREDEFINED_MACROS` - макросы препроцессора, которые IDA считает определёнными даже без доступного заголовочного файла.

Во второй половине ida.cfg находятся настройки процессорных модулей. Часто их единственная документация - комментарии в самом файле. Они, в частности, определяют исходные значения Processor options в диалоге загрузки.

В конце IDA пытается подключить <IDADIR>/cfg/idauser.cfg. Этот файл не входит в поставку: пользователь создаёт его самостоятельно. Его значения переопределяют ida.cfg. Такой способ безопаснее прямого редактирования основного файла и удобнее при обновлении IDA: достаточно перенести один пользовательский файл.

Файл графического интерфейса idagui.cfg

<IDADIR>/cfg/idagui.cfg содержит три основные группы параметров: поведение GUI, горячие клавиши и фильтры расширений в File > Open.

В Windows параметр HELPFILE указывает дополнительный файл справки, не заменяя основную справку IDA. По Ctrl+F1 IDA ищет в нём тему для слова под курсором, а при отсутствии совпадения открывает индекс. Так можно подключить, например, справочник инструкций x86. Поддерживается старый формат WinHelp .hlp, но не compiled HTML .chm. В Windows Vista и новее 32-разрядный WinHelp изначально отсутствовал.

DISPLAY_PATCH_SUBMENU показывает или скрывает Edit > Patch Program. Команды меняют данные в базе, а не исходный двоичный файл; подробности рассматриваются в главе 14.

DISPLAY_COMMAND_LINE управляет однострочной командной строкой IDA внизу окна. Её можно скрыть ради места, если не нужны короткие сценарии. Это строка языка сценариев IDA, а не оболочка операционной системы.

Раздел горячих клавиш связывает действия IDA с комбинациями. Он позволяет назначать клавиши командам без стандартных сочетаний, выбирать более запоминающиеся варианты и обходить конфликты с операционной системой или терминалом. Имена действий не всегда совпадают с меню. Например, Jump > Jump to Problem называется JumpQ. Найти соответствие часто помогает поиск имени действия во встроенной справке.

Пример назначений:

```
"Abort" = 0       // завершить IDA без сохранения
"Quit"  = "Alt-X" // выйти с сохранением
```

Незаключённый в кавычки 0 означает отсутствие клавиши. Комбинация записывается строкой в кавычках.

Параметр FILE_EXTENSIONS описывает известные типы файлов. Обычная запись содержит внутреннее имя, описание и один или несколько шаблонов, разделённых точкой с запятой:

```
CLASS_JAVA, "Java Class Files", "*.cla*;*.cls"
```

Можно объединять уже существующие ассоциации по шаблону имени:

```
EXE, "Executable Files", EXE_*
```

Собственный тип определяется так:

```
IDA_BOOK, "Ida Book Files", "*.book"
```

Одной записи в FILE_EXTENSIONS недостаточно: чтобы тип появился в File > Open, его имя нужно также добавить в DEFAULT_FILE_FILTER.

Последняя строка idagui.cfg подключает <IDADIR>/cfg/idauserg.cfg. В пользовательском файле можно переопределять GUI-параметры без изменения поставляемого idagui.cfg.

Файл консольного интерфейса idatui.cfg

`<IDADIR>/cfg/idatui.cfg` устроен почти так же, включая синтаксис горячих клавиш. Однако в консоли нет `DISPLAY_PATCH_SUBMENU`, `DISPLAY_COMMAND_LINE` и сложных ассоциаций File > Open.

Консольные версии предлагают параметр `NOVICE`, который включает упрощённый режим и скрывает часть сложных функций, в том числе почти все дополнительные представления.

Для автоматизации последовательностей клавиш поддерживаются макросы. Пользовательские макросы лучше хранить в `<IDADIR>/cfg/idausert.cfg`, консольном аналоге `idauserg.cfg`. Пример из `idatui.cfg`:

```
MACRO "Alt-H" // перейти к метке start
{
    "G"
    's' 't' 'a' 'r' 't'
    "Enter"
}
```

Макрос, вызываемый `Alt+H`, открывает диалог Jump to Address клавишей `G`, посимвольно вводит `start` и нажимает Enter. Запись `"start"` неверна: IDA воспримет её как имя клавиши. Макросы и novice mode в GUI отсутствуют.

Ошибка синтаксиса любого конфигурационного файла немедленно завершает IDA. Запуск станет возможен только после исправления файла.

Дополнительные параметры IDA

Многие настройки доступны только через интерфейс. В большинстве случаев изменения из меню Options относятся к текущей базе и записываются в неё при закрытии.

Цвета Options > Colors и шрифты Options > Font являются глобальными. В Windows они хранятся в реестре под `HKEY_CURRENT_USER\Software\Hex-Rays\IDA`, а в других системах - в `$HOME/.idapro/ida.reg` в собственном формате.

Если в сообщении выбрать Do not display this dialog box again, запись создаётся под `HKEY_CURRENT_USER\Software\Hex-Rays\IDA\Hidden Messages`. Чтобы вернуть скрытый диалог, нужно удалить соответствующее значение реестра.

Цвета IDA

Почти каждый элемент интерфейса можно перекрасить через Options > Colors.

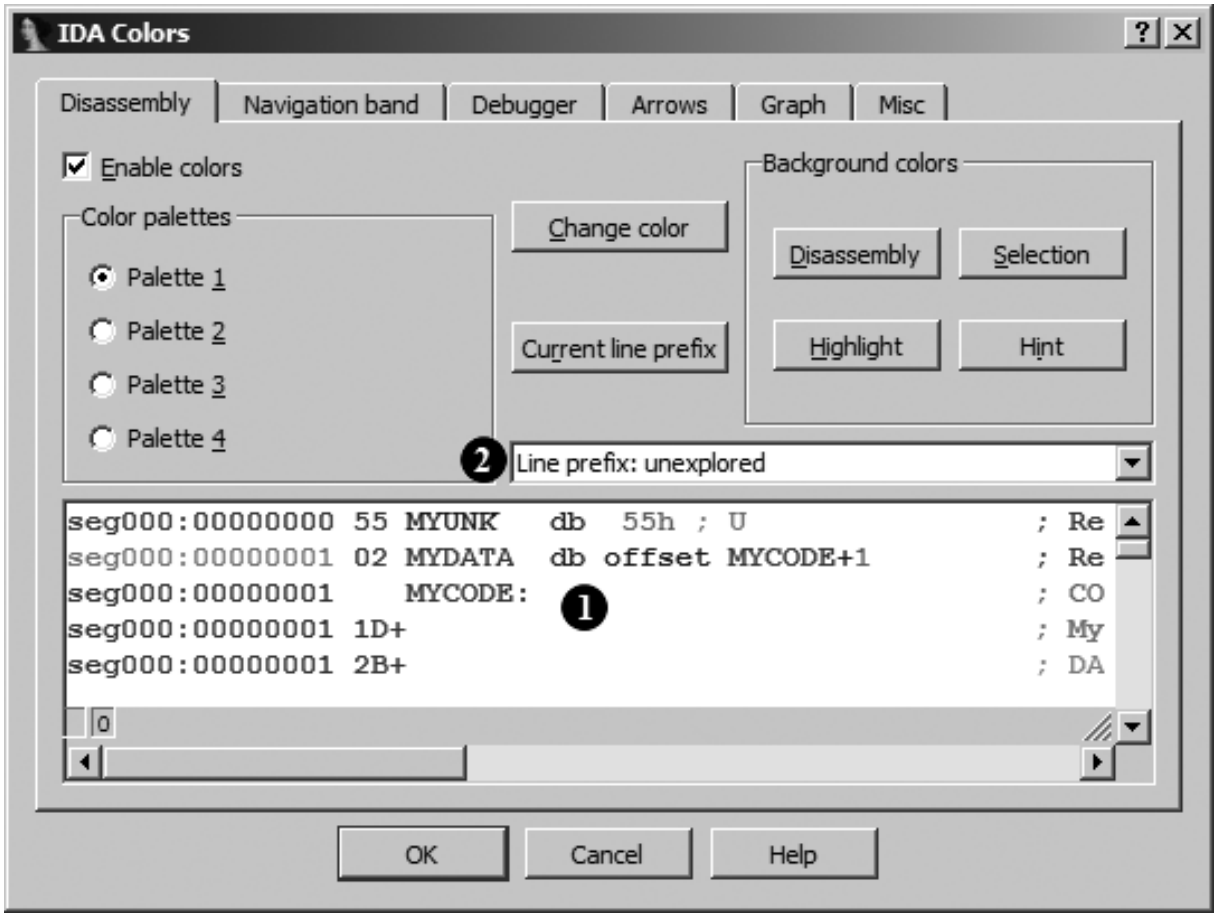

Рис. 11-1. Диалог выбора цветов

На вкладке Disassembly пример листинга показывает типы текста. Выбранный элемент и его тип видны в нижней части, а Change Color назначает ему цвет. Другие вкладки отвечают за навигационную полосу, отладчик, стрелки переходов, графы и окно сообщений. В Graph настраиваются узлы, заголовки и рёбра, а окраска дизассемблированного текста внутри графа по-прежнему задаётся на Disassembly.

Настройка панелей инструментов

GUI содержит более двух десятков панелей. Они обычно закреплены под меню. View > Toolbars предлагает Basic mode с семью панелями и Advanced mode со всеми панелями. Любую панель можно отсоединить, переместить или скрыть.

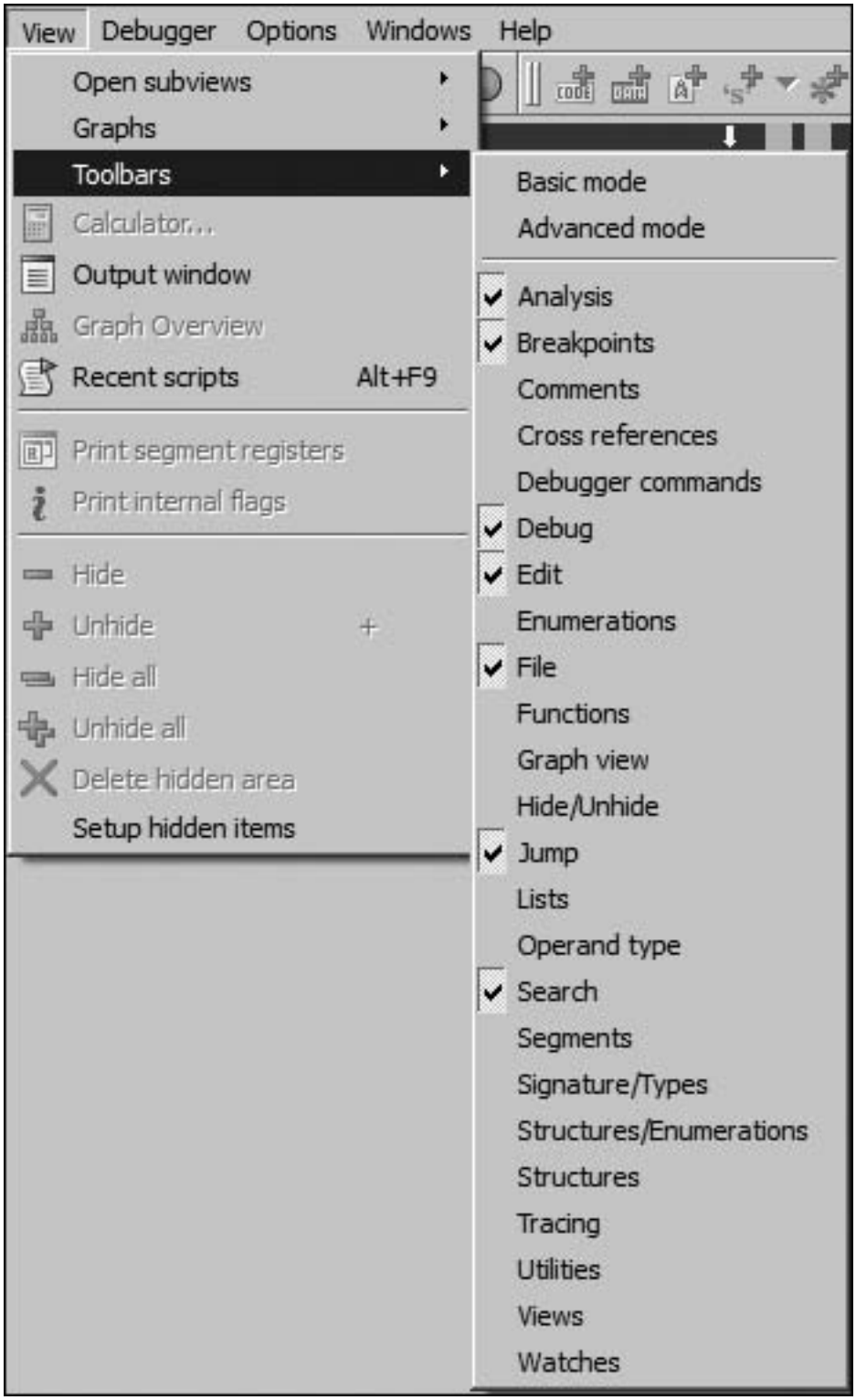

Рис. 11-2. Меню настройки панелей инструментов

То же меню появляется по правому щелчку в области закрепления. Отключение Main убирает все панели и освобождает максимум места под листинг.

Расположение панелей сохраняется в текущей базе. При открытии другой базы восстанавливается её собственный рабочий стол, а новая база получает текущий набор настроек по умолчанию.

Понравившуюся компоновку можно сохранить через Windows > Save Desktop.

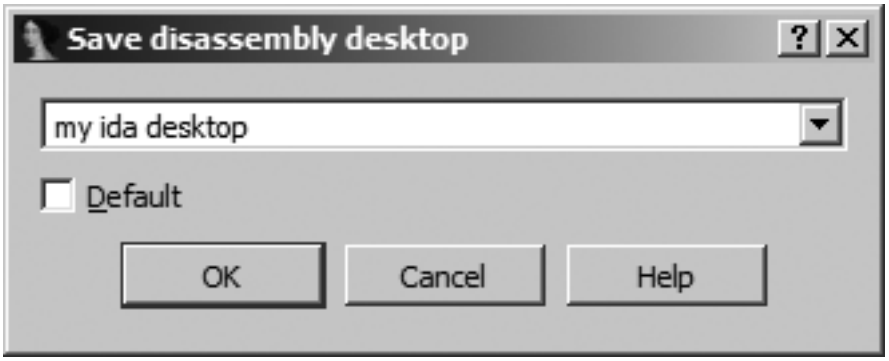

Рис. 11-3. Диалог Save Disassembly Desktop

Каждой конфигурации присваивается имя. Флажок Default делает её исходной для новых баз и целью Windows > Reset desktop. Сохранённый вариант загружается через Windows > Load Desktop. Несколько рабочих столов особенно полезны при смене мониторов, док-станций и проекторов с разными разрешениями.

Итоги

IDA сочетает системные конфигурационные файлы, безопасные пользовательские файлы переопределений, глобальные параметры интерфейса и настройки внутри базы. Основные точки настройки - `ida.cfg`, `idagui.cfg`, `idatui.cfg`, меню Options и сохранённые рабочие столы. Освоив их, можно адаптировать анализатор к своему процессу без повторной ручной настройки каждого проекта.

Глава 12. Распознавание библиотек с помощью FLIRT-сигнатур

После завершения первичного автоанализа важно отделить уникальный код программы от стандартных библиотечных и служебных последовательностей компилятора. Повторный анализ известных функций тратит время: их поведение проще узнать из документации или исходников. Особенно трудно различать части статически скомпонованного файла, где приложение и библиотеки объединены в один исполняемый модуль.

Технология FLIRT

FLIRT, Fast Library Identification and Recognition Technology, представляет набор алгоритмов сопоставления, которыми IDA узнаёт библиотечные функции. Шаблоны хранятся в `<IDADIR>/sig`. В поставке преобладают библиотеки распространённых Windows-компиляторов, хотя есть и некоторые сигнатуры других систем.

Файл `.sig` содержит сжатые данные и заголовок IDA. Имя файла не всегда объясняет его содержимое, поэтому создатель может встроить комментарий с названием библиотеки. В Unix его часто видно среди первых строк ASCII-текста:

```
strings sigfile | head -n 3
```

В IDA комментарии применённых сигнатур показывает View > Open Subviews > Signatures. Полный список файлов виден при ручной загрузке через File > Load File > FLIRT Signature File.

Применение FLIRT-сигнатур

При первом открытии файла IDA применяет к точке входа специальные startup signatures. Стартовый код разных компиляторов достаточно характерен, чтобы по нему определить средство сборки. После распознавания компилятора IDA автоматически загружает сигнатуры его стандартных библиотек.

Точка входа и main

Точка входа - адрес первой выполняемой инструкции, а не обязательно функция `main`. Формат файла определяет передачу аргументов загрузчиком, тогда как язык ожидает собственное соглашение. Поэтому перед `main` работает код инициализации, который IDA обычно называет `_start`.

В C++ этот код также вызывает конструкторы глобальных объектов. После возврата из `main` завершающий код вызывает их деструкторы и только затем заканчивает процесс.

Сигнатуры коммерческих компиляторов проще поставлять централизованно: число бинарных версий их библиотек ограничено. У GNU gcc и других открытых компиляторов варианты зависят от версии ОС, дистрибутива, исходников, компилятора и ключей оптимизации. Собирать сигнатуры для каждой `libc.a` практически невозможно, поэтому IDA поставляется лишь с небольшой их частью. Недостающие наборы можно создать самостоятельно средствами FLAIR.

Ручное применение нужно, если IDA распознала компилятор, но не имеет его библиотек, либо вообще не смогла определить компилятор. Последнее бывает в обфусцированном стартовом коде: сначала приходится частично снять обфускацию.

File > Load File > FLIRT Signature File открывает список доступных модулей.

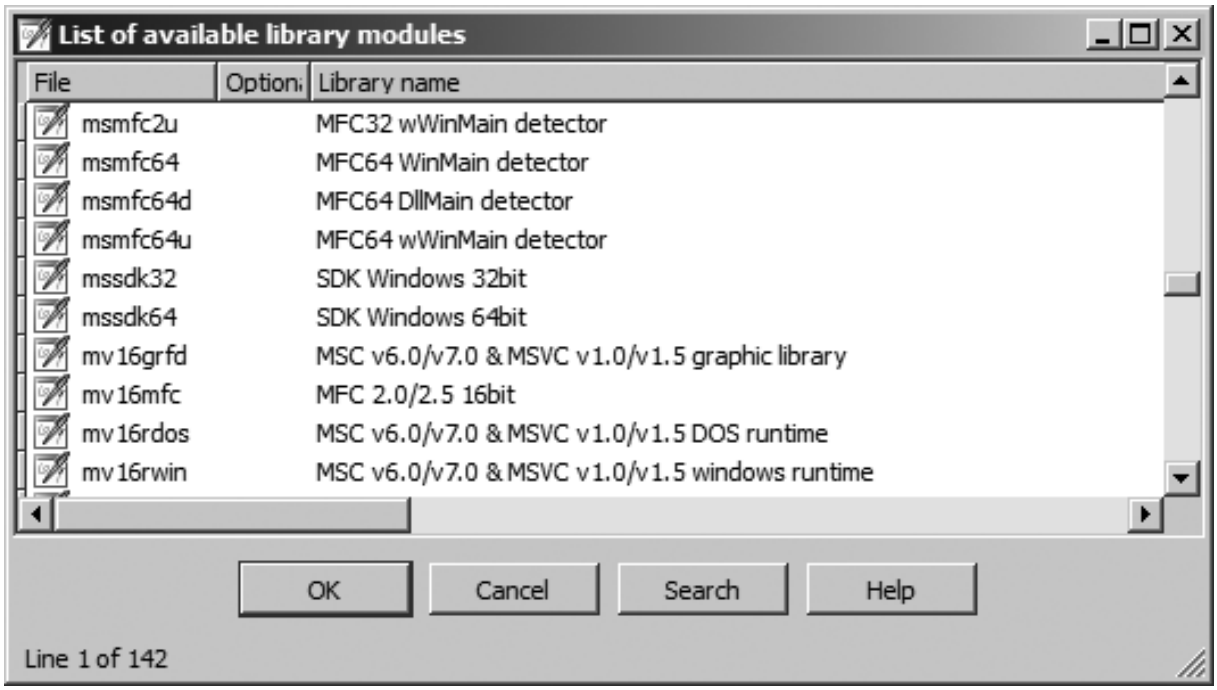

Рис. 12-1. Диалог выбора FLIRT-сигнатуры

В колонке File перечислены `.sig` из `<IDADIR>/sig`; выбрать другой каталог нельзя. Library name показывает встроенный автором комментарий. После выбора IDA сравнивает сигнатуры со всеми функциями базы. За один раз применяется один набор, поэтому для нескольких библиотек операцию повторяют.

Совпавшая функция отмечается как библиотечная и получает имя из сигнатуры. Автоматически переименовываются только функции с фиктивными именами IDA. Если пользователь уже присвоил функции имя, совпадение не заменит его. Поэтому сигнатуры лучше применять в самом начале анализа.

Даже неочищенный от символов файл полезно проверять FLIRT: имеющиеся имена не дают IDA достаточного основания отметить функцию как библиотечную.

До применения сигнатур навигатор статически скомпонованного файла не различает библиотеку и приложение.

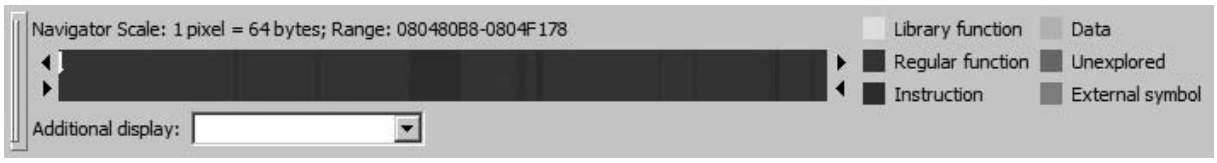

Рис. 12-2. Статическая компоновка без применённых сигнатур

После подходящего набора значительная часть диапазона получает цвет библиотечного кода и известные имена.

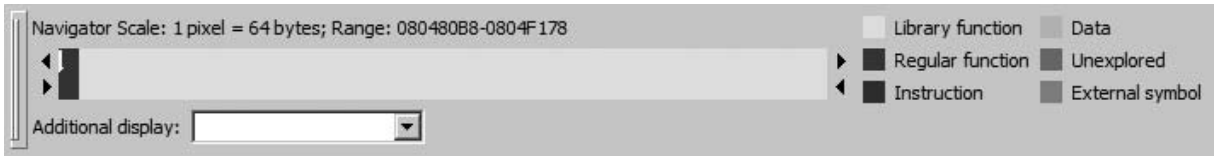

Рис. 12-3. Статически скомпонованный файл после применения сигнатур

Навигатор наглядно показывает эффективность набора. Неокрашенный участок слева, вероятно, содержит код самого приложения.

Один исполняемый файл может включать несколько библиотек. Например, после распознавания OpenSSL становится видна только одна светлая полоса, а большая часть кода остаётся неидентифицированной.

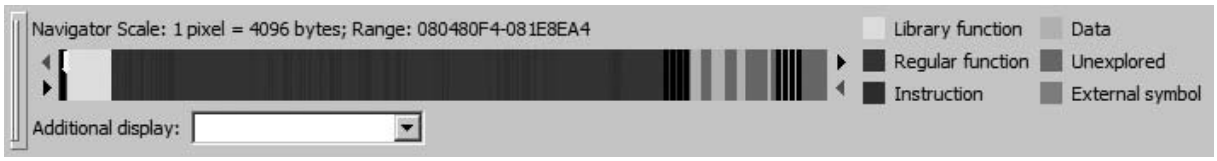

Рис. 12-4. Статический файл после первого из нескольких наборов сигнатур

При обычной компоновке сначала идёт код приложения, затем библиотеки. Поэтому узкая полоса слева может быть приложением, а область справа от OpenSSL - другими библиотеками.

После последовательного применения сигнатур `libc`, `libcrypto`, `libkrb5`, `libresolv` и других нераспознанных областей становится гораздо меньше.

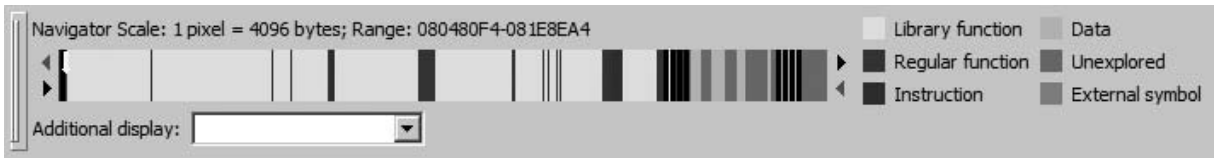

Рис. 12-5. Статический файл после применения нескольких сигнатур

Оставшиеся тёмные полосы требуют ручного анализа. В примере широкая средняя область оказалась неизвестной библиотекой, а узкая крайняя слева - кодом приложения. Выбирать дополнительные наборы помогают строки внутри файла и зависимости уже найденных библиотек.

Создание FLIRT-сигнатур

Hex-Rays распространяла комплект FLAIR, Fast Library Acquisition for Identification and Recognition. Его версия не должна быть новее установленной IDA. Архив лучше распаковывать в отдельный каталог, поскольку верхнего каталога в нём нет.

Основная документация:

• `readme.txt` - обзор процесса;
• `plb.txt` - статический библиотечный парсер `plb`;
• `pat.txt` - формат промежуточных pattern files;
• `sigmake.txt` - создание `.sig` и разрешение коллизий.

Исполняемые средства находятся в `bin`, а каталог `startup` содержит шаблоны стартового кода популярных компиляторов и форматов PE, ELF и других. До версии 6.1 FLAIR был доступен только для Windows, однако готовые `.sig` работали во всех вариантах IDA.

Общий процесс состоит из четырёх шагов:

1. Получить точную статическую библиотеку.
2. Подходящим парсером FLAIR создать `.pat`.
3. Преобразовать `.pat` в `.sig` программой `sigmake`.
4. Скопировать `.sig` в `<IDADIR>/sig`.

Поиск нужной статической библиотеки

Сначала нужно определить не только название, но и точную сборку библиотеки. В файле с символами помогают имена функций. В очищенном файле полезны уникальные строки версий, copyright и ошибки. Для командной `strings` следует применять `-a`, чтобы просканировать весь файл. Например:

```
OpenSSL 1.0.0b-fips 16 Nov 2010
```

Открытые исходники помогают понять код, но самостоятельно собранная библиотека часто отличается из-за версии компилятора и ключей. Лучшие сигнатуры получаются из библиотеки идентичной системы: нужны точные ОС, выпуск и дистрибутив.

Утилита `file` иногда сообщает достаточно много:

```
$ file sample_file_1
sample_file_1: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD),
statically linked, for FreeBSD 8.0 (800107), stripped
```

Здесь логично взять `libc.a` из FreeBSD 8.0. Для менее конкретного Linux-файла дополнительную подсказку может дать `strings`:

```
GCC: (GNU) 4.5.1 20100924 (Red Hat 4.5.1-4)
```

Метки GCC обычно переживают удаление символов. Но `file` не всегда точен: системно-нейтральное описание SYSV может относиться, например, к Solaris, и тогда снова нужны строки.

Создание pattern file

FLAIR содержит парсеры разных библиотечных форматов:

• `plb` - OMF, обычно Borland;
• `pcf` - COFF, обычно Microsoft;
• `pelf` - ELF в Unix-системах;
• `ppsx` - Sony PlayStation PSX;
• `ptmobj` - TriMedia;
• `pomf166` - Keil OMF 166.

Парсеру передают библиотеку и имя результата. Для FreeBSD:

```
$ ./pelf libc.a libc_FreeBSD80.pat
libc.a: skipped 1, total 1089
```

Сообщение означает, что одна функция пропущена и создано 1089 шаблонов. Набор параметров различается; запуск без аргументов показывает справку. Часто двух имён достаточно.

`.pat` - текстовый файл с одной функцией на строку. Начало строки кодирует до 32 первых байтов. Байты, меняющиеся из-за relocation, заменяются точками; короткая функция также дополняется точками до 64 символов. Далее идут CRC16, длина и имена внешних символов. Длинные функции со множеством ссылок могут порождать строки в десятки тысяч символов.

```
57568B7C240C8B742410FC8B4C2414C1E902F3A775108B4C241483E103F3A675 1E A55D 003E :0000 _memcmp
```

```
0FBC442404740340C39031C0C3..................................... 00 0000 000D :0000 _ffs
```

Существовали и сторонние генераторы по готовой базе, например плагин `IDB_2_PAT`. Они полезны, если библиотечного архива нет, но похожий код ожидается в будущих проектах.

Создание signature file

`sigmake` преобразует текстовые шаблоны в компактный собственный бинарный формат, удобный для быстрого поиска:

```
$ ./sigmake libssl.pat libssl.sig
```

Документация рекомендовала имена в формате DOS 8.3. Более длинные допустимы, но старый диалог показывает лишь первые восемь символов основы.

Главная сложность - коллизии, когда разные функции имеют одинаковый шаблон. IDA не может выбрать имя однозначно, поэтому `sigmake` вместо `.sig` создаёт `.exc`:

```
$ ./sigmake libc_FreeBSD80.pat libc_FreeBSD80.sig
libc_FreeBSD80.sig: modules/leaves: 1088/1024, COLLISIONS: 10
```

В начале нового `.exc` находятся четыре строки-инструкции, начинающиеся с `;`. Их обязательно удалить, иначе следующий запуск не прочитает файл. Затем для каждой группы конфликтов выбирают поведение:

• + перед одной функцией - применять это имя;
• - перед одной функцией - добавить только комментарий;
• отсутствие знака - не выбирать имя.

Например, `index` и `strchr`, а также `rindex` и `strrchr` могут иметь совершенно одинаковую реализацию. Допустимое решение:

```
+_index   00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3..............
_strchr   00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3..............
_rindex   00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3..........
_strrchr  00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3..........
_flsl     01 EF04 5531D289E58B450885C0741183F801B201740AD1E883C20183F80175F65D89D0
-_fls     01 EF04 5531D289E58B450885C0741183F801B201740AD1E883C20183F80175F65D89D0
```

Нельзя ставить знак более чем у одной функции группы. Если группа состоит из одной функции, знак не нужен. После повторной ошибки `sigmake` дописывает сведения и комментарии в существующий `.exc`; лишнее следует удалить и исправить исходное решение.

После сохранения `.exc` повторяют ту же команду. Успех отмечается отсутствием ошибок и появлением `.sig`, который копируют в `<IDADIR>/sig`.

Параметр `-n` встраивает понятное описание, отображаемое в диалоге выбора:

```
$ ./sigmake -n"FreeBSD 8.0 C standard library" libc_FreeBSD80.pat libc_FreeBSD80.sig
```

Название можно задать и директивой `.exc`, но ключ удобнее, поскольку файл исключений создаётся не всегда.

Стартовые сигнатуры

Startup signatures применяются сразу после загрузки для определения компилятора. Они сгруппированы по формату входного файла: для PE проверяются PE-шаблоны, после совпадения автоматически подключаются библиотеки найденного компилятора.

`sigmake` объединяет шаблоны стартовых функций в один файл формата. Каталог `startup` содержит готовые `.pat` и сценарий `startup.bat`. Например, `pe_vc.pat` описывает Visual Studio, а `pe_gcc.pat` - Cygwin/gcc. Новый PE-шаблон добавляют в существующий файл либо в новый с префиксом `pe_`, чтобы сценарий его нашёл.

Формат строки startup pattern немного отличается от обычного: совпадение может ссылаться на дополнительные наборы библиотечных сигнатур. В комплекте FLAIR он документировался только примерами.

Итоги

Автоматическое распознавание библиотек резко сокращает объём ручного анализа статически скомпонованных файлов. FLIRT применяет готовые шаблоны, а FLAIR позволяет получить собственные `.sig` из точных версий статических библиотек. Качество результата зависит от правильного выбора библиотеки, формата парсера и аккуратного разрешения коллизий.

Глава 13. Расширение знаний IDA

Полезный листинг - не просто мнемоники и операнды. Качество анализа повышают прототипы API, соглашения о вызовах, стандартные типы и комментарии. IDA получает эти сведения из библиотек типов, IDS-файлов и базы предопределённых комментариев. Дополнительные утилиты `idsutils` и `loadint` позволяют заполнить пробелы в поставке.

Дополнение сведений о функциях

IDA использует два основных источника:

- `.til` - библиотеки типов;
- `.ids` - описания экспортов библиотек.

Во время первичного анализа они помогают правильно учитывать стек, показывать имена и типы параметров, переименовывать связанные переменные и добавлять комментарии.

Библиотека `.til` хранит не только структуры данных, но также соглашения о вызовах и последовательности параметров. Это важно для разделяемых библиотек: один лишь импорт не сообщает IDA, кто очищает стек. Найденный прототип позволяет, например, распознать `stdcall` и скорректировать указатель стека после вызова.

Сравним два корректных объявления C:

```
LSTATUS _stdcall RegOpenKey(HKEY, LPCTSTR, PHKEY);
LSTATUS _stdcall RegOpenKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult);
```

Второе полезнее, поскольку содержит имена. Для `RegOpenKeyA` IDA может подписать каждую подготовку аргумента:

```
.text:00401006  00C  lea   eax, [ebp+hKey]
.text:00401009  00C  push  eax          ; phkResult
.text:0040100A  010  push  offset SubKey ; "Software\\Hex-Rays\\IDA"
.text:0040100F  014  push  80000001h    ; hKey
.text:00401014  018  call  ds:RegOpenKeyA
.text:0040101A  00C  mov   [ebp+var_8], eax
```

IDA также переименовала локальную `hKey` и глобальную `SubKey`. Если бы прототип содержал только типы, в комментариях появились бы типы вместо имён. Для `lpSubKey` уже показывается повторяемый комментарий со строкой, поэтому отдельная подпись параметра не нужна. После `stdcall` IDA уменьшает расчётную глубину стека на объём очищенных аргументов.

В таблице импорта прототип отображается как комментарий:

```
.idata:0040A000 ; LSTATUS __stdcall RegOpenKeyA(HKEY hKey, LPCSTR lpSubKey, PHKEY phkResult)
.idata:0040A000 extrn RegOpenKeyA:dword
```

Собственная библиотека типов нужна для распространённой графической, криптографической или другой библиотеки, которой нет в поставке IDA. При наличии заголовков прототипы можно добавить в локальную `.til` через File > Load File > Parse C Header File. Утилита `tilib` создаёт отдельную библиотеку; после копирования в `<IDADIR>/til` она становится доступной всем базам.

Когда исходников и заголовков нет, используются IDS utilities.

Ручное указание очищаемых байтов

Неизвестная импортированная `stdcall`-функция нарушает анализ указателя стека: IDA не знает ни соглашения, ни числа удаляемых при возврате байтов. Она пытается вывести поведение математически симплекс-методом, но пользователь может задать точное значение.

Нужно перейти к записи функции в таблице импорта и выбрать Edit > Functions > Edit Function или `Alt+P`.

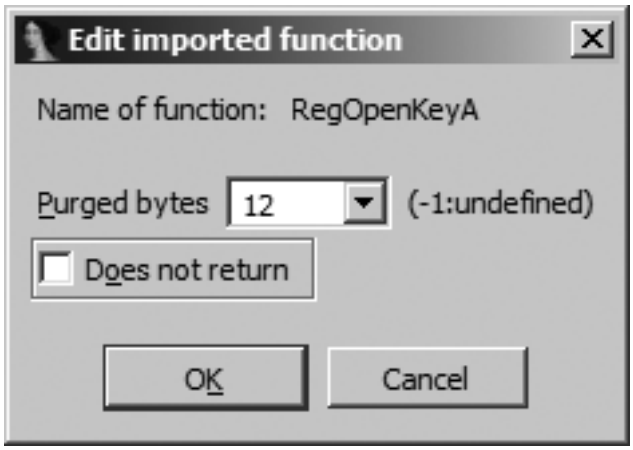

Рис. 13-1. Редактирование импортированной функции

У импортируемой функции нет тела, поэтому диалог короче обычного и не содержит сведений о кадре. Значение `-1` в Purged bytes означает, что IDA не знает, очищает ли функция стек. Введённое число включается в анализ всех мест вызова. Если поведение известно из `.til` или `.ids`, поле уже заполнено. Результат симплекс-метода сам по себе в нём не показывается.

IDS-файлы

`.ids` описывает экспорты разделяемой библиотеки. Для каждой функции могут быть заданы:

• имя;
• ordinal, то есть целочисленный индекс экспорта;
• признак `stdcall` и число очищаемых байтов;
• необязательный комментарий для мест ссылки.

Фактически `.ids` - сжатый текстовый `.idt`. При загрузке исполняемого файла IDA определяет зависимости и ищет подходящие описания в дереве `<IDADIR>/ids`.

IDS не обязан содержать полный прототип, поэтому имён и типов аргументов может не быть. Но точное соглашение и размер очистки позволяют правильно учитывать стек. Если DLL экспортирует декорированное имя, IDA иногда выводит сигнатуру из него. В целом `.til` информативнее, однако для его создания нужны объявления C.

Создание IDS-файлов

Комплект `idsutils` включал:

• `dll2idt` - извлечение экспортов из Windows DLL;
• `ar2idt` - обработка библиотек формата `ar`;
• `zipids` - упаковка `.idt` в `.ids`.

Парсеры создают по строке на экспорт и связывают ordinal с именем. Основной синтаксис `.idt`:

1. Положительное число в начале - ordinal.
2. `Name=function` - имя экспорта.
3. Особый ordinal 0 задаёт имя самой библиотеки.
4. `Pascal=N` означает `stdcall` и очистку `N` байтов.
5. `Comment=text` задаёт комментарий.

Примеры:

```
0 Name=advapi32.dll
483 Name=RegOpenKeyA Pascal=12
483 Name=RegOpenKeyA Pascal=12 Comment=Open a registry key
```

Чтобы описать `ssleay32.dll`, запускают:

```
$ ./dll2idt.exe ssleay32.dll
Convert DLL to IDT file. Copyright 1997 by Yury Haron. Version 1.5
File: ssleay32.dll ... ok
```

Имя результата берётся из метаданных DLL, поэтому регистр может отличаться: `SSLEAY32.idt`. Начало файла выглядит так:

```
ALIGNMENT 4
;DECLARATION
;
0 Name=SSLEAY32.dll
;
121 Name=BIO_f_ssl
173 Name=BIO_new_buffer_ssl_connect
122 Name=BIO_new_ssl
```

Парсер не определяет надёжно `stdcall`, число очищаемых байтов и смысл функции. Директивы `Pascal` и `Comment` добавляются вручную перед упаковкой.

```
$ ./zipids.exe SSLEAY32.idt
File: SSLEAY32.idt ... {219 entries [0/0/0]} packed
$ cp SSLEAY32.ids ../Ida/ids
```

После копирования IDA автоматически загружает описание при зависимости от `ssleay32.dll`. Неустановленный файл можно применить вручную через File > Load File > IDS File.

Файл `<IDADIR>/ida/idsnames` связывает разные источники знаний. Он умеет:

• сопоставлять настоящее имя библиотеки с коротким IDS-именем, например `libc.so.6` с `libc.ids`;
• загружать `.til` вместе с `.ids`, например `openssl.til` при `SSLEAY32.ids`;
• загружать `.ids` после применения `.sig`, например `SSLEAY32.ids` вместе с `libssl.sig`.

Примеры записей:

```
libc.so.6    libc.ids       +
SSLEAY32.ids SSLEAY32.ids   + openssl.til
libssl.sig   SSLEAY32.ids   +
```

В главе 15 рассматривается сценарный вариант генерации `.idt`, который использует анализ функций самой IDA и способен давать более подробные результаты.

Дополнение комментариев с помощью loadint

Автоматические комментарии IDA объясняют инструкции:

```
.text:08048654 lea ecx, [esp+arg_0] ; Load Effective Address
.text:08048658 and esp, 0FFFFFFF0h  ; Logical AND
```

Их источник - `<IDADIR>/ida.int`, где тексты сгруппированы по процессору и инструкции. Комплект `loadint` позволяет изменить или добавить их. В нём находятся исходные `.cmt` для процессорных модулей и документация.

Для x86 редактируют `pc.cmt`, затем пересобирают базу и копируют `ida.int` в основной каталог IDA:

```
$ ./loadint comment.cmt ida.int
Comment base loader. Version 2.04. Copyright (c) 1991-2011 Hex-Rays
17566 cases, 17033 strings, total length: 580575
```

Некоторые частые инструкции намеренно отключены, чтобы не загромождать листинг:

```
NN_ltr:   "Load Task Register"
//NN_mov: "Move Data"
NN_movsp: "Move to/from Special Registers"
```

Чтобы включить комментарий для `mov`, нужно убрать `//` и пересобрать файл.

`loadint` требуется `ida.hlp`. При ошибке `Can't initialize help system` справку копируют в каталог утилиты либо указывают `<IDADIR>`:

```
$ ./loadint -n <IDADIR> comment.cmt ida.int
```

`comment.cmt` является главным входом и связывает типы процессоров с их файлами комментариев. Процессорные `.cmt` связывают константы инструкций с текстами. Перечисления процессоров определены в `comment.cmt`, а полный набор инструкций - в `allins.hpp`.

Для совершенно нового процессора требуется:

1. Добавить в `allins.hpp` общее с процессорным модулем перечисление всех инструкций.
2. Создать `.cmt`, сопоставляющий каждой константе комментарий.
3. Добавить константу процессора и связь с `.cmt` в `comment.cmt`.
4. Запустить `loadint` и получить новую базу.

Эта работа тесно связана с созданием процессорного модуля, рассмотренным в главе 19.

Итоги

Один качественный `.til` или `.ids` экономит часы во всех последующих проектах с той же библиотекой. `idsutils` заполняет сведения об экспортах и стеке без исходников, а `loadint` расширяет пояснения инструкций. Эти средства особенно ценны за пределами самых распространённых платформ и API, которые невозможно полностью охватить стандартной поставкой IDA.

Глава 14. Патчинг бинарных файлов и ограничения IDA

IDA предназначена для понимания двоичного файла, а не для его редактирования. Команды Patch Program изменяют базу, а Create EXE File поддерживает далеко не все форматы. Тем не менее IDA хорошо подходит для проектирования и проверки небольших исправлений, которые затем можно перенести во входной файл внешним средством.

Меню Patch Program

В GUI меню Edit > Patch Program скрыто и включается через `idagui.cfg`; в консольной версии оно доступно по умолчанию.

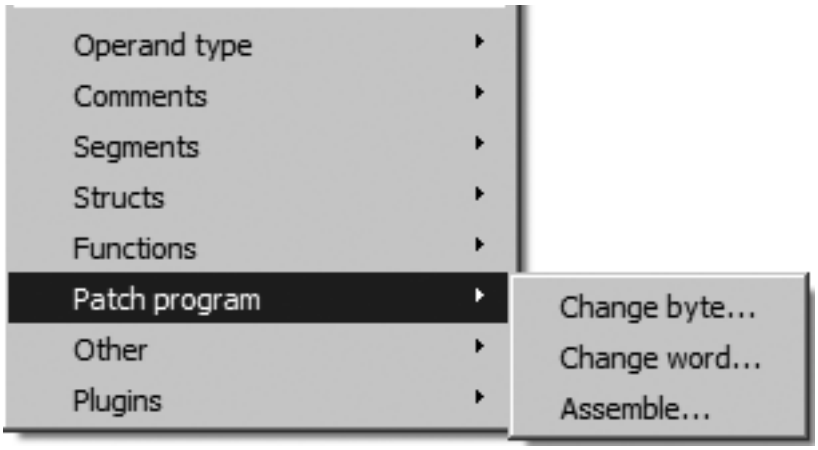

Рис. 14-1. Подменю Patch Program

Все три команды меняют байты базы IDA. После создания базы исходный файл для анализа больше не читается, поэтому точнее было бы назвать меню Patch Database. Зато листинг сразу показывает последствия изменения, а позднее разницу можно экспортировать.

Изменение отдельных байтов

Edit > Patch Program > Change Byte открывает редактор до 16 байтов от курсора.

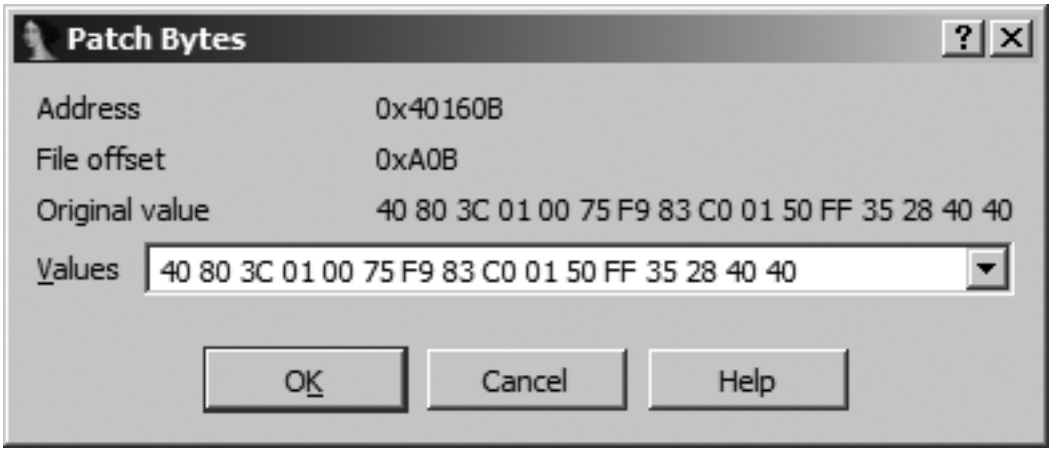

Рис. 14-2. Диалог Patch Bytes

Для следующего блока нужно закрыть окно, переместить курсор и открыть его снова. Address показывает виртуальный адрес, File offset - смещение в исходном файле. IDA сохраняет соответствие каждого байта исходному смещению, что затем позволяет построить патч.

Original value всегда содержит первоначальные байты, независимо от числа последующих правок. Автоматического отката всех изменений нет, хотя его легко реализовать сценарием. Начиная с IDA 5.5 удобнее редактировать прямо в Hex View.

Изменение слова

Patch Word изменяет одно двухбайтовое слово.

Рис. 14-3. Диалог Patch Word

Число отображается в естественном порядке байтов процессора: для x86 little-endian, для соответствующего MIPS-варианта может быть big-endian. Original value также остаётся исходным. На практике Hex View обычно удобнее.

Диалог Assemble

Edit > Patch Program > Assemble доступно только если процессорный модуль содержит встроенный ассемблер. Модуль x86 его поддерживал, MIPS - нет; в последнем случае IDA сообщает, что assembler не поддерживается.

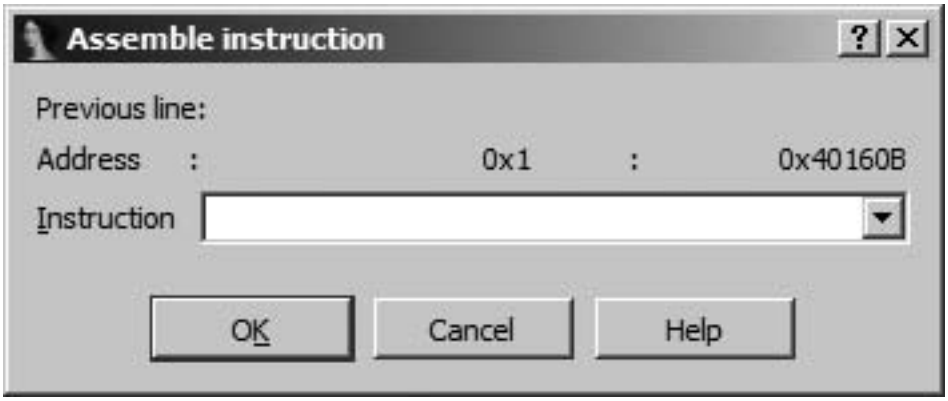

Рис. 14-4. Диалог Assemble Instruction

Вводится по одной инструкции в синтаксисе текущего листинга. После подтверждения полученные байты записываются от Address, а диалог остаётся открытым на следующем адресе. В x86 допустимы известные базе символы:

```
mov [ebp+var_4], eax
call sub_401896
```

IDA разрешает эти ссылки автоматически.

Ассемблер работает как редактор в режиме замены. Если новая инструкция короче старой, оставшиеся байты нужно заполнить, например `NOP`. Если длиннее, она перезапишет начало следующих инструкций. Вставить место со сдвигом содержимого нельзя, поэтому патч необходимо заранее рассчитать.

Крупное дополнение требует свободной области, которая уже присутствует в файле. Часто это padding для выравнивания секций. Например:

```
.text:0040963E ; [00000006 BYTES: COLLAPSED FUNCTION RtlUnwind]
.text:00409644 align 200h
.text:00409644 _text ends
.idata:0040A000 ; Section 2. (virtual address 0000A000)
```

Секция `.text` дополнена до файловой границы 512 байтов. Между `0x409644` и `0x409800` находится 444 байта нулей. Можно заменить часть функции переходом в эту область, выполнить добавленный код и вернуться.

В памяти `.idata` начинается лишь с `0x40A000`, поскольку PE-секции выровнены по 4 КБ. Диапазон `0x409800-0x40A000` теоретически даёт ещё 2048 байтов, но их нет в дисковом образе. Для использования пришлось бы:

1. Вставить в файл блок 2048 байтов.
2. Увеличить размер `.text` в заголовке PE.
3. Сдвинуть файловые позиции `.idata` и всех следующих секций в заголовках.

Это уже не простая перезапись и требует точного знания формата PE.

Выходные файлы IDA и создание патчей

File > Produce File предлагает форматы MAP, ASM, INC, LST, EXE, DIF и HTML. Их возможности различаются.

MAP

`.map` описывает секции и символы. При создании выбираются категории включаемых имён.

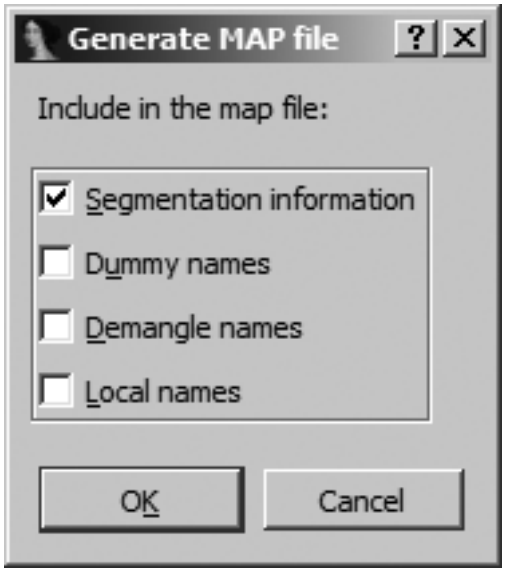

Рис. 14-5. Параметры создания MAP-файла

Адрес записывается как номер сегмента и смещение:

```
Start         Length     Name   Class
0001:00000000 000008644H .text  CODE
0002:00000000 000001DD6H .rdata DATA
0003:00000000 000002B84H .data  DATA

Address         Publics by Value
0001:00000000   _main
0001:00000069   _fprintf
```

`_fprintf` расположен по смещению `0x69` первого сегмента. Формат IDA совместим с Borland Turbo Debugger и полезен для восстановления имён при отладке файла без символов.

ASM

`.asm` пытается представить базу как исходный ассемблерный файл, включая структуры. Успешная повторная сборка зависит от полноты базы и поддержки синтаксиса выбранным ассемблером. Target assembler задаётся на вкладке Analysis в Options > General.

По умолчанию выводится вся база. В GUI можно заранее выделить диапазон мышью или `Shift` со стрелками. В консоли используют Anchor (`Alt+L`) и расширяют область стрелками.

INC

`.inc` содержит структуры и перечисления из окна Structures в виде, пригодном для подключения к ассемблерному исходнику.

LST

`.lst` - обычный текстовый снимок листинга. Его также можно ограничить выделенным диапазоном.

EXE

Самый многообещающий пункт обычно отвечает: `This type of output file is not supported.` Восстановить исполняемый файл из базы трудно, поскольку IDA загружает и сохраняет не все части оригинала. Например, PE-секция ресурсов `.rsrc` по умолчанию не попадает в базу. Таблицы символов, импорта и экспорта обрабатываются для анализа, но не сохраняются в удобной для обратной сборки форме.

Существовал комплект `pe_scripts` Атли Мара Гудмундссона. Сценарий `pe_write.idc` пытался выгружать рабочий PE. Последовательность была такой:

1. Открыть PE и снять Make imports section в диалоге загрузчика.
2. Запустить `pe_sections.idc`, чтобы отобразить в базу все исходные секции.
3. Внести изменения.
4. Запустить `pe_write.idc` и получить новый PE.

IDC рассматривается в главе 15.

DIF

`.dif` - наиболее удобный выход для переноса изменений. Это текстовый список всех изменённых байтов:

```
This difference file is created by The Interactive Disassembler

dif_example.exe
000002F8: 83 FF
000002F9: EC 75
000002FA: 04 EC
000002FB: FF 68
```

После заголовка и исходного имени каждая строка содержит файловое смещение, старый байт и новый байт. Здесь изменён диапазон `0x2F8-0x2FB`. Небольшая внешняя программа легко читает такой файл и применяет изменения к копии оригинала.

HTML

HTML-вывод представляет цветной LST с разметкой, похожей на окно IDA. В рассматриваемой версии он не создавал гиперссылок по именам, поэтому навигация оставалась такой же неудобной, как в простом тексте.

Итоги

IDA не является редактором двоичных файлов. Она позволяет безопасно спроектировать небольшие замены в базе, увидеть новый листинг и экспортировать точные файловые смещения через DIF. Сам патч применяется внешним инструментом или сценарием. Изменения, требующие вставки секций и перестройки заголовков, выходят далеко за пределы простого меню Patch Program.

Глава 15. Сценарии IDA

Ни одно приложение не покрывает все задачи пользователя. IDA решает эту проблему встроенными сценариями: от одной команды до программ, автоматизирующих повторяемые действия и сложный анализ базы. Сценарий можно воспринимать и как макрос, и как язык запросов к содержимому IDB.

В рассматриваемой версии доступны два языка: встроенный C-подобный IDC и Python через плагин IDAPython.

Запуск сценариев

File > Script File запускает отдельный файл. После первого запуска он попадает в View > Recent Scripts.

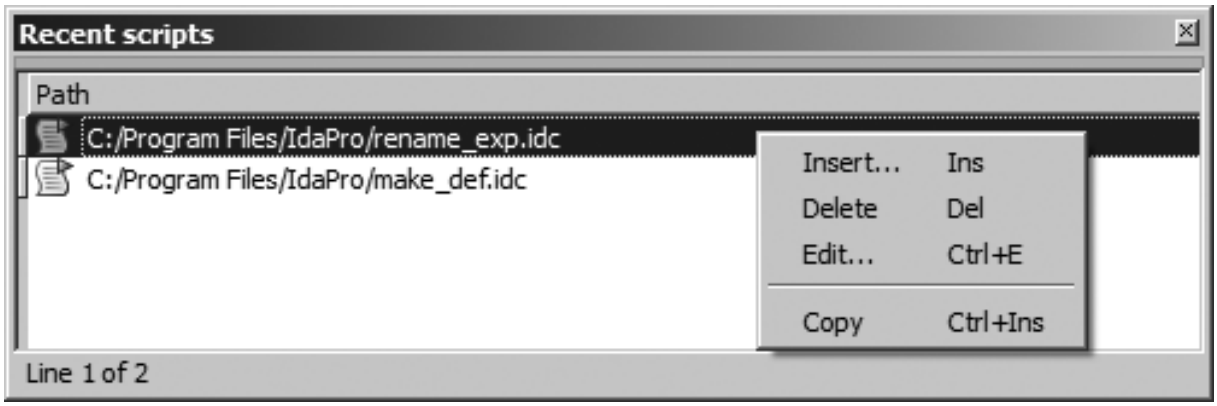

Рис. 15-1. Окно последних сценариев

Двойной щелчок запускает сценарий. Контекстное меню удаляет запись либо открывает файл в редакторе, выбранном на вкладке Misc в Options > General.

Для нескольких выражений служат File > IDC Command и, при установленном Python, File > Python Command.

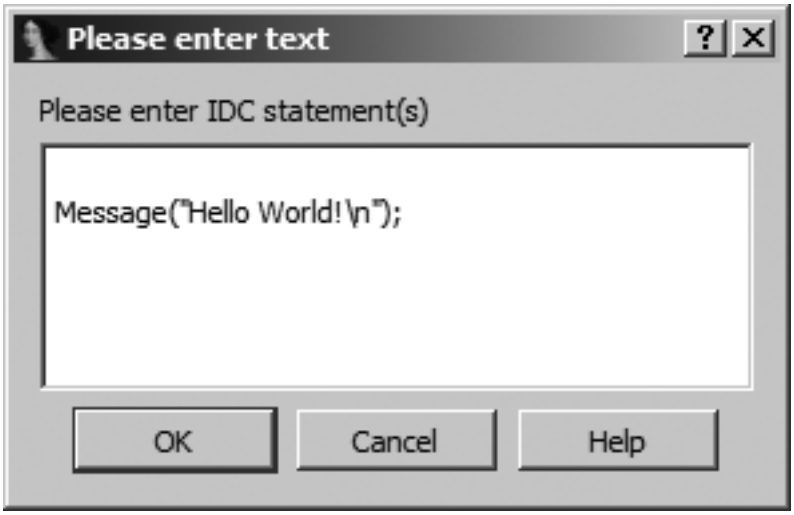

Рис. 15-2. Диалог ввода сценария

В GUI также есть однострочная командная строка под Output. Её наличие задаёт `DISPLAY_COMMAND_LINE` в `<IDADIR>/cfg/idagui.cfg`.

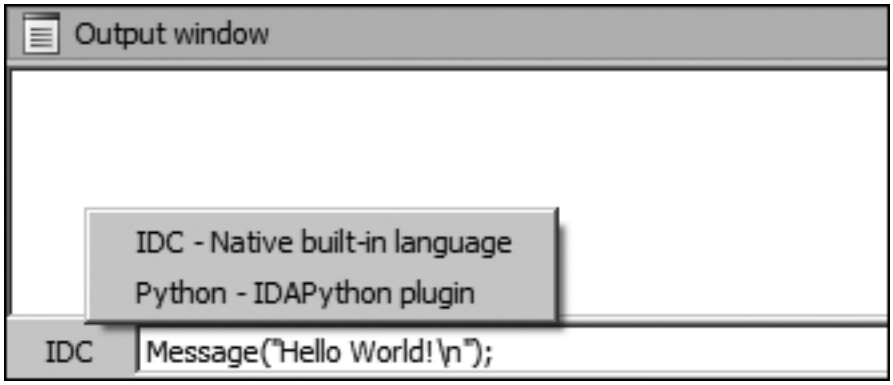

Рис. 15-3. Командная строка IDA

Метка слева выбирает IDC или Python. Несколько выражений разделяются точкой с запятой, стрелка вверх открывает историю.

Язык IDC

В справке IDA есть разделы IDC language и Index of IDC functions. Синтаксис происходит от C, а начиная с IDA 5.6 включает классы, ссылки и исключения в духе C++.

Переменные и выражения

IDC имеет динамическую слабую типизацию. Основные значения - целые `long`, строки и числа с плавающей точкой. С IDA 5.6 появились объекты, ссылки и ссылки на функции. Строка является самостоятельным типом, поэтому не нужно выделять память и следить за завершающим нулём.

Локальные переменные объявляются через `auto` до использования:

```
auto addr, reg, val;
auto count = 0;
```

Поддерживаются комментарии `/* ... */` и `//`. Все простые выражения заканчиваются `;`.

Глобальные переменные объявляются `extern` внутри или вне функции, но без инициализатора:

```
extern outsideGlobal;

static main() {
    extern insideGlobal;
    outsideGlobal = "Global";
    insideGlobal = 1;
}
```

Они создаются при первом обращении и живут до завершения сеанса IDA, даже при смене баз.

Почти все арифметические и логические операторы C доступны, включая `?:`. Операторов составного присваивания +=, *=, >>= нет. Целые считаются знаковыми, сравнение всегда знаковое, а >> выполняет арифметический сдвиг. Логический сдвиг вправо приходится маскировать:

```
result = (x >> 1) & 0x7fffffff;
```

Присваивание строки копирует значение, а + объединяет строки. В IDA 5.6 появились slices:

```
auto str = "String to slice";
```

```
auto s1 = str[7:9]; // "to"
auto s2 = str[:6];  // "String"
auto s3 = str[10:]; // "slice"
auto s4 = str[5];   // "g"
```

Традиционных массивов, указателей, структур и объединений в старом IDC нет. Строковые slices частично заменяют индексируемый массив.

Управляющие конструкции и функции

IDC поддерживает основные конструкции C, кроме `switch`. Из-за отсутствия `+=` цикл с шагом 2 пишется так:

```
for (i = 0; i < 10; i = i + 2) {}
```

С IDA 5.6 доступны `try`, `catch` и `throw`. Блоки заключаются в `{}`. Переменные должны объявляться в начале блока, но их область видимости соблюдается необычно слабо: переменная остаётся доступной после блока и может существовать со значением 0 даже из невыполненной ветви. Между разными функциями локальные переменные всё же изолированы.

Пользовательские функции разрешены только в `.idc` и объявляются `static` без типов:

```
static my_func(x, y, z) {
    auto a, b, c;
}
```

До 5.6 все параметры передавались по значению. Позднее ссылку указывает вызывающая сторона оператором &:

```
my_func(q, r, s);
my_func(q, &r, s);
```

Во втором вызове изменение формального `y` меняет `r`. Тип результата не объявляется; разные ветви могут возвращать разные типы. Функция без явного `return` возвращает 0.

В диалоге IDC нельзя определять функции: IDA уже оборачивает введённые выражения в собственную функцию, а вложенные определения запрещены.

Ссылки на функции можно передавать и возвращать:

```
static getFunc() { return Message; }
static useFunc(func, arg) { func(arg); }

static main() {
    auto f = getFunc();
    f("Hello World\n");
    useFunc(f, "Print me\n");
}
```

Объекты

IDC 5.6 ввёл корневой класс `object` и одиночное наследование. Все члены фактически публичны. В объявлении класса перечисляются методы, а поля возникают при присваивании:

```
class ExampleClass {
    ExampleClass(x, y) {
        this.a = x;
        this.b = y;
    }
    ~ExampleClass() {}
    foo(x) { this.a = this.a + x; }
};
```

```
static main() {
    auto ex = ExampleClass(1, 2);
    ex.foo(10);
    ex.z = "string";
}
```

Запись `ExampleClass ex;` недопустима. Объектная переменная создаётся присваиванием результата конструктора. `ex.z` добавляет поле конкретному объекту, но не классу в целом.

Отдельная программа IDC

Файл должен как минимум определять `main()` без аргументов. Обычно подключают полезные макросы `idc.idc`:

```
#include <idc.idc>

static main() {
    // действия
}
```

Препроцессор понимает `#include`, `#define`, `#ifdef`, `#else`, `#endif` и `#undef`. Предопределённые символы `_NT_`, `_LINUX_`, `_MAC_`, `_GUI_`, `_TXT_` позволяют учитывать среду.

Ошибки разбора включают синтаксис, неизвестную переменную и неверное число аргументов. IDC сообщает только первую ошибку, иногда точно, иногда лишь `Syntax error near: <END>`. Ошибка выполнения сразу прекращает сценарий. Встроенного отладчика нет, поэтому основное средство диагностики - вывод сообщений.

Если сценарий работает больше нескольких секунд, IDA показывает окно отмены.

Рис. 15-4. Диалог остановки сценария

Это единственный штатный способ прервать зависший или бесконечный IDC. Обработка исключений помогает завершаться аккуратнее.

Постоянные данные IDC

Global persistent arrays - не обычные массивы, а именованные разреженные объекты внутри базы. Они сохраняются между вызовами сценариев и сеансами IDA. Один индекс одновременно может иметь целое и строковое значение, но не float. Внутренняя реализация основана на `netnode`.

Основные функции:

- `CreateArray(name)` создаёт массив и возвращает handle, либо `-1`, если имя существует;
- `GetArrayId(name)` находит handle, либо `-1`;
- `SetArrayLong(id, idx, value)` и `SetArrayString(id, idx, str)` записывают значение;
- `GetArrayElement(AR_LONG|AR_STR, id, idx)` читает выбранный тип;

• `DelArrayElement(tag, id, idx)` удаляет один тип по индексу;
• `DeleteArray(id)` удаляет объект целиком;
• `RenameArray(id, newname)` переименовывает его.

Такие объекты имитируют глобальные структуры и сохраняют результаты между запусками. Если данные относятся лишь к одному запуску, массив следует удалить перед завершением, иначе он останется в IDB.

Привязка IDC к горячей клавише

При старте выполняется `<IDADIR>/idc/ida.idc`. В него добавляют подключение собственного файла и вызов `AddHotkey`:

```
#include <idc.idc>
#include <my_amazing_script.idc>

static main() {
    AddHotkey("z", "MyAmazingFunc");
}
```

Если клавиша занята меню или плагином, функция молча завершается неудачей. Стандартный каталог include - `<IDADIR>/idc`; для другого места нужен полный путь. Функцию собственного сценария нельзя оставить с именем `main`, иначе она конфликтует с `ida.idc`.

Полезные функции IDC

Полный список находится в справке. Имена не полностью последовательны: чтение может называться `GetXXX` или иначе, изменение - `SetXXX`, `MakeXXX` и другими способами.

Байты базы

• `Byte(addr)`, `Word(addr)`, `Dword(addr)` читают 1, 2 и 4 байта;
• `PatchByte`, `PatchWord`, `PatchDword` записывают соответствующий размер;
• `isLoaded(addr)` проверяет наличие данных.

Порядок байтов берётся из процессорного модуля. Записываемое число обрезается до младших байтов. При неверном адресе чтение возвращает соответственно `0xFF`, `0xFFFF` или `0xFFFFFFFF`, поэтому перед ним лучше вызывать `isLoaded`.

Взаимодействие с пользователем

• `Message(format, ...)` печатает форматированный текст в Output;
• `print(...)` печатает строковые представления;
• `Warning(format, ...)` показывает диалог;
• `AskStr`, `AskFile`, `AskYN` запрашивают строку, файл или выбор yes/no/cancel;
• `ScreenEA()` возвращает адрес курсора;
• `Jump(addr)` переводит листинг к адресу.

Есть дополнительные `AskXXX` для чисел и других типов.

Строки и файлы

form до IDA 5.6 и sprintf после неё форматируют строку. atol и xtol разбирают десятичное и шестнадцатеричное число, ltoa форматирует основание 2, 8, 10 или 16. ord, strlen, strstr, substr работают с символами и подстроками.

Файловый API похож на C: fopen, fclose, filelength, fgetc, fputc, fprintf, writestr, readstr. writelong/readlong и writeshort/readshort явно получают порядок байтов. loadfile копирует диапазон файла в базу, savefile - из базы в файл.

Имена и функции

• Name(addr) возвращает глобальное имя;
• NameEx(from, addr) также видит локальное имя в контексте функции;
• MakeNameEx(addr, name, flags) назначает имя с атрибутами;
• LocByName(name) и LocByNameEx(funcaddr, localname) возвращают адрес или BADADDR;
• GetFunctionAttr(addr, attrib) читает атрибут функции, например FUNCATTR_END;
• GetFunctionName(addr) возвращает имя содержащей функции;
• NextFunction и PrevFunction переходят между функциями.

Перекрёстные ссылки

Для исходящих ссылок кода используются Rfirst(from) и Rnext(from, current), для входящих - RfirstB(to) и RnextB(to, current). XrefType() сразу после получения ссылки сообщает fl_CN/fl_CF для вызова, fl_JN/fl_JF для перехода или fl_F для обычного потока.

Аналогично данные обходятся через Dfirst, Dnext, DfirstB, DnextB. Типами являются, среди прочих, dr_O для взятия адреса, dr_W для записи и dr_R для чтения. Следующая операция поиска перезаписывает внутренний тип, поэтому XrefType нужно вызывать сразу.

Форматирование и поиск

MakeUnkn снимает определение, MakeCode создаёт инструкцию, MakeByte/MakeWord/MakeDword - данные, MakeComm - комментарий, MakeFunction - функцию, MakeStr - строку. BADADDR в качестве конца позволяет IDA найти конец автоматически.

FindCode, FindData, FindBinary, FindText используют флаги SEARCH_DOWN, SEARCH_NEXT, SEARCH_CASE. SEARCH_NEXT пропускает текущее совпадение, но не определяет направление.

Компоненты строки читают:

• GetDisasm(addr) - текст инструкции с комментариями;
• GetMnem(addr) - мнемоника;
• GetOpnd(addr, opnum) - отображение операнда;
• GetOpType и GetOperandValue - его тип и значение;
• CommentEx(addr, 0|1) - обычный или повторяемый комментарий.

Примеры IDC

Перечисление функций и инструкций

Функции обходятся от `NextFunction(0)` до `BADADDR`. Для каждой можно получить конец, размер локальных данных и frame. Смещение члена `" r"` обозначает сохранённый адрес возврата, первый аргумент идёт ещё через 4 байта. Отсутствие этого члена помогает отфильтровать импорт без кадра.

Число инструкций функции под курсором можно подсчитать между `FUNCATTR_START` и `FUNCATTR_END`, переходя через `FindCode(inst, SEARCH_DOWN | SEARCH_NEXT)`. Такой адресный обход предполагает непрерывную функцию; chunked-функции правильнее обходить по потоку ссылок.

Вызовы и вызывающие функции

Чтобы перечислить вызовы из функции, для каждой инструкции обходят `Rfirst`/`Rnext` и оставляют типы `fl_CN` и `fl_CF`. Такой подход не зависит от архитектурной мнемоники вызова.

Обратная задача полезна для опасных `strcpy` и `sprintf`: `LocByName` находит цель, `RfirstB`/`RnextB` - все места вызова, `GetFunctionName` - вызывающую функцию. В ELF нужно учитывать PLT/GOT и преобразование недопустимой точки в подчёркивание: ссылки обычно ведут к PLT-символу, а не к одноимённому символу секции `extern`.

Генерация IDT

После открытия DLL сценарий может создать `.idt`: `GetEntryPointQty` возвращает число экспортов, `GetEntryOrdinal` - ordinal, `GetEntryPoint` - адрес, `GetInputFile` - имя входного файла. `FUNCATTR_ARGSIZE` позволяет добавить `Pascal=N`, если IDA определила очищаемые аргументы. Такой результат информативнее простого `dll2idt`.

Поиск аргументов GCC

GCC новее 3.4 часто передавал x86-аргументы записью в стек, а не `push`:

```
mov [esp+8], 0
mov [esp+4], 1
mov [esp], 2
call _socket
```

Старые версии IDA могли не подписать параметры. Сценарий проверяет `FUNC_FRAME`, обходит инструкции, распознаёт операнды `[esp]` и `[esp+disp]` через `GetOpType`, `GetOperandValue`, `GetOpnd`, затем создаёт комментарии `arg_0`, `arg_4`, `arg_8`. Более развитый вариант использует известный прототип и пишет `int domain`, `int type`, `int protocol`.

Эмуляция поведения кода

Сценарий может безопасно воспроизвести простой декодер без запуска подозрительной программы или без целевой аппаратуры. Пример из CTF расшифровывал 0x3C2 байта от `0x804B880` XOR-значением `0x4B`:

```
auto i, addr;
for (i = 0; i <= 0x3C1; i++) {
    addr = 0x804B880 + i;
    PatchByte(addr, Byte(addr) ^ 0x4B);
}
```

Механический метод таков: создать переменные для регистров и стека, переводить инструкции по одной, читать память через `Byte`/`Word`/`Dword`, писать через `PatchXXX`, а неизвестный цикл сначала выразить как `while (1)` с `break`. Вызовы функций значительно усложняют эмуляцию, поскольку нужно воспроизвести их эффекты и результат. При необходимости исходную базу можно не менять, а вывести декодированные данные сообщениями или в файл.

IDAPython

IDAPython встраивает интерпретатор Python и предоставляет весь IDC API, значительную часть SDK, стандартные структуры данных Python и его модули. Начиная с IDA 5.4 плагин поставлялся штатно и включался при обнаружении совместимого Python. Windows-установщик включал нужную версию, Linux и OS X требовали установить её отдельно.

Три основных модуля:

• `idc` - аналоги встроенных функций IDC;
• `idautils` - удобные генераторы функций, инструкций и ссылок;
• `idaapi` - низкоуровневый API SDK, который нужно импортировать явно.

`idc` и `idautils` импортируются автоматически. В IDA живёт один интерпретатор до закрытия приложения, поэтому импортированные модули, переменные и определения функций сохраняются между запусками сценариев.

Примеры Python

Перечисление функций упрощает генератор `Functions()`:

```
for f in Functions():
    name = Name(f)
    end = GetFunctionAttr(f, FUNCATTR_END)
    Message("Function: %s, starts at %x, ends at %x\n" % (name, f, end))
```

Функцию под курсором можно получить SDK-функцией, а инструкции - `FuncItems`:

```
from idaapi import *
func = get_func(here())
if func is not None:
    count = 0
    for ea in FuncItems(func.startEA):
        count += 1
```

Исходящие ссылки возвращает `XrefsFrom` как объекты с полями `type` и `to`:

```
for ea in FuncItems(func.startEA):
    for xref in XrefsFrom(ea, 0):
        if xref.type == fl_CN or xref.type == fl_CF:
            Message("%s calls %s from 0x%x\n" %
                    (Name(func.startEA), Name(xref.to), ea))
```

Генератор IDT почти повторяет IDC, но для файла предпочтителен штатный Python API и `with open(...)`, а не устаревшие IDC-функции.

Итоги

IDC удобен для короткой автоматизации и доступен без внешней среды. IDAPython добавляет полноценный язык, коллекции, модули и часть SDK, поэтому лучше подходит для развивающихся инструментов анализа. Сценарии быстро закрывают локальные задачи, но для максимального доступа и производительности нужны компилируемые расширения из следующих глав.

Глава 16. Комплект разработчика IDA SDK

Интеллект IDA распределён между процессорными, загрузочными и другими модулями. Когда IDC или Python недостаточно быстры либо не открывают нужную часть внутреннего API, расширение компилируют с IDA SDK.

IDC сам построен поверх SDK: всё доступное в IDC можно реализовать через SDK, но обратное неверно. SDK предоставляет заголовки и C++-библиотеки для плагинов, загрузчиков новых форматов и процессорных модулей.

IDA традиционно была однопоточной. До 5.5 дополнительный поток не должен одновременно обращаться к базе. Начиная с 5.5 вызовы SDK из фоновых потоков требуется ставить в очередь через `exec_request_t` и `execute_sync` из `kernwin.hpp`. Блокирующая операция делает интерфейс неотзывчивым.

Введение в SDK

Архив назывался по совместимой версии, например `idasdk61.zip` для IDA 6.1. Документация состояла главным образом из верхнего `readme.txt`, README для loaders, processors и plugins, а также комментариев в заголовках.

До SDK 4.9 интерфейсы могли заметно меняться. С 4.9 Hex-Rays стабилизировала API и бинарную совместимость вперёд: старый модуль обычно работал в новой IDA. Новые возможности, разумеется, не работают в старой версии. Иногда функции переименовывались или устаревали; макросы позволяют запретить deprecated API и обнаружить его при сборке.

Установка и структура

До 5.4 архив не имел верхнего каталога, поэтому его следовало распаковывать в отдельный `idasdk53` и не смешивать с IDA. Поздние архивы уже содержали `idasdk61`. Расположение относительно `<IDADIR>` не важно; далее корень обозначается `<SDKDIR>`.

- `bin` - результаты сборки примеров и postprocessor процессорных модулей;
- `etc` - исходники двух служебных программ и их готовые версии;
- `include` - весь опубликованный API, структуры и фактическая документация;
- `ldr` - примеры загрузочных модулей;
- `lib` - библиотеки для сочетаний платформ, компиляторов и разрядности;
- `module` - примеры процессорных модулей;
- `plugins` - примеры плагинов;
- корень - make-файлы, общий README и `install_xxx.txt` для компиляторов.

Подкаталоги `lib` имели имена вроде `x86_win_vc_32` для 32-разрядной Windows/Visual Studio и `x64_mac_gcc_64` для 64-разрядной OS X/gcc.

Среда сборки

Исходные примеры ориентировались на Borland C++ Builder 4.0, но `install_xxx.txt` описывали другие варианты. В `plugins/vcsample` были проекты Visual Studio, а `install_visual.txt` объяснял Visual C++ Express 2005.

Для Unix, Linux или MinGW сценарий `idamake.pl` преобразовывал Borland makefiles в Unix-формат; детали находились в `install_linux.txt`. Штатные сценарии ожидали переменную среды `IDA`, указывающую на `<SDKDIR>`. Её можно было задать глобально либо в `allmake.mak` и `allmake.unx`.

API IDA

Единого индекса API не было. Ответ обычно находили поиском по `<SDKDIR>/include`, по аналогии с IDC или через форум. Названия одинаковых операций IDC и SDK часто различаются, поэтому знание назначения заголовков особенно важно.

`ida.hpp` - главный include любого проекта. SDK намеренно блокирует опасные `strcpy`, `sprintf` и похожие функции. Для их возврата до `ida.hpp` определяется:

```
#define USE_DANGEROUS_FUNCTIONS
#include <ida.hpp>
```

Без него компоновка или компиляция может ссылаться на `dont_use_snprintf`. Предпочтительны безопасные `qstrncpy`, `qsnprintf` из `pro.h`.

Стандартные `stdin`, `stdout`, `fopen`, `fwrite`, `fprintf` также ограничены, отчасти из-за Borland. SDK предлагает `qfopen`, `qfprintf` и другие `qXXX`. Для стандартного API до подключения `fpro.h` определяется `USE_STANDARD_FILE_FUNCTIONS`.

Основные заголовки

- `area.hpp` - диапазон `area_t` с включительным `startEA` и исключительным `endEA`;
- `auto.hpp` - очередь автоанализа;
- `bytes.hpp` - байты базы, flags операндов и комментарии;
- `dbg.hpp` - управление отладчиком;
- `entry.hpp` - точки входа и экспорты;
- `expr.hpp` - IDC-значения, вызов и добавление функций IDC;
- `fpro.h` - альтернативный файловый API;
- `frame.hpp` - стековые кадры;
- `funcs.hpp` - функции, их chunks и FLIRT;
- `gdl.hpp` - графы DOT и GDL;
- `ida.hpp` - `idainfo` и глобальная `inf`;
- `idp.hpp` - основы процессорного модуля, глобальные `ph` и `ash`;
- `kernwin.hpp` - интерфейс, запросы, переходы и горячие клавиши;
- `lines.hpp` - форматированные цветные строки листинга;
- `loader.hpp` - `loader_t`, `plugin_t`, загрузка файлов и активация плагинов;
- `name.hpp` - имена адресов;
- `netnode.hpp` - низкоуровневое постоянное хранилище;
- `pro.h` - базовые typedef, макросы и `IDA_SDK_VERSION`;
- `search.hpp` - поиск по базе;
- `segment.hpp` - `segment_t` и секции;
- `struct.hpp` - `struc_t`, `member_t` и структуры;
- `typeinf.hpp` - TIL, прототипы и параметры;
- `ua.hpp` - `op_t`, `insn_t`, разбор инструкции и вывод её частей;
- `xref.hpp` - перекрёстные ссылки.

IDA_SDK_VERSION появился в 5.2 и позволяет условную компиляцию. Опубликованные функции отмечены ida_export; только они присутствуют в link libraries. idaapi обозначает соглашение stdcall в Windows, а не экспорт. Неотмеченную интересную функцию вызвать из модуля нельзя.

Netnode

netnode - самое общее постоянное хранилище внутри IDB. Высокоуровневые объекты скрывают его использование: например, сведения об импорте лежат в import_node, а persistent arrays IDC также основаны на netnode.

Внутренняя структура закрыта. C++-класс хранит только целочисленный netnodenumber. В 32-разрядной IDA это 32 бита, в 64-разрядной - 64. Часто номер совпадает с виртуальным адресом, естественно связывая произвольные данные, например комментарий, с адресом.

У каждого узла есть:

• имя до 512 символов;
• основное значение до 1024 байт;
• 256 разреженных массивов с 32-разрядными индексами;
• 256 массивов с 8-разрядными индексами;
• 256 hash tables со строковыми ключами.

Элемент хранит до 1024 байт. Данные находятся в B-tree файла ID0, который архивируется в IDB. Они не показываются в обычных окнах и подходят для состояния плагина между запусками.

Создание узла

Объявление C++-переменной ещё не создаёт запись. Она появляется после назначения имени, primary value или элемента массива.

```
netnode n0;
netnode n1(0x00401110);
netnode n2("$ node 2");
netnode n3("$ node 3", 0, true);
```

Только n3 гарантированно создан. n1 связан с номером и готов к записи. n2 лишь ищет существующее имя; при BADNODE его нужно создать:

```
if (BADNODE == (nodeidx_t)n2)
    n2.create("$ node 2");
```

n0.create() создаёт безымянный узел с автоматически назначенным номером. Без сохранённого номера или последующего имени найти его снова трудно. Автоматические номера начинались с 0xFF000000. Префикс "$ " рекомендован для пользовательских имён, чтобы не конфликтовать с внутренними.

Значения и массивы

Primary value можно трактовать как число (set_long, long_value), строку (set, valstr) или blob (set, valobj).

Для массива задаются индекс, его разрядность, данные, размер и tag массива. По соглашению:

• altval - целое, обычно tag 'A';

• supval - произвольные данные или строка, обычно 'S';
• hashval - строковой ключ, обычно 'H';
• charval - один байт без tag по умолчанию.

Тип определяется способом доступа, а не самим массивом. Автор обязан помнить формат конкретного tag.

Altval:

```
netnode n("$ idabook", 0, true);
n.altset(1000, 0x12345678);
ulong value = n.altval(1000);
n.altset(1000, value, (char)3);
```

Для 8-разрядного индекса используются altset_idx8 и altval_idx8, причём tag обязателен.

Supval принимает до 1024 байт:

```
char binary_data[] = {0xfe, 0xdc, 0x4e, 0xc7};
n.supset(1000, binary_data, sizeof(binary_data));
n.supset(1001, "example string");
n.supset(500, binary_data, sizeof(binary_data), (char)200);
```

Нулевой или отсутствующий размер означает строку, и сохраняется strlen + 1. При чтении сначала можно передать NULL, получить требуемый размер, выделить буфер, затем вызвать supval или supstr. supval возвращает число скопированных байтов, supstr - длину без завершающего нуля, хотя сам ноль копирует.

Altval фактически является удобной оболочкой над supval фиксированного размера. Hashval связывает число, строку или blob со строковым ключом через hashset, hashval, hashstr, hashval_long. Charval оборачивает однобайтовый supval.

Массивы обходятся семействами XXX1st, XXXnxt, XXXlast, XXXprev:

```
for (nodeidx_t i = n.alt1st(); i != BADNODE; i = n.altnxt(i))
    msg("altval[%d] = %d\n", i, n.altval(i));
```

Hash iteration возвращает ключи, по которым затем получают значения.

IDC CreateArray(name) создаёт netnode с префиксом $ idc_array и возвращает его номер. SetArrayLong пишет altval 'A', SetArrayString - supval 'S', а AR_LONG/AR_STR выбирают соответствующий tag.

n.kill() удаляет весь узел. altdel(100) удаляет один элемент, но altdel() без аргумента - весь массив 'A'. Из-за сходства имён легко потерять данные; для полного массива также есть явное altdel_all(tag). Аналоги существуют для остальных видов.

Основные типы SDK

• area_t - диапазон [startEA, endEA);
• func_t наследует area_t, хранит flags, локальные данные, аргументы и описывает как основную функцию, так и её tail chunks;
• segment_t наследует area_t, содержит имя, права R/W/X, класс и разрядность адреса;
• idc_value_t хранит строку, целое или float при взаимодействии с IDC;
• idainfo, доступный как глобальная inf, описывает процессор, формат filetype, entry point, minEA, maxEA, endianness mf и параметры ida.cfg;
• struc_t описывает structure, union или stack frame и содержит членов;
• member_t хранит начальное и конечное смещение члена;
• op_t описывает номер операнда, optype_t и зависимые от типа поля;

• insn_t содержит ea, внутренний itype, размер и до шести Operands типа op_t.

itype задаёт процессорный модуль и обычно не совпадает с двоичным opcode.

Часто используемые функции

Доступ к базе

В bytes.hpp: get_byte, get_word, get_long, get_many_bytes; изменения выполняют patch_byte, patch_word, patch_long, patch_many_bytes. get_original_byte/word/long возвращают самое первое значение до любых патчей. Промежуточное значение после первого из двух патчей не сохраняется. isLoaded проверяет адрес.

Пользовательский интерфейс

Низкоуровневый диспетчер callui получает ui_notification_t, но обычно удобнее оболочки kernwin.hpp: msg, warning, askstr, askfile_c, askyn_c, get_screen_ea, jumpto.

AskUsingForm_c строит собственный диалог из ASCII-описания. choose и choose2 создают одно- и многоколоночные списки, которых нет в IDC.

Имена, функции и структуры

get_name(from, addr, buffer, size) учитывает локальный контекст, set_name назначает имя с flags, get_name_ea ищет адрес.

get_func(addr) возвращает func_t*, get_func_qty - число функций, getn_func(n) - функцию по индексу, get_next_func - следующую, get_func_name - имя. get_frame возвращает struc_t* кадра.

Для структур: get_struc_id, get_struc, get_struc_size, get_member, get_member_by_name, add_struc, add_struc_member. Последняя получает имя, offset, flags FF_XXX, дополнительный typeinfo_t и размер.

Сегменты

getseg(addr) и get_segm_by_name находят объект. add_segm или add_segm_ex действительно создают секцию. get_segm_qty, getnseg, set_segm_name, get_segm_name, get_true_segm_name перечисляют и именуют её. Обычное получение имени может заменить запрещённые NameChars на SubstChar, а get_true_segm_name возвращает точную строку.

Одной инициализации segment_t, func_t или struc_t недостаточно: это оболочка представления. Постоянное изменение выполняется соответствующей API-функцией.

Перекрёстные ссылки

Код: get_first_cref_from, get_next_cref_from, get_first_cref_to, get_next_cref_to. Данные: get_first_dref_from, get_next_dref_from, get_first_dref_to, get_next_dref_to.

Эквивалента IDC XrefType нет. Для точного типа применяется xrefblk_t с полями from, to, iscode, type, user и методами first_from/next_from, first_to/next_to. Flags: XREF_ALL, XREF_FAR без обычного flow, XREF_DATA только для данных. Отдельного флага code-only нет, поэтому проверяют iscode или конкретный type.

Приёмы обхода

Функции можно перечислять тремя способами:

1. get_next_func от нулевого адреса.
2. Индексы от 0 до get_func_qty() с getn_func.
3. Низкоуровневый глобальный area control block funcs и методы get_next_area, getn_area.

Аналогичный глобальный segs содержит сегменты.

Стековый кадр обходится как struc_t. func_t::frsize даёт локальную область, frregs - сохранённые регистры, а адрес возврата находится после них. frame->memqty задаёт число членов, members[m].soff - смещение. Значение меньше frsize означает локальную переменную, больше offset возврата - параметр.

Для списка вызывающих функций простой get_first_cref_to не сообщает тип. Правильный вариант:

```
xrefblk_t xr;
for (bool ok = xr.first_to(func, XREF_ALL); ok; ok = xr.next_to()) {
    if (xr.type != fl_CN && xr.type != fl_CF)
        continue;
    char name[MAXNAMELEN];
    get_func_name(xr.from, name, sizeof(name));
    msg("call from %x in %s\n", xr.from, name);
}
```

Проверки одного iscode недостаточно: она также истинна для jump и ordinary flow.

Итоги

SDK открывает существенно больше базы и интерфейса, чем IDC, но требует уверенной работы с C++, заголовками и особенностями сборки. Главные навыки - находить API по функциональному заголовку, различать оболочку объекта и постоянное изменение базы, безопасно работать с netnode и выбирать корректный способ обхода функций, структур и ссылок.

Глава 17. Архитектура плагинов IDA

Плагин - компилируемый и более мощный аналог сценария. Обычно он запускается горячей клавишей или через Edit > Plugins после открытия базы, но может также реагировать на события IDA. Плагин бывает универсальным либо привязанным к формату или процессору и имеет полный доступ к SDK и системным библиотекам.

Создание плагина

Модуль является shared library своей платформы. Вместо обязательных экспортируемых функций он экспортирует объект `plugin_t` с именем `PLUGIN`. Класс определён в `loader.hpp`:

```
class plugin_t {
public:
    int version;
    int flags;
    int  (idaapi *init)(void);
    void (idaapi *term)(void);
    void (idaapi *run)(int arg);
    char *comment;
    char *help;
    char *wanted_name;
    char *wanted_hotkey;
};
```

Чтобы случайно не использовать устаревший API, до заголовков можно определить `NO_OBSOLETE_FUNCS`.

• `version` обычно равен `IDP_INTERFACE_VERSION`;
• `flags` - комбинация `PLUGIN_XXX`; для обычного плагина достаточно 0;
• `init` решает, следует ли загрузить модуль;
• `term` освобождает ресурсы перед выгрузкой и может быть `NULL`;
• `run` реализует основное действие и получает целый аргумент активации;
• `comment` и `help` в рассматриваемой версии напрямую IDA не использовала;
• `wanted_name` добавляется в Edit > Plugins;
• `wanted_hotkey` задаёт желаемую комбинацию. При конфликте её получает последний плагин.

Пример:

```
int idaapi idabook_init(void);
void idaapi idabook_term(void);
void idaapi idabook_run(int arg);

plugin_t PLUGIN = {
    IDP_INTERFACE_VERSION, 0,
    idabook_init, idabook_term, idabook_run,
    "Example plugin", NULL, "Idabook", "Alt-F9"
};
```

Указатели позволяют выбирать любые имена внутренних функций, не экспортируя их отдельно.

Жизненный цикл

IDA предлагает загрузить модуль в трёх точках:

1. При старте IDA, если установлен `PLUGIN_FIX`.
2. Сразу после процессорного модуля, если установлен `PLUGIN_PROC`.

3. При открытии базы, если предыдущих флагов нет.

`init` возвращает:

• `PLUGIN_SKIP` - не загружать;
• `PLUGIN_OK` - сделать доступным и загружать при активации;
• `PLUGIN_KEEP` - сделать доступным и оставить в памяти.

После загрузки пользователь вызывает `run`, либо callback получает событие. Перед выгрузкой вызывается `term`.

Обычный плагин выгружается при закрытии базы. `PLUGIN_UNL` выгружает его после каждого `run`; следующая активация снова вызывает `init`. `PLUGIN_PROC` живёт до выгрузки процессорного модуля, `PLUGIN_FIX` - до закрытия IDA.

Статическая инициализация задаёт поля `PLUGIN`, динамическая выполняется в `init`. Плагин `PLUGIN_FIX` может сделать одно действие при запуске и вернуть `PLUGIN_SKIP`. Плагин `PLUGIN_PROC` обычно проверяет глобальную `ph`:

```
int idaapi mips_init() {
    return ph.id == PLFM_MIPS ? PLUGIN_OK : PLUGIN_SKIP;
}
```

Обычный `init` может проверять формат, processor или compiler. Все выделенные ресурсы освобождаются в `term`. `PLUGIN_KEEP` полезен, если следующий запуск зависит от состояния предыдущего. Альтернатива - хранить состояние в netnode, сохраняя его и между сеансами. Для независимых вызовов `PLUGIN_OK` уменьшает расход памяти.

Уведомления о событиях

`hook_to_notification_point` регистрирует callback:

```
typedef int idaapi hook_cb_t(void *user_data,
                             int notification_code, va_list va);

hook_to_notification_point(HT_IDB, database_cb, NULL);
```

Категории:

• `HT_IDP` - processor notifications;
• `HT_UI` - события интерфейса;
• `HT_DBG` - debugger events;
• `HT_IDB` - изменения базы.

Например, `idb_event::byte_patched` означает патч байта, `cmt_changed` - изменение комментария. Дополнительные параметры зависят от кода и извлекаются `va_arg` в порядке, указанном в заголовке:

```
int idaapi database_cb(void *, int code, va_list va) {
    if (code == idb_event::byte_patched) {
        ea_t ea = va_arg(va, ea_t);
        msg("%x patched to %x, original %x\n",
            ea, get_byte(ea), get_original_byte(ea));
    }
    return 0;
}
```

В `term` обязательно вызывается:

```
unhook_from_notification_point(HT_IDB, database_cb, NULL);
```

Оставшийся callback после выгрузки почти наверняка приведёт к падению IDA.

Выполнение

`run` и callbacks работают в главном event loop. Пока плагин занят, автоанализ и интерфейс не обрабатываются. Долгая операция должна вызвать `show_wait_box`, периодически проверять `wasBreak`, а затем обязательно `hide_wait_box`. `wasBreak` заодно даёт IDA обновить интерфейс.

Нельзя просто вынести SDK-работу в новый поток: несинхронизированный доступ способен повредить базу. Для версий с безопасной очередью используются механизмы, описанные в главе 16.

Минимальный `run`:

```
void idaapi idabook_run(int arg) {
    msg("idabook plugin activated!\n");
}
```

Практический плагин может получить `func_t` через `get_func(get_screen_ea())`, затем `get_frame` и вывести локальные переменные, сохранённые регистры и параметры.

Сборка

Форматы модулей:

• Windows DLL с расширением `.plw` или `.p64`;
• Linux shared object `.plx` или `.plx64`;
• OS X `.pmc` или `.pmc64`.

Примеры SDK имели Borland makefiles. `idamake.pl` преобразовывал их для GNU make/gcc. Собственный makefile должен определить платформенные макросы (`__NT__`, `__IDP__`, `__LINUX__`, `__MAC__`), каталог include, подходящую `ida` library, shared linking и правильное расширение. Результат часто помещают в `<SDKDIR>/bin/plugins`, но это ещё не установка.

Visual Studio

В Visual Studio 2008 создают File > New > Project, тип Visual C++/Win32, шаблон Win32 Project.

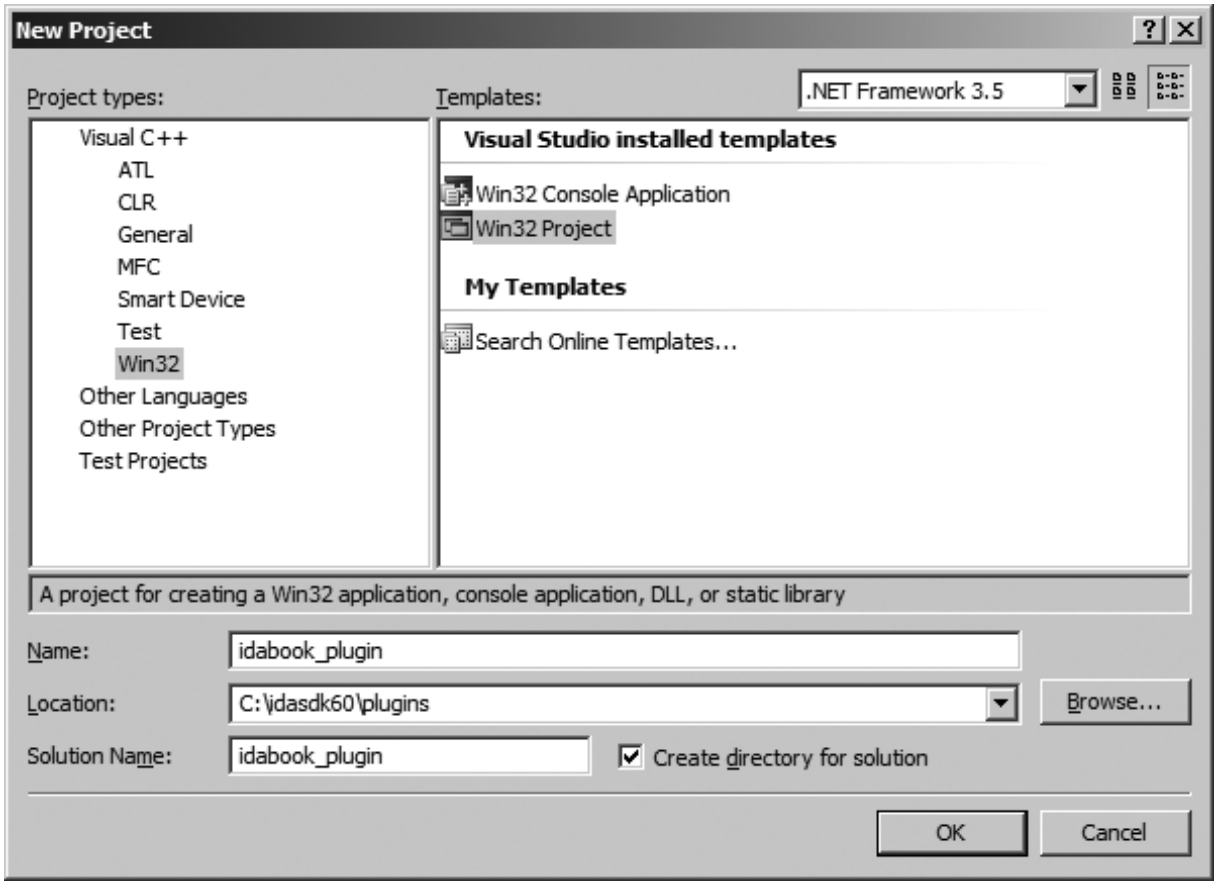

Рис. 17-1. Диалог создания проекта Visual Studio

В мастере выбирают DLL и Empty project.

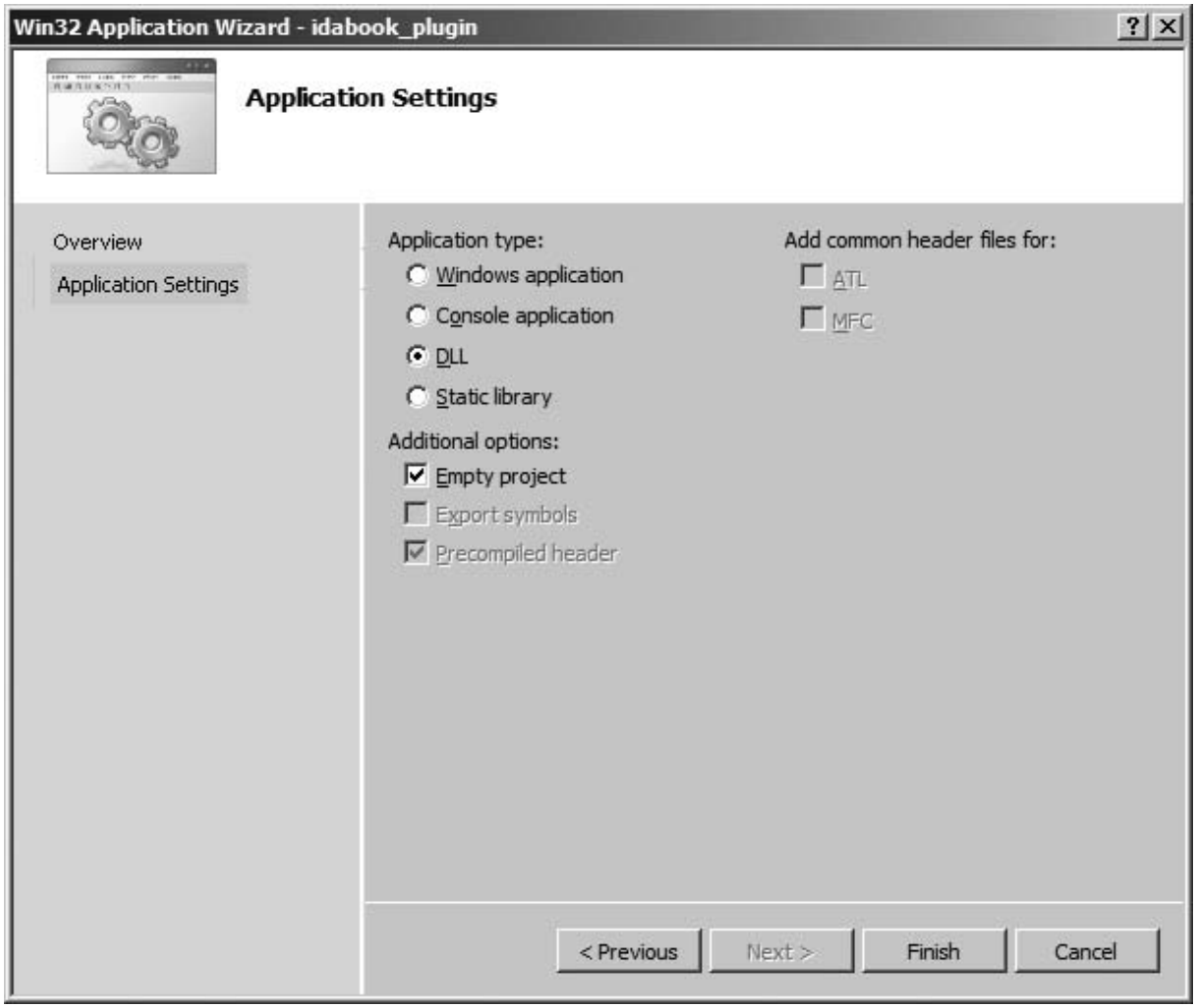

Рис. 17-2. Мастер Win32 Application

После добавления исходника становятся доступны C/C++ properties.

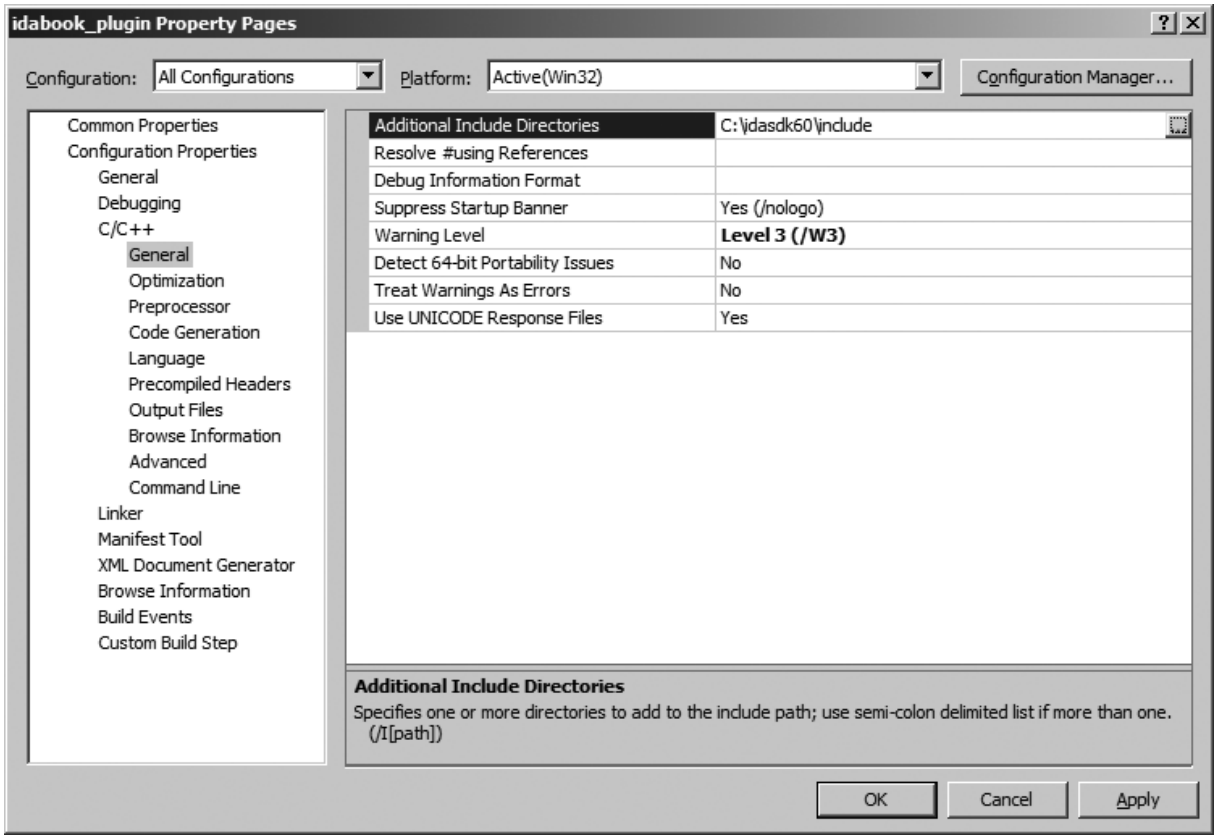

Рис. 17-3. Свойства проекта Visual Studio

Для всех конфигураций задаются:

• Output Directory, например `<SDKDIR>\bin\plugins`;
• Additional Include Directories: `<SDKDIR>\include`;
• Preprocessor Definitions: `__NT__;__IDP__`;
• Runtime Library: статическая Multithreaded или Multithreaded Debug, не DLL-вариант;
• Output File с расширением `.plw`;
• Additional Library Directories: `<SDKDIR>\lib\x86_win_vc_32`;
• Additional Dependencies: `ida.lib`;
• linker option `/EXPORT:PLUGIN`.

При раздельных Debug/Release параметры нужно изменить в обеих или выбрать All Configurations.

Установка и конфигурация

Готовый модуль копируется в `<IDADIR>/plugins`. Windows не позволяет заменить загруженный файл: обычному плагину может хватить закрытия базы, а `PLUGIN_FIX` требует закрыть IDA. В Linux и OS X файл можно перезаписать, но новая версия загрузится только в следующей подходящей точке жизненного цикла. Следует проверять документацию: плагину могут требоваться другие компоненты.

`<IDADIR>/plugins/plugins.cfg` переопределяет меню, путь/расширение, горячую клавишу и integer для `run`, а debugger plugins могут иметь `DEBUG`:

```
The_IdaBook_Plugin idabook_plugin Alt-F2 1
IdaBook_Plugin_Alt idabook_plugin Alt-F3 2
```

Подчёркивания в menu name превращаются в пробелы. Полный путь позволяет хранить модуль вне стандартного каталога. Отсутствующее расширение заменяется платформенным. Несколько строк могут дать одному модулю несколько пунктов и клавиш. Разные аргументы `run` позволяют выбрать разные режимы поведения; при `Alt+F3` выше передаётся 2.

Расширение IDC

`set_idc_func_ex` из `expr.hpp` добавляет IDC-функцию с C++-реализацией:

```
typedef error_t (idaapi *idc_func_t)(idc_value_t *argv,
                                     idc_value_t *res);
bool set_idc_func_ex(const char *name, idc_func_t impl,
                     const char *args, int flags);
```

Например, `CreateNetnode(string)` обходит автоматический префикс `$ idc_array`:

```
static error_t idaapi idc_create_netnode(idc_value_t *argv,
                                         idc_value_t *res) {
    res->vtype = VT_LONG;
    if (argv[0].vtype == VT_STR2) {
        netnode n(argv[0].c_str(), 0, true);
        res->num = (nodeidx_t)n;
    } else {
        res->num = -1;
    }
    return eOk;
}

int idaapi init() {
    static const char args[] = { VT_STR2, 0 };
    set_idc_func_ex("CreateNetnode", idc_create_netnode, args, 0);
    return PLUGIN_KEEP;
}
```

Плагин можно сделать `PLUGIN_FIX | PLUGIN_HIDE | PLUGIN_MOD`: он запускается при старте, не показывается в меню и модифицирует базу. Тогда `run` пуст, а IDC получает доступ к любому именованному netnode:

```
auto n = CreateNetnode("$ imports");
auto val = GetArrayElement(AR_STR, n, 0);
```

Прямого аналогичного расширения IDAPython из compiled plug-in в этой версии не было.

Интерфейс плагина

Chooser

`choose` из `kernwin.hpp` отображает одну колонку и получает `obj`, ширину, callback числа строк, callback форматирования и заголовок. Форматтер получает номера 1..n, а 0 означает header. Результат - индекс выбора или 0 при отмене.

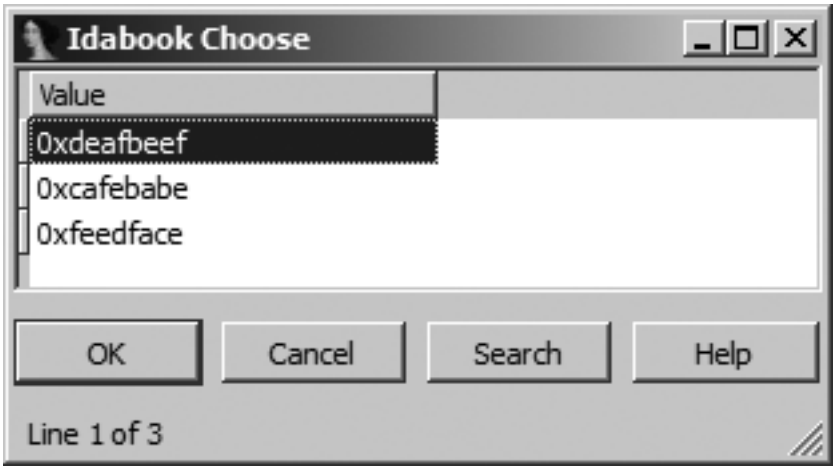

Рис. 17-4. Пример chooser dialog

`choose2` добавляет число колонок, массив их ширин и массив выходных cell buffers.

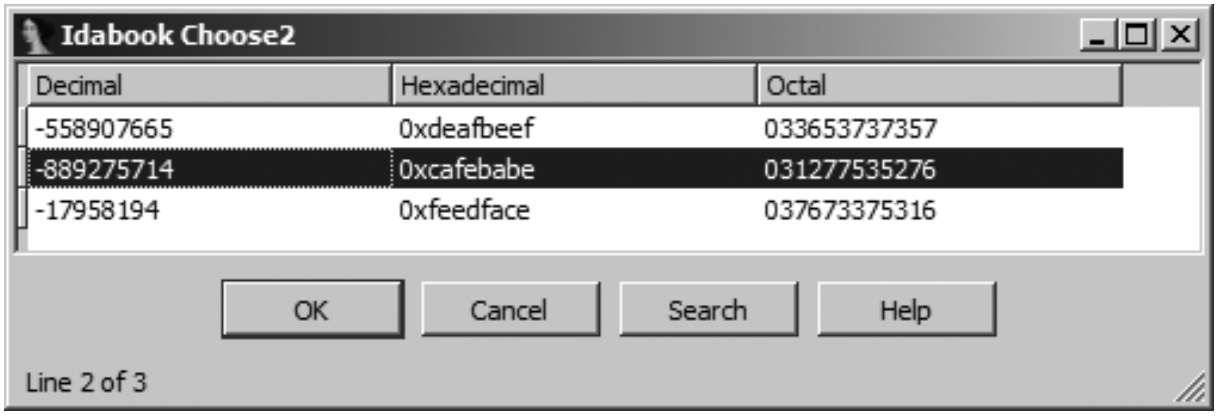

Рис. 17-5. Пример choose2 dialog

Оба поддерживают modal/modeless режим, множественный выбор и callbacks событий. Modeless chooser появляется как вкладка IDA; Imports реализовано через `choose2`.

AskUsingForm_c

`AskUsingForm_c(form, ...)` строит сложный диалог по format string. Поле имеет вид:

```
<#hint text#label:type:width:swidth:@hlp[]>
```

Type определяет input и тип указателя: `A` string, `D` decimal, `N` hex C, `M` hex без `0x`, `$` address, `I` identifier, `B` button, `K` color, `C` checkbox, `R` radio и другие. Числа трактуются как IDC expressions. Указатель одновременно даёт исходное и получает итоговое значение. Для button передаётся `formcb_t`.

Checkbox и radio группируются последовательными короткими specifiers, последний заканчивается дополнительным `>`; у первого можно задать заголовок рамки.

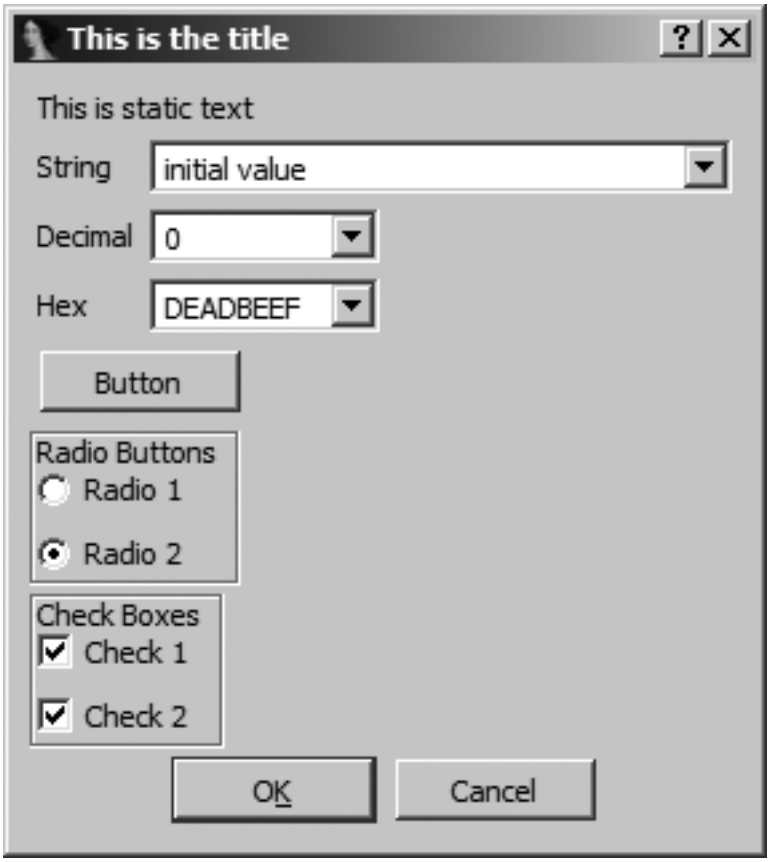

Рис. 17-6. Пример диалога AskUsingForm_c

Form может содержать `STARTITEM`, title с двумя переводами строки, статический текст и поля:

```
STARTITEM 0
This is the title

This is static text
<String:A:32:32::>
<Decimal:D:10:10::>
<#No leading 0x#Hex:M:8:10::>
<Button:B::::>
<##Radio Buttons##Radio 1:R>
<Radio 2:R>>
```

IDA сохраняет историю совместимых значений и показывает их в dropdown. Для radio первый элемент имеет номер 0; checkbox result является bitmask.

Native GUI и Qt

Windows-плагин мог получить HWND главного окна:

```
HWND mainWindow = (HWND)callui(ui_get_hwnd).vptr;
```

Затем можно использовать resources и Windows API, получая максимум гибкости ценой платформенной зависимости.

Qt-интерфейс IDA 6.0 позволил единый GUI на всех платформах. IDA 6.1 использовала Qt 4.7.2, собранный в namespace `QT`; исходники нужно конфигурировать с `-qtnamespace QT`, а qmake project - с `QT_NAMESPACE = QT`. Windows требовал Visual Studio и link libraries. В Linux и OS X можно было линковаться с Qt из поставки IDA.

При конфликте `qstrlen` IDA и Qt глобальная версия вызывается `::qstrlen`. Родительское окно:

```
QWidget *mainWindow = QApplication::activeWindow();
```

Пример `qwindow` показывал `create_tform`, visibility callbacks, получение `QWidget*` и заполнение формы Qt widgets.

Сценарные плагины

С IDA 5.7 плагины можно писать на Python или IDC без сложной сборки. Python обычно предпочтительнее благодаря SDK bindings. Требуется `PLUGIN_ENTRY`, возвращающая объект `plugin_t`:

```
from idaapi import *

class idabook_plugin_t(plugin_t):
    flags = 0
    wanted_name = "IdaBook Python Plugin"
    wanted_hotkey = "Alt-8"
    comment = "Example"
    help = "Help"

    def init(self):
        return PLUGIN_OK

    def term(self):
        pass

    def run(self, arg):
        warning("run(%d)" % arg)

def PLUGIN_ENTRY():
    return idabook_plugin_t()
```

IDC создаёт собственный класс с теми же полями и методами, поскольку готового base class нет. Файл `.py` или `.idc` устанавливается копированием в `<IDADIR>/plugins`.

Итоги

Плагин расширяет IDA через `plugin_t`, пользовательскую активацию и hooks. Надёжный модуль правильно выбирает момент загрузки, быстро возвращает управление, снимает callbacks и хранит состояние подходящим способом. Для быстрого прототипа достаточно Python plug-in; для системного API, нативного GUI или максимальной производительности применяется C++ SDK.

Глава 18. Бинарные файлы и загрузочные модули IDA

Если ни один loader не узнаёт файл, диалог показывает только Binary file. Пользователь должен хотя бы выбрать правильный processor, после чего вручную разложить образ по адресам, отделить code от data и дать автоанализу точки входа. Повторяемую процедуру лучше оформить собственным загрузочным модулем.

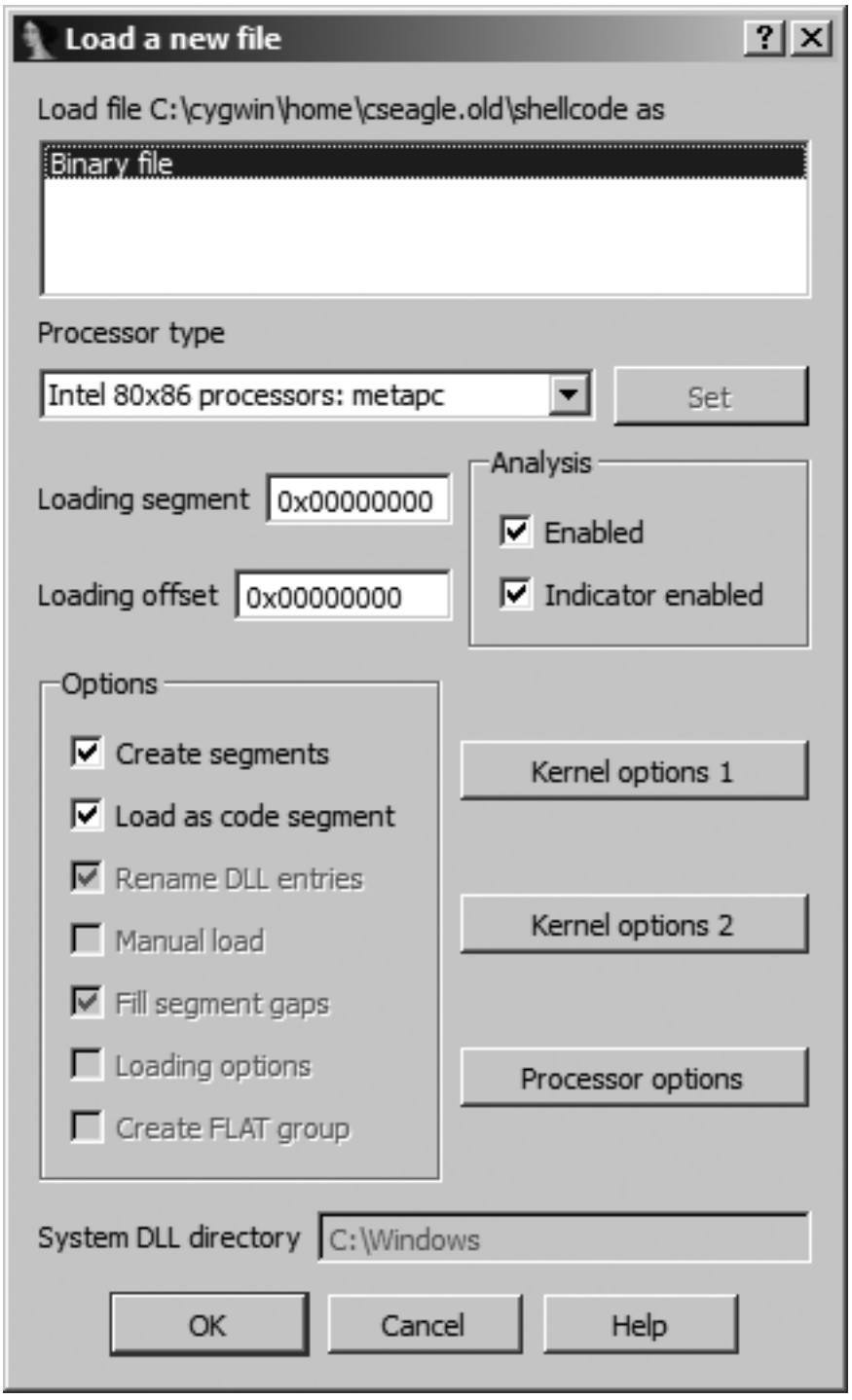

Рис. 18-1. Загрузка бинарного файла

Анализ неизвестного формата

Исполняемый код встречается в образах ОС, ROM embedded systems, firmware updates и даже network captures. Формат определяется ОС, архитектурой либо отсутствует, как у shellcode.

При Binary file IDA обычно складывает все байты в единственный `seg000` от адреса 0:

```
seg000:00000000 db 4Dh ; M
seg000:00000001 db 5Ah ; Z
seg000:00000002 db 90h
```

Некоторые embedded processor modules могут сделать предположения о ROM layout и начать разбор. В остальных случаях нужны спецификация CPU, формат ОС, документация hardware, memory maps и результаты debugger или logic analyzer.

Ручная загрузка Windows PE

Предположим, что PE loader отсутствует. Первые байты `4D 5A` подтверждают DOS header. По спецификации 32-разрядное поле по offset `0x3C`, `e_lfanew`, указывает на PE header. Удобнее не размечать поля по одному, а создать и применить `IMAGE_DOS_HEADER`.

Если `e_lfanew = 0x80`, по этому offset ожидается подпись `PE (50 45)` и структура `IMAGE_NT_HEADERS`. Из неё особенно важны:

- `Machine = 0x14C` - x86;
- `NumberOfSections = 4`;
- `AddressOfEntryPoint = 0x1000`;
- `ImageBase = 0x400000`;
- `SectionAlignment = 0x1000`;
- `FileAlignment = 0x200`.

Если Machine указывает, например, ARM `0x1C0`, базу нужно закрыть и загрузить заново с правильным processor: сменить архитектуру открытой базы нельзя.

ImageBase задаётся через Edit > Segments > Rebase Program.

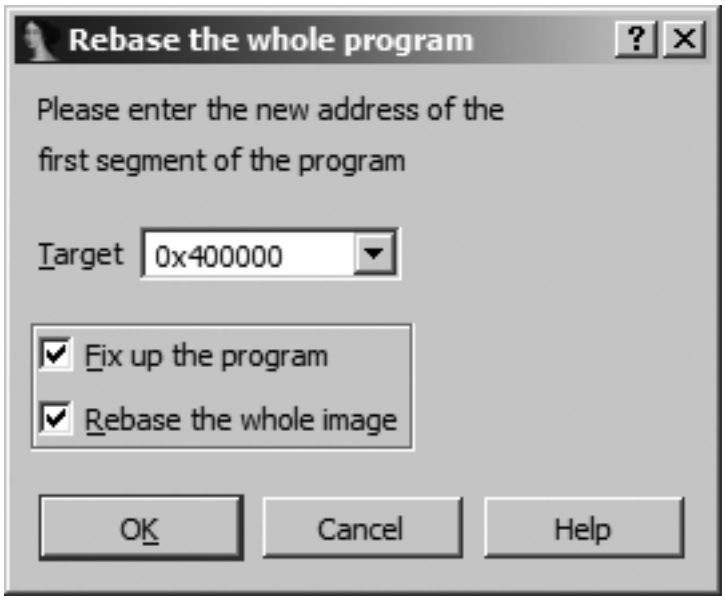

Рис. 18-2. Новый базовый адрес программы

Fix up the program применяет relocations, если они известны; Rebase the whole image двигает все сегменты. В Binary mode relocations неизвестны, а segment пока один.

Entry point RVA прибавляется к ImageBase: `0x400000 + 0x1000 = 0x401000`. Но сначала следует правильно отобразить секции.

После `IMAGE_NT_HEADERS` идёт массив `IMAGE_SECTION_HEADER`. Для `.text` пример содержит:

```
Name             .text
VirtualSize      0x440
VirtualAddress   0x1000
SizeOfRawData    0x600
PointerToRawData 0x400
```

Значит первые `0x400` байтов логически являются headers. Их можно выделить в `.headers` через Edit > Segments > Create Segment.

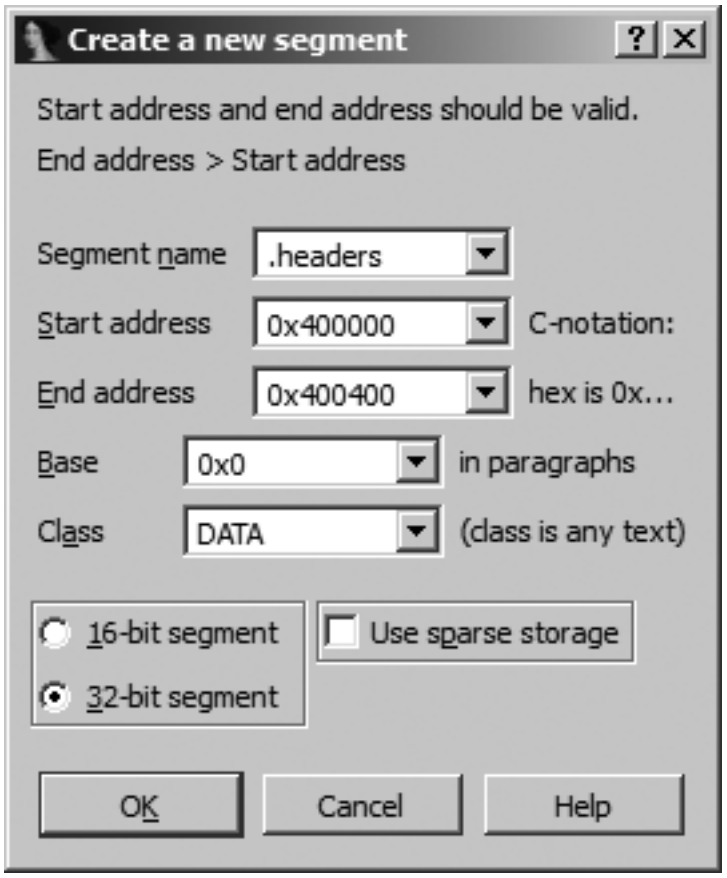

Рис. 18-3. Диалог создания сегмента

Start включителен, End исключителен. При плоской адресации Base равен 0. Для segmented x86 виртуальный адрес вычисляется как `(base << 4) + offset`. Class может быть `CODE`, `DATA`, `BSS` и другим.

Создание segment снимает определения данных внутри диапазона, поэтому структуры заголовков придётся применить снова.

`.text` берёт `0x600` байтов с file offsets `0x400-0x9FF` и отображает их в `0x401000-0x4015FF`. После отделения `.headers` IDA оставляет остаток в `seg001`. Его двигают на `0x401000`.

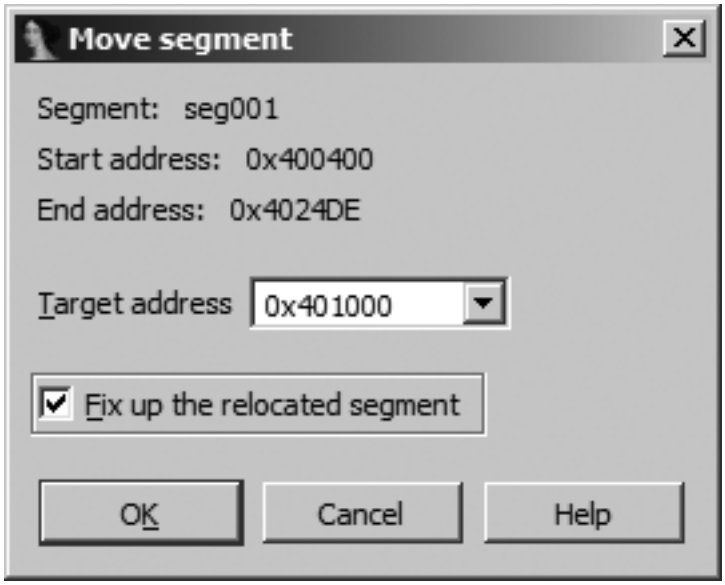

Рис. 18-4. Перемещение сегмента

Затем первые `0x600` байтов нового положения превращают в `.text` с диапазоном `[0x401000, 0x401600)`.

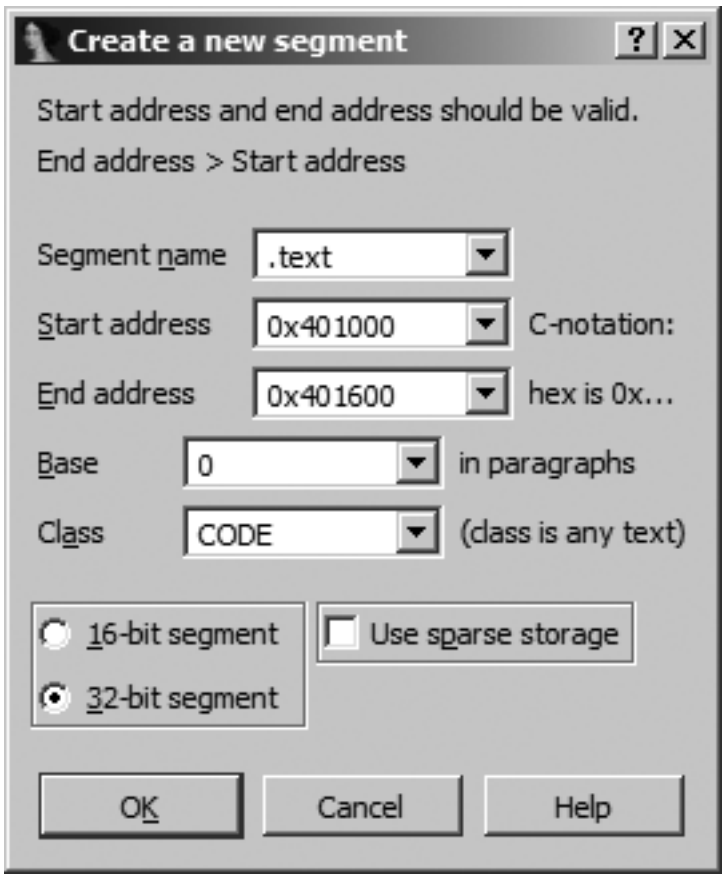

Рис. 18-5. Ручное создание секции .text

Остаток снова получает временное имя. Следующая `.rdata` с `PointerToRawData=0xA00`, `SizeOfRawData=0x200`, `VirtualAddress=0x2000` перемещается на `0x402000` и выделяется как `[0x402000,0x402200)`. Создавать секции в обратном порядке бывает удобнее: тогда ещё не перемещённый остаток сохраняет соответствие file offsets.

`.bss` отличается: `SizeOfRawData=0`, но `VirtualSize=0x40`. В файле байтов нет, а loader ОС выделит и обнулит 64 байта по RVA `0x3000`. В IDA нужно создать gap и BSS segment `[0x403000,0x403040)`.

Итоговый layout может включать `.headers`, `.text`, `.rdata`, `.bss`, `.idata`. Если end addresses рассчитаны только по raw size, между ними останутся gaps до aligned virtual starts. Для точного образа ends можно расширить с учётом VirtualSize и SectionAlignment.

Permissions берутся из `Characteristics`, но GUI ручного создания их не задаёт. IDC использует Unix-подобные bits read=4, write=2, execute=1:

```
SetSegmentAttr(0x401000, SEGATTR_PERM, 1);
```

После раскладки нужно создать code и точку входа. Python one-liner:

```
AddEntryPoint(0x401000, 0x401000, 'start', 1)
```

IDA добавит `start` в Exports, создаст instruction и начнёт recursive descent. Но Binary mode не определяет compiler, imports, libraries, types и signatures. До качества штатного PE loader остаётся много ручной работы. Именно эту работу автоматизирует loader module.

Загрузочные модули IDA

Loader используется только при создании новой базы. Он распознаёт формат, читает файл, создаёт sections, maps bytes и передаёт работу processor module. Позднее исходный loader может обрабатывать move segment и Create EXE File.

IDA динамически открывает каждый модуль из `<IDADIR>/loaders`, предлагает ему проверить вход и показывает все совпавшие форматы пользователю. Loader можно писать на SDK или, начиная с IDA 5.6, как script.

Loader на SDK

Каждый модуль экспортирует `loader_t LDSC` из `loader.hpp`:

```
struct loader_t {
    ulong version;
    ulong flags;
    int  (idaapi *accept_file)(linput_t *li,
            char fileformatname[MAX_FILE_FORMAT_NAME], int n);
    void (idaapi *load_file)(linput_t *li, ushort neflags,
            const char *fileformatname);
    int  (idaapi *save_file)(FILE *fp, const char *fileformatname);
    int  (idaapi *move_segm)(ea_t from, ea_t to, asize_t size,
            const char *fileformatname);
    bool (idaapi *init_loader_options)(linput_t *li);
};
```

`linput_t` - независимая от compiler оболочка input stream; операции определены в `diskio.hpp`.

• `version` равен `IDP_INTERFACE_VERSION`;
• `flags` обычно 0, специальный `LDRF_RELOAD` описан в заголовке;
• `accept_file` читает минимум для распознавания, записывает display name и возвращает 0 или nonzero; `ACCEPT_FIRST` поднимает вариант в списке;
• `load_file` получает выбранный format и `NEF_XXX` flags из dialog, maps data или вызывает `loader_failure`;

• save_file необязателен и создаёт output по базе;
• move_segm необязателен и может пересчитать relocations;
• init_loader_options конфигурирует wizard File > New только в Windows native GUI. При File > Open он не вызывается.

XML wizard templates находились в <IDADIR>/cfg, но отдельно почти не документировались.

Простой формат Simpleton

Вымышленный little-endian формат:

```
struct simpleton {
    uint32_t magic; // 0x1DAB00C
    uint32_t size;
    uint32_t base;  // base и entry point
    uint8_t code[size];
};
```

Loader включает ../idaldr.h, объявляет SIMPLETON_MAGIC и LDSC с accept, load, save; callbacks move/options равны NULL.

accept_file вызывается с растущим n, позволяя одному loader распознать несколько родственных форматов. Для одного формата при n != 0 возвращается 0. lread4bytes(li, &magic, false) читает little-endian magic; true читало бы big-endian. При совпадении записывается Simpleton Executable и возвращается 1.

Основной load_file:

```
void idaapi load_simpleton_file(linput_t *li, ushort, const char *) {
    simpleton hdr;
    lread(li, &hdr, sizeof(hdr));
    file2base(li, sizeof(hdr), hdr.base, hdr.base + hdr.size,
              FILEREG_PATCHABLE);
    if (!add_segm(0, hdr.base, hdr.base + hdr.size,
                  NAME_CODE, CLASS_CODE))
        loader_failure();
    segment_t *s = getseg(hdr.base);
    set_segm_addressing(s, 1);
    create_filename_cmt();
    add_entry(hdr.base, hdr.base, "_start", true);
}
```

Шаги: прочитать header, скопировать code, создать segment, включить 32-bit addressing, добавить стандартный file comment с input name/MD5/license и entry point.

file2base(li, pos, ea1, ea2, patchable) читает ea2-ea1 байтов. FILEREG_PATCHABLE сохраняет связь database address с file offset, необходимую DIF.

add_entry(ord, ea, name, makecode) добавляет export/entry. Если ordinal отсутствует, ord=ea. Для exported data makecode=false.

save_file сначала вызывается с fp=NULL как запрос возможности. Возврат 1 разрешает output dialog. При настоящем FILE он пишет magic, size, base и вызывает base2file для segment.

В Windows FILE* ядра Borland нельзя передавать стандартной C library. Нужно применять qfwrite, qfprintf и другие qfxxx. Если определён USE_STANDARD_FILE_FUNCTIONS, всё равно нельзя смешивать IDA FILE с C functions или C FILE с qfxxx.

Loader extensions: Windows .ldw/.l64, Linux .llx/.llx64, OS X .lmc/.lmc64. Сборка похожа на plugin, результат можно хранить в <SDKDIR>/bin/loaders, установка - копирование в <IDADIR>/loaders.

Пример loader для pcap

Даже network capture может содержать shellcode. Pcap состоит из общего header и последовательности packet header + content. Учебный loader разделяет `.file_header` и `.packets`, а затем применяет structures pcap, Ethernet, IP, TCP и UDP.

Первый способ получить types: `add_til2("gnuunx.til", ADDTIL_SILENT)`, затем `til2idb(-1, name)` и сохранить `tid_t`. Недостатки - импорт большой библиотеки ради шести типов и возможная ошибка готового определения.

Второй способ строит временную TIL из C declarations:

```
til_t *t = new_til("pcap.til", "pcap header types");
parse_decls(t, pcap_types, NULL, HTI_PAK1);
sort_til(t);
pcap_hdr_struct = import_type(t, -1, "pcap_file_header");
// остальные типы
free_til(t);
```

`HTI_PAK1` задаёт packing 1. После изменения TIL обязательно сортируется. При необходимости `compact_til` и `store_til` сохраняют её, но для нескольких типов проще строить при загрузке.

Loader переносит file header через `file2base`, создаёт `.file_header` и применяет `doStruct`. Затем циклически читает `pcap_pkthdr`; уже прочитанный в memory header копирует `mem2base`, content - `file2base`. После чтения создаётся один `.packets` segment. Structures нужно применять после создания segment, иначе Create Segment их снимет.

При разметке каждого packet loader:

1. Применяет `pcap_pkthdr`.
2. Следом `ether_header`.
3. Проверяет EtherType с byte swap.
4. Для IP применяет `iphdr`, читает protocol и вычисляет IHL как `(first_byte & 0xF) * 4`.
5. Для protocol TCP/UDP применяет соответствующий header.
6. Переходит к следующему packet по captured length.

Так HTTP data становится видна сразу после разворачиваемых header structures. На этой основе plugin может делать TCP reassembly, extraction и более читаемое форматирование IP/byte order.

Другие стратегии

Loader и processor могут быть loosely coupled: loader создаёт layout и entries, processor лишь дизассемблирует. Java использовал tight coupling: loader устанавливал processor `java`, затем отправлял `ph.loader` notification, и процессор сам выполнял всю загрузку, сохраняя полезное внутреннее состояние.

Другой вариант - loader пишет state в netnodes, а processor читает. Это проще разделяет модули и сохраняет данные, но частые обращения к IDB медленнее C++ structures.

Сценарный loader

Script должен определить как минимум `accept_file(li,n)` и `load_file(li,neflags,format)`. IDC Simpleton читает magic/size/base через `loader_input_t`, вызывает `loadfile`, `AddSeg`, `AddEntryPoint` и возвращает 1.

Python позволяет использовать `struct` и больше SDK:

```
def accept_file(li, n):
    if n:
        return 0
    li.seek(0)
    magic = struct.unpack("<I", li.read(4))[0]
    return "Python Simpleton Loader" if magic == 0x1DAB00C else 0

def load_file(li, neflags, format):
    li.seek(0)
    magic, size, base = struct.unpack("<III", li.read(12))
    li.file2base(12, base, base + size, 1)
    add_segm(0, base, base + size, ".text", "CODE")
    add_entry(base, base, "_start", 1)
    return 1
```

Файл копируется в `<IDADIR>/loaders`. Scripted loader удобен для быстрого prototype, который позднее можно перенести на SDK.

Итоги

Loader отвечает за формат файла, mapping, segments, types и entry points, но не должен дизассемблировать инструкции: это задача processor module. Основные заголовки разработки - `loader.hpp`, `segment.hpp`, `entry.hpp`, `diskio.hpp`. Ручное исследование неизвестного формата даёт точный план будущего loader и превращает повторяемую раскладку в автоматическую загрузку.

Глава 19. Процессорные модули IDA

Processor module - самый сложный вид расширения. Он превращает opcodes в instructions, создаёт functions и cross-references, отслеживает stack pointer и формирует весь текст листинга. Собственный модуль нужен для неизвестного CPU или bytecode custom virtual machine. Начиная с IDA 5.7 его можно писать также на Python/IDC.

В качестве учебного примера используется Python bytecode. Без отдельного `.pyc` loader файл приходится открыть как Binary, выбрать Python processor, вручную найти начало code и применить Edit > Code.

Python bytecode

Python source при выполнении компилируется в bytecode stack virtual machine. Импортируемые modules кешируются в `.pyc`, примерно как Java `.class`. Такой файл может распространяться без source и становиться объектом reverse engineering.

У VM есть главным образом instruction pointer и stack pointer. Например, `BINARY_ADD` снимает два значения, складывает и кладёт один результат. Каждая Python instruction содержит однобайтовый opcode и либо 0, либо 2 байта operand.

Processor module на SDK

Документация ограничивалась `idp.hpp`, README и примерами `<SDKDIR>/module`. Простым ориентиром считался processor z8. Практический путь - скопировать модуль с похожей логической моделью и изменить. Имена `ana.cpp`, `emu.cpp`, `out.cpp` являются лишь традицией build scripts.

Processor состоит из трёх логических стадий:

1. Analyzer декодирует структуру одной instruction.
2. Emulator моделирует её ссылки, flow и stack effects.
3. Outputter создаёт текст instruction и operands.

Каждый модуль экспортирует `processor_t LPH`, автоматически через `module/idaidp.hpp`. `processor_t` имел 56 полей, включая 26 function pointers, а assemblers `asm_t` - ещё 59 полей. Поэтому точное копирование готового skeleton снижает риск смещения initializer.

Базовая инициализация LPH

Все mnemonics описывает массив `instruc_t`:

```
struct instruc_t {
    const char *name;
    ulong feature;
};
```

`feature` - комбинация canonical flags:

• `CF_STOP` - нет последовательного продолжения;
• `CF_CHGn` - instruction изменяет operand n;
• `CF_USEn` - читает или ссылается через operand n;

• CF_CALL - function call.

Порядок не обязан совпадать с binary opcodes. В Python удобно совпадение один к одному, поэтому gaps заполняются {NULL,0}. Параллельный enum сопоставляет integer с instruction, а завершающий PYTHON_LAST обозначает границу.

Поля LPH instruc_start, instruc_end, instruc получают первый enum, значение after-last и массив.

Registers задаются массивом names и синхронным enum. Для x86 это eax, ebx и другие; LPH получает regsNum=qnumber(RegNames) и regNames.

IDA требует segment registers даже у архитектуры без них. Поэтому Python объявляет фиктивные:

```
static char *RegNames[] = { "cs", "ds" };
enum py_registers { rVcs, rVds };
```

Поля regFirstSreg, regLastSreg, segreg_size, regCodeSreg, regDataSreg получают rVcs, rVds, 0, rVcs, rVds. Модуль затем их игнорирует.

Начальные поля processor_t:

• version = IDP_INTERFACE_VERSION;
• custom id > 0x8000;
• flag - PR_XXX, в примере PR_RNAMESOK и decimal default PRN_DEC;
• cnbits=8, dnbits=8.

Analyzer

u_ana указывает на функцию:

```
int idaapi ana(void);
```

Kernel заранее записывает в глобальную insn_t cmd сегментный и linear address. Analyzer не меняет базу и должен заполнить:

• cmd.itype - enum instruction, не opcode как таковой;
• cmd.size - длину;
• до шести cmd.Operands типа op_t.

При ошибке возвращается 0, иначе length. Bytes берутся ua_next_byte, ua_next_word, ua_next_long, ua_next_qword; каждая функция автоматически увеличивает cmd.size.

op_t содержит номер n, type, offsets offb/offo, flags, datatype dtyp, а unions хранят register/phrase, immediate value, address addr и processor-specific specval/specflag.

Основные operand types:

• o_reg - register;
• o_mem - известный memory address;
• o_imm - immediate;
• o_displ - register + displacement;
• o_near/o_far - code target;
• o_void - operand отсутствует.

Размер задают dt_byte, dt_word, dt_dword и другие. Analyzer не решает source это или destination: эту семантику дают CF_USEn и CF_CHGn.

Processor-specific fields удобно использовать для передачи уже вычисленной информации emulator/outputter.

У Python opcode меньше 90 не имеет operand, остальные получают little-endian word. Минимальная логика:

```
int idaapi py_ana(void) {
    cmd.itype = ua_next_byte();
    if (cmd.itype >= PYTHON_LAST ||
        Instructions[cmd.itype].name == NULL)
        return 0;

    if (cmd.itype < 90) {
        cmd.Op1.type = o_void;
        cmd.Op1.dtyp = dt_void;
    } else {
        cmd.Op1.type = flags[cmd.itype] & (HAS_JREL|HAS_JABS)
                     ? o_near : o_mem;
        cmd.Op1.offb = 1;
        cmd.Op1.dtyp = dt_dword;
        cmd.Op1.value = ua_next_word();
        cmd.auxpref = flags[cmd.itype];

        if (cmd.auxpref & HAS_JREL)
            cmd.Op1.addr = cmd.ea + cmd.size + cmd.Op1.value;
        else if (cmd.auxpref & HAS_JABS)
            cmd.Op1.addr = cmd.Op1.value;
        else if (cmd.auxpref & HAS_CALL) {
            cmd.Op1.specflag1 = cmd.Op1.value & 0xFF;
            cmd.Op1.specflag2 = cmd.Op1.value >> 8;
        }
    }
    return cmd.size;
}
```

Supplementary flags distinguish relative/absolute jumps и call argument counts. Общий analyzer: прочитать достаточно bytes для valid itype, разобрать count/addressing/components operands и вернуть total length. Fixed-size RISC обычно проще variable-length x86.

Emulator

`u_emu` указывает на:

```
int idaapi emu(void);
```

После analyzer `cmd` заполнен. Emulator может менять базу. Он создаёт code/data xrefs, sequential flow, stack points и stack variables.

Типовая схема смотрит canonical features и вызывает helper для каждого read/write operand. Если нет `CF_STOP`, добавляется flow:

```
ua_add_cref(0, cmd.ea + cmd.size, fl_F);
```

Рекомендуемые правила:

• `o_imm`, read и pointer: `ua_add_off_drefs(op, dr_O)`;
• `o_displ` pointer: `ua_add_off_drefs(op, dr_R или dr_W)`;
• `o_mem`: `ua_add_dref(op.offb, op.addr, dr_R или dr_W)`;
• `o_near`: `ua_add_cref(op.offb, op.addr, fl_CN или fl_JN)`.

Следует использовать xref functions из `ua.hpp`, а не generic `xref.hpp`.

Stack change сообщается:

```
add_auto_stkpnt2(func_t *pfn, ea_t end, sval_t delta);
```

Отрицательный delta означает рост stack, положительный - shrink. `push` 4 bytes даёт `-4`; `sub esp, 0x48` даёт `-0x48`. Нужно распознавать все способы изменения SP, не только push/pop.

Python emulator может вычислить только relative branches и fallthrough:

```
int idaapi py_emu(void) {
    if (cmd.auxpref & HAS_JREL)
        ua_add_cref(cmd.Op1.offb, cmd.Op1.addr, fl_JN);
    if (!(cmd.get_canon_feature() & CF_STOP))
        ua_add_cref(0, cmd.ea + cmd.size, fl_F);
    return 1;
}
```

Data operands в `.pyc` часто являются indexes tables, а absolute locations требуют metadata file format. Без loader processor их не знает. Stack tracking также опущен из-за неизвестных function boundaries; полноценная реализация могла бы иметь table stack deltas для каждого opcode.

Outputter

`u_out` указывает на `void out()`, `u_outop` - formatter одного operand, `d_out` - formatter data. Outputter не меняет базу.

IDA использует output buffer:

1. `init_output_buffer(buf,size)`.
2. Добавить mnemonic/operands функциями `OutMnem`, `out_one_operand`, `OutValue`, `out_symbol`, `out_line`, `out_register`.
3. `term_output_buffer()`.
4. `MakeLine` или `printf_line`.

`out` вызывает `out_one_operand`, а не `outop` напрямую. Register name можно вывести как `out_register(ph.regNames[regnum])`.

Для multiline instruction каждый display line требует отдельного `MakeLine`, без `\n`. `indent=-1` применяет `inf.indent` и отмечает основную строку. Глобальную `gl_comm` надо поставить в 1, иначе даже существующий comment не появится.

Пример Python:

```
void py_out(void) {
    char str[MAXSTR];
    init_output_buffer(str, sizeof(str));
    OutMnem(12);
    if (cmd.Op1.type != o_void)
        out_one_operand(0);
    term_output_buffer();
    gl_comm = 1;
    MakeLine(str);
}
```

`outop` обычно switches по `op.type`. В примере `COMPARE_OP` преобразует индекс в `<`, `<=`, `==`, `!=`, `>`, `>=`, `in`, `is` и другие. Relative jump использует `out_name_expr` для label, прочее - `OutValue`.

`d_out(ea)` вызывается для unknown/data bytes. Он читает flags из `bytes.hpp`, выбирает word/dword/byte и синтаксис текущего assembler `ash.a_word`, `ash.a_dword`, `ash.a_byte`. Простейший Python formatter также показывает printable ASCII после byte. Arrays и complex data требуют дополнительной логики.

Processor notifications

LPH `notify` получает `idp_notify` и variable arguments. События сообщают init/term, создание code/data/functions/segments и другое. Хороший processor сначала вызывает:

```
int result = invoke_callbacks(HT_IDP, msgid, va);
```

Если callback не обработал событие (`result==0`), module делает это сам. Учебный notify при `init` ставит `inf.mf=0` для little-endian. При `make_data` извлекает `ea`, flags, tid, len и отклоняет `len>4`, поскольку `d_out` умеет лишь byte/word/dword.

Остальные поля processor_t

Некоторые callbacks допускают `NULL`: `is_far_jump`, `translate`, `is_switch`, `is_sp_based`, `create_func_frame`, `get_frame_retsize`, `u_outspec`, `set_idp_options`. Для обязательных, но ненужных header/footer/segstart/segend можно использовать empty stub. `realcvt` часто указывает на `ieee_realcvt`.

Имена processor:

- `shnames` - NULL-terminated short names до 8 символов;
- `lnames` - столько же full names;
- `asms` - NULL-terminated pointers на `asm_t`.

Пользователь выбирает processor и assembler на Analysis tab.

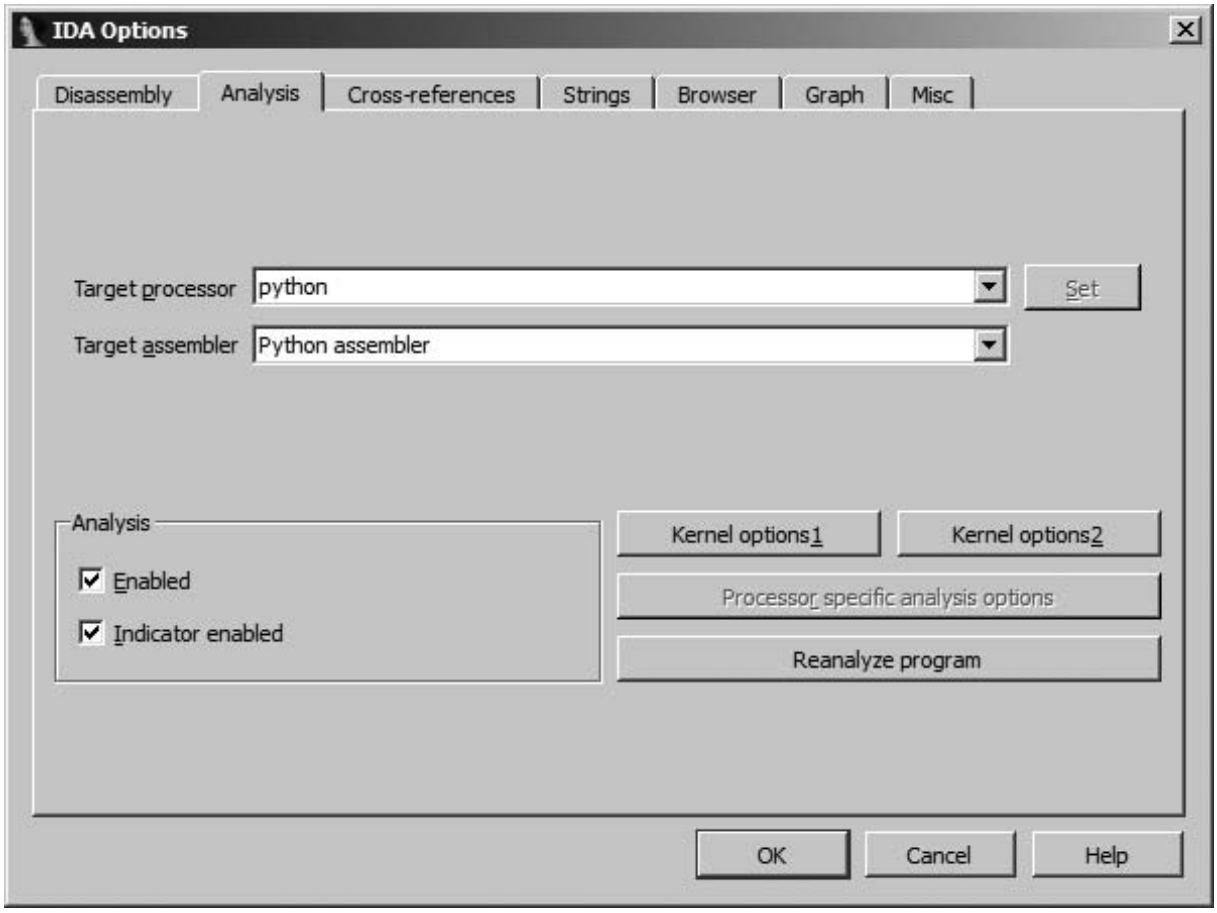

Рис. 19-1. Выбор альтернативного процессора и ассемблера

Multi-processor module обрабатывает `newprc`, multi-syntax - `newasm`. `asm_t` описывает number formats, quote delimiters и data directives. Готовый Python processor способен выводить `LOAD_CONST`, symbolic comparison, labels для relative jumps и xrefs, но tables constants/names требуют loader knowledge.

Сборка processor module

Extensions: Windows `.w32`/`.w64`, Linux `.ilx`/`.ilx64`, OS X `.imc`/`.imc64`. Результаты примеров идут в `<SDKDIR>/bin/procs`, установка - `<IDADIR>/procs`.

Windows требует processor description по offset 128, иначе module не появится в load dialog. Исторически использовался custom DOS stub `<SDKDIR>/module/stub` и postprocessor:

```
mkidp procs/python.w32 "Python Bytecode:python"
```

`mkidp` записывает `Long name:short name`; при нехватке места сообщает `too long processor description`. Borland мог задать stub через `idp.def`, Microsoft обычно оставлял место, GNU - нет.

Утилита книги `fix_proc` заменяла DOS stub, переносила PE headers, создавала пространство и записывала description, поэтому работала с разными linkers. Строго важен не сам stub, а строка по offset 128.

Makefile отличается от plugin: задаёт `DESCRIPTION`, `PROC_EXT`, `OUTDIR=bin/procs`, objects `ana.o emu.o ins.o out.o reg.o` и после Windows link вызывает `fix_proc binary "description"`.

Настройка существующего processor

Если модуль почти подходит, plugin может hook processor notifications:

- `custom_ana` добавляет instruction с `itype >= 0x8000`;
- `custom_emu` моделирует custom instruction;
- `custom_out` меняет output;
- `custom_outop` меняет operand;
- `custom_mnem` возвращает mnemonic.

При необходимости вызываются оригинальные `(*ph.u_emu)()`, `(*ph.u_out)()`, `(*ph.u_outop)(op)`.

Пример x86 plugin проверяет `ph.id==PLFM_386`, hooks `HT_IDP`, в `custom_out` заменяет `leave` на `cya`, а для двух operands временно меняет `cmd.Op1`/`cmd.Op2`, вызывает original output и возвращает их. Flags: `PLUGIN_PROC | PLUGIN_HIDE | PLUGIN_MOD`. Nonzero callback result сообщает, что output обработан.

Архитектура loader/processor

Real CPU processor обычно независим от file format: x86 работает с PE, ELF, Mach-O. Хороший loader также поддерживает несколько processors.

У VM bytecode и container тесно связаны. Python processor без `.pyc` metadata не знает tables, function boundaries и absolute targets. Решения:

- loader создаёт специальный layout и передаёт state через netnodes;

• loader только узнаёт `.pyc`, устанавливает Python processor и посылает `processor_t::loader`, после чего processor сам загружает файл.

Tight coupling аналогичен Java modules SDK:

```
if (ph.id != PLFM_PYTHON)
    set_processor_type("python", SETPROC_ALL | SETPROC_FATAL);
if (ph.notify(processor_t::loader, li, neflag))
    error("Python processor/loader failed");
```

Так analyzer, emulator и outputter имеют всё состояние file format.

Scripted processor

IDA 5.7 устранила compile phase. Лучшие starting points - `<SDKDIR>/module/script/proctemplate.py` и shipped Python processors, например `<IDADIR>/procs/ebc.py`.

Нужно:

1. Унаследоваться от `idaapi.processor_t`, заполнить все обязательные data fields, assembler dictionary, instruction list и methods `ana`, `emu`, `out`, `outop`.
2. Определить глобальную `PROCESSOR_ENTRY`, возвращающую instance.

```
class demo_processor_t(idaapi.processor_t):
    # id, names, registers, assembler, instruc, callbacks...
    def ana(self): pass
    def emu(self): pass
    def out(self): pass
    def outop(self): pass

def PROCESSOR_ENTRY():
    return demo_processor_t()
```

Script копируется в `<IDADIR>/procs`.

Итоги

Processor module требует терпения и множества проверок, но однажды созданный decoder повторно используется со всеми новыми binaries. Ключ к надёжности - строгая граница стадий: analyzer только декодирует `cmd`, emulator меняет базу и создаёт semantics, outputter только форматирует. Для file-dependent VM заранее проектируется связь с loader и persistent state.

Глава 20. Особенности компиляторов

Один и тот же source превращается в разный assembly в зависимости от compiler, platform, version и optimization flags. Знакомство только с gcc/Linux не гарантирует узнавания Visual C++ debug, Delphi или Visual Basic. Эти различия помогают не только понимать листинг, но и профилировать среду и автора программы.

Jump tables и switch

Compiler оптимизирует `switch` по распределению case values. Для редких значений вроде 1, 211, 295, 462 обычно строится binary search с O(log N). Для плотного диапазона 1..12 выгоднее table lookup O(1).

Borland

Для consecutive cases Borland мог создать 13-entry table, включая dummy case 0:

```
mov edx, [ebp+arg_0]
cmp edx, 0Ch
ja  loc_default
jmp ds:off_401168[edx*4]

off_401168 dd offset loc_default ; case 0
           dd offset loc_case_1
           dd offset loc_case_2
```

IDA обычно распознаёт comparison, table, default и каждый case. `ja` выполняет unsigned comparison, поэтому negative index вроде `-1` воспринимается как `0xFFFFFFFF` и тоже уходит в default.

Microsoft Visual C++

MSVC сначала сдвигает диапазон к нулю:

```
sub ecx, 1
cmp ecx, 0Bh
ja  loc_default
jmp ds:off_401478[ecx*4]
```

Теперь 0 соответствует исходному case 1, поэтому dummy slot не нужен и table имеет 12 entries. Она может располагаться сразу после function, отдельно от code blocks.

gcc

gcc мог разместить table в read-only `.rdata` и выполнить indirect jump через register:

```
cmp [ebp+arg_0], 0Ch
ja  loc_default
mov eax, [ebp+arg_0]
shl eax, 2
mov eax, ds:off_402010[eax]
jmp eax
```

Здесь снова 13 entries и case 0 ведёт в default. IDA распознаёт все три варианта, но отсутствие её comments не доказывает отсутствие switch. Нужно самостоятельно видеть range check, scale index, table of code pointers и indirect branch.

RTTI

Единого binary format C++ RTTI нет. IDA исторически лучше знала Borland. Для Microsoft существовали IDC scripts и Class Informer plugin.

Практический способ изучить незнакомый compiler:

1. Написать маленькую программу с classes и virtual methods.
2. Собрать теми же tools.
3. Найти в IDA strings с class names.
4. По data xrefs найти RTTI candidate, содержащий pointer на name.
5. Идти по обратным data xrefs до table function pointers, то есть vtable.

Общим свойством разных RTTI остаётся связь со строковым именем класса.

Поиск main

Entry point не равен `main`: startup code готовит runtime, constructors, arguments и termination. В других языках главная function может называться иначе. Startup FLIRT signatures часто находят её автоматически, но покрытие неполно, а obfuscation разрушает match. Для ELF/Mach-O рассматриваемая IDA могла не иметь startup signatures.

У обычного C `main` три аргумента: `argc`, `argv`, `envp`. Полезные признаки: calls initialization перед ним, `exit` после него, три prepared arguments и передача return value в exit.

gcc на FreeBSD

Типичный stripped dynamic startup:

```
mov  [esp], offset term_proc
call atexit
call init_proc
lea  eax, [ebp+arg_0]
mov  [esp+8], esi
mov  [esp+4], eax
mov  [esp], ebx
call sub_8048400 ; main
mov  [esp], eax
call exit
```

В static stripped имён library не будет, но main остаётся единственным call с тремя arguments. Раннее применение FLIRT восстанавливает подсказки.

gcc на Linux

Linux startup обычно вызывает единственную `__libc_start_main` с несколькими parameters. Последний pushed, то есть first argument, является pointer на main:

```
push offset fini
push offset init
push ecx
push esi
push offset loc_8048384 ; main
call __libc_start_main
```

IDA может назвать target только `loc_...`, а не function: его не вызывает direct `call`, а unusual prologue начинается с alignment и лишь затем `push ebp; mov ebp,esp`. Нужно вручную Create Function, затем при необходимости `Alt+P` включить BP-based frame и исправить saved registers/local size.

В static stripped имя `__libc_start_main` исчезнет, но остаётся шаблон: один call и first parameter указывает main.

gcc/Cygwin и MinGW

Cygwin startup также делает один call и передаёт function pointer:

```
mov [esp], offset sub_4010B0 ; main
call sub_401120
```

MinGW сначала вызывает runtime helper без очевидных arguments. Внутри него ищут единственный nonlibrary call с тремя parameters. Сильная подсказка - предварительный `getmainargs` и третье значение от `__p__environ`, соответствующее `envp`.

Microsoft Visual C++

Entry point короткий:

```
call __security_init_cookie
jmp  __tmainCRTStartup
```

Startup signatures обычно дают имена. Внутри `__tmainCRTStartup` main выделяется тремя pushes `envp`, `argv`, `argc`, call и последующим `cexit`. Имя `_main` может быть полностью создано сигнатурой, даже если строка `main` отсутствует в stripped file.

Borland

Старт передаёт `__startup` pointer на служебную structure, содержащую main. В `__startup` main снова является единственным indirect call с тремя args, а EAX сразу передаётся `_exit`. Calls `GetEnvironmentStrings` и `GetCommandLine` служат дополнительными признаками.

Visual Basic 6

VB6 entry point:

```
push offset runtime_info
call ThunRTMain
```

`runtime_info` содержит единственный pointer в code segment, который обычно ведёт к main. Это тот же общий принцип: runtime получает descriptor с адресом пользовательского entry.

Если готовый шаблон неизвестен, полезно собрать minimal test тем же compiler, сослаться в main на легко узнаваемую string и изучить startup. Затем pattern переносится на исследуемый binary.

Debug и Release

Visual Studio Debug обычно не оптимизируется, содержит symbols, debug runtime и Basic Runtime Checks `/RTCx`. Release оптимизируется и лишён этой instrumentation. Дополнительный code мешает startup signatures.

Debug часто вставляет thunk functions:

```
start:      jmp start_0
sub_411050: jmp sub_412AE0
start_0:    call sub_411050
```

Одноинструкционные jumps создают лишний слой между caller и implementation.

Debug prologue может выделить намного больше stack, чем нужно, и заполнить его `0xCC`:

```
push ebp
mov  ebp, esp
sub  esp, 0F0h
push ebx
push esi
push edi
lea  edi, [ebp+var_F0]
mov  ecx, 3Ch
mov  eax, 0CCCCCCCCh
rep stosd
```

Это эквивалентно `memset(&var_F0,0xCC,240)`. Byte `0xCC` является x86 `int 3`, вызывающим debugger при случайном исполнении stack. Extra gaps между variables позволяют заметить overwrite.

Release того же code выделяет только `0x10`, не заполняет stack и располагает четыре locals рядом. Оптимизированный code может быть компактнее и быстрее, но труднее связывается с source.

Нестандартные calling conventions

Внутренние optimized functions не обязаны следовать public ABI. Если register используется до initialization, он, вероятно, содержит parameter от caller.

Пример function имеет один stack arg, но сразу читает `EAX` и `CL`, значит фактически arguments три. Edit > Functions > Set Function Type позволяет задать `__usercall`.

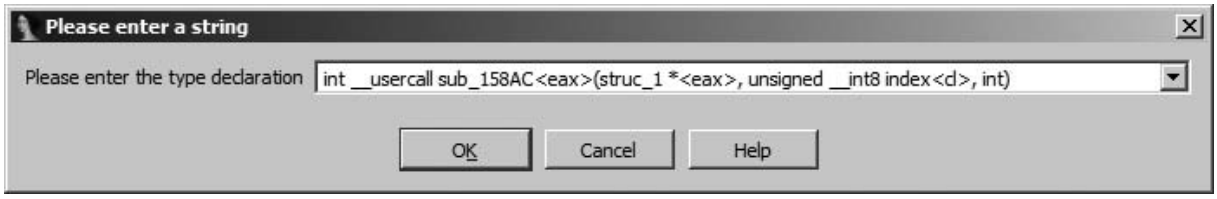

Рис. 20-1. Объявление функции как __usercall

Прототип:

```
int __usercall sub_158AC<eax>(struc_1 *<eax>,
                              unsigned __int8 index<cl>, int);
```

После function name `<eax>` задаёт return register. Для `void` он не нужен. В parameters register annotation добавляется к type. Третий parameter без annotation лежит на stack.

После применения IDA подписывает подготовку вызова:

```
lea  eax, [ebp+var_218] ; struc_1 *
mov  cl, 1              ; index
push edx                ; int
call sub_158AC
```

В том же binary другая function может принимать register argument в `ESI`. Custom convention следует выводить отдельно для каждой function, проверяя происхождение каждого register.

Итоги

Compiler personality проявляется в switch lowering, RTTI layout, startup, optimization, debug instrumentation и calling conventions. Даже один compiler выдаёт разные patterns на другой platform или с другими flags. Надёжный анализ сочетает annotations IDA, знание ABI, собственные test builds и изучение data/control flow, а не полагается на единственный знакомый шаблон.

Глава 21. Анализ обфусцированного кода

Даже в идеальных условиях понять дизассемблированный листинг непросто. Качественный дизассемблер необходим каждому, кто исследует внутреннее устройство бинарного файла, и эффективность IDA заметно снизила порог входа в эту область. Одновременно разработчики защитных средств получили дополнительный стимул скрывать устройство программ. Так возникло постоянное противоборство между специалистами по обратной разработке и авторами кода, не желающими раскрывать его работу.

Обфускация делает объект неясным и запутанным, чтобы затруднить понимание. Anti-reverse engineering - более широкое понятие: оно охватывает все методы, мешающие анализу, а обфускация является лишь одним из них. Здесь речь идет о приемах, применяемых к исполняемым бинарным файлам. Категории неизбежно пересекаются, а новые приемы появляются постоянно, поэтому приведенная классификация не претендует на абсолютную полноту.

Методы против статического анализа

Их основная задача - не дать аналитику понять программу без запуска. Именно они непосредственно атакуют дизассемблеры, включая IDA.

Рассинхронизация дизассемблирования

Один из старейших приемов смешивает инструкции и данные так, чтобы дизассемблер потерял правильную границу инструкции. Например, Shiva выполняет `call` не на начало, а в середину уже распознанной команды:

```
LOAD:0A04B0D1  call  near ptr loc_A04B0D6+1
LOAD:0A04B0D6  mov   dword ptr [eax-73h], 0FFEB0A40h
LOAD:0A04B0DD  loopne loc_A04B06F
LOAD:0A04B0DF  mov   dword ptr [eax+56h], 5CDAB950h
LOAD:0A04B0E6  iret
LOAD:0A04B0E7  db 47h
LOAD:0A04B0E8  db 31h, 0FFh, 66h
```

Цель находится по адресу `0A04B0D7`, но соответствующие байты IDA уже включила в пятибайтовую инструкцию с адреса `0A04B0D6`. Неожиданный `iret` в пользовательском коде и россыпь байтов данных служат признаками ошибки. Такой прием обманывает и recursive descent, и linear sweep дизассемблеры.

Исправление в IDA выполняется интерактивно: на ошибочной инструкции нажмите `U` (Edit > Undefine), затем на истинном адресе цели - `C` (Edit > Code). После первого исправления получается:

```
LOAD:0A04B0D1  call  loc_A04B0D7
LOAD:0A04B0D6  db 0C7h
LOAD:0A04B0D7  pop eax
LOAD:0A04B0D8  lea eax, [eax+0Ah]
LOAD:0A04B0DB  jmp short near ptr loc_A04B0DB+1
LOAD:0A04B0DD  db 0E0h
```

Байт `0A04B0D6` никогда не исполняется. `pop eax` снимает со стека фиктивный return address, после чего выполнение продолжается. Однако следующая двухбайтовая команда прыгает в середину самой себя. После еще одного цикла `U` и `C` видна реальная последовательность:

```
LOAD:0A04B0D7  pop eax
LOAD:0A04B0D8  lea eax, [eax+0Ah]
```

```
LOAD:0A04B0DB  db 0EBh
LOAD:0A04B0DC  jmp eax
```

Теперь цель скрыта в регистре. В EAX попал адрес возврата 0A04B0D6, затем к нему прибавили 10, поэтому фактическая цель - 0A04B0E0. Аналитик должен продолжить создание кода именно там.

Условный переход также можно превратить в безусловный, заранее задав флаги:

```
.text:00401000  xor eax, eax
.text:00401002  jz  short near ptr loc_401009+1
.text:00401004  mov ebx, [eax]
.text:00401006  mov [ecx-4], ebx
.text:00401009  call near ptr 0ADFEFFC6h
.text:0040100E  ficom word ptr [eax+59h]
```

xor eax,eax гарантированно устанавливает ZF, поэтому jz всегда выполняется. Команды между ним и целью являются мертвым кодом, а сама цель снова находится внутри неверно распознанной инструкции. После исправления видно, что лишним был байт E8h, а реальный код начинается с mov eax,0DEADBEEFh. Чем длиннее цепочка операций, влияющих на флаги перед условным переходом, тем труднее доказать, какая ветвь достижима.

Динамически вычисляемые адреса перехода

Слово "динамически" здесь не означает защиту от динамического анализа. Оно означает, что адрес следующего участка вычисляется во время исполнения и потому отсутствует как явная константа в инструкции перехода.

В завершении стартового кода Shiva адрес строится арифметически, а затем подменяет верхушку стека:

```
mov  ecx, 7F131760h  ; ecx = 7F131760
xor  edi, edi        ; edi = 00000000
mov  di, 1156h       ; edi = 00001156
add  edi, 133AC000h  ; edi = 133AD156
xor  ecx, edi        ; ecx = 6C29C636
sub  ecx, 622545CEh  ; ecx = 0A048068
mov  edi, ecx
pop  eax
pop  esi
pop  ebx
pop  edx
pop  ecx
xchg edi, [esp]      ; TOS = 0A048068
retn                  ; переход на 0A048068
```

Статический анализ вынужден вручную исполнять такие цепочки. Более сложные варианты делят вычисление между потоками или дочерними процессами и передают результат через IPC, семафоры и другие средства синхронизации. Тогда необходимо понять не только каждую исполнительную сущность, но и точный порядок обмена данными.

Еще один распространенный прием использует Structured Exception Handling. tElock получает текущий адрес через call/pop, вычисляет адрес обработчика, помещает его в цепочку через fs:[0], а затем намеренно вызывает int 3. Позднее тот же код может спровоцировать деление на ноль. Обработчик получает структуру CONTEXT, меняет сохраненный Eip и тем самым выбирает продолжение, а заодно очищает Dr0-Dr7, уничтожая аппаратные точки останова. Обычный граф IDA такого неявного перехода не показывает.

Обфускация машинных команд

Оригинальные код и данные можно зашифровать, сжать или преобразовать и оставить открытым только небольшой распаковщик. Точка входа направляется в этот stub; он восстанавливает программу и передает управление на original entry point, OEP.

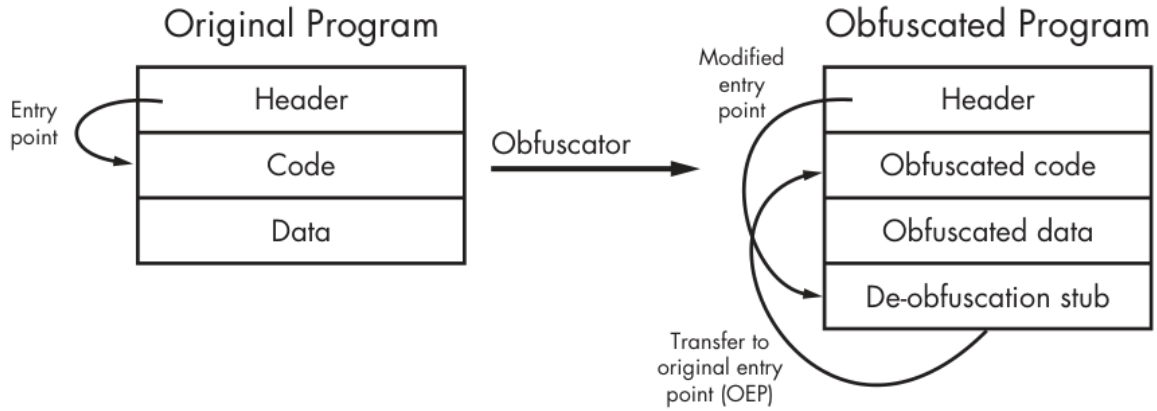

Figure 21-1: Generic obfuscation process

Рис. 21-1. Общая схема обфускации и последующего восстановления программы

Этим занимаются packer/protector-инструменты, например UPX и ASPack. Само наличие упаковщика не обязательно означает злой умысел, но для анализа сначала требуется получить исходный код программы. Хорошим индикатором служит несоответствие между исполняемыми сегментами и результатом автоанализа. У UPX сегменты `UPX0` и `UPX1` исполняемые, однако `UPX0` в файле пуст и будет заполнен после распаковки. Поэтому IDA видит лишь небольшой участок в `UPX1`, а Navigator оставляет почти всю адресную область неисследованной.

Рис. 21-2. Navigator показывает малую долю распознанного кода в упакованном UPX файле

Сокрытие импортируемых функций

Таблица импорта заранее раскрывает многие возможности программы, поэтому защитные средства стараются ее уменьшить или восстановить только во время исполнения. У tElock в статическом импорте могут остаться лишь `GetModuleHandleA` и `MessageBoxA`. UPX обычно импортирует `LoadLibraryA` и `GetProcAddress`, а затем в цикле заново строит настоящую таблицу.

tElock может вообще вручную найти `kernel32`, пройти ее export directory и сравнивать части имен с константами, которые складываются в `LoadLibraryA`. Более развитый вариант хеширует каждое имя экспорта и сравнивает только хеш. Для восстановления имен приходится обратить алгоритм хеширования или заранее вычислить словарь хешей известных API.

Поврежденная или намеренно нестандартная таблица импорта может вызвать предупреждение IDA. Просьба загрузчика "сделать импорт более нормальным" сама по себе является важной уликой, а не причиной автоматически игнорировать сообщение.

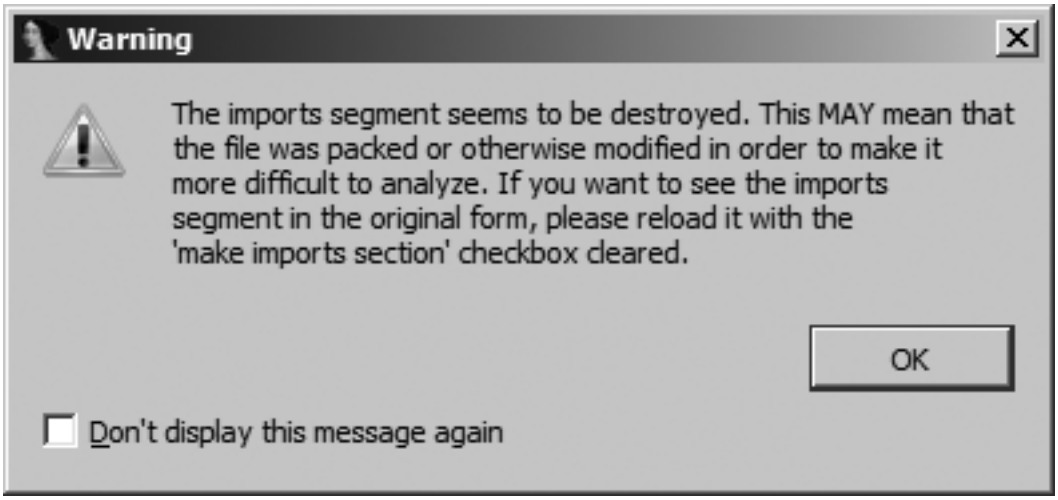

Рис. 21-3. IDA предупреждает о намеренно искаженной таблице импорта

Целенаправленные атаки на анализаторы

Загрузчик и процессорный модуль являются парсерами недоверенного формата и тоже могут содержать уязвимости. Специально подготовленный заголовок или поток инструкций способен атаковать саму IDA. Подозрительный файл разумно впервые открывать в изолированной среде. Если анализ выполняет группа, один сотрудник может безопасно создать IDB и распространять уже базу, а не исходный образец. Это не устраняет все риски, но сокращает число систем, непосредственно обрабатывающих враждебный формат.

Методы против динамического анализа

Эти приемы пытаются обнаружить или нарушить контролируемое исполнение. Они особенно важны для отладчиков, но влияют и на стратегии деобфускации в IDA.

Обнаружение виртуализации

Вредоносная программа может искать VMware Tools в реестре, проверять OUI сетевого MAC-адреса или использовать VMware backdoor: загрузить сигнатуру `VMXh` и выполнить привилегированную `in eax,dx`, поведение которой внутри VMware отличается от настоящего компьютера. Известный прием Red Pill использует `sidt`: положение IDT в виртуальной машине часто отличается от ожидаемого на физической системе.

Любой одиночный признак может дать ложное срабатывание, поэтому серьезная защита комбинирует несколько тестов или просто изменяет поведение, не завершая процесс явно.

Обнаружение инструментирования

Программа может искать процессы, файлы, драйверы, окна и классы окон известных анализаторов. Например, вызов `FindWindow("FilemonClass", NULL)` обнаруживал Filemon. Простое переименование исполняемого файла помогает только против поверхностной проверки: классы окон, устройства и особенности поведения остаются.

Обнаружение отладчика

В Windows доступны `IsDebuggerPresent`, флаги в PEB и другие системные признаки. Старые защиты проверяли наличие устройства SoftICE, например `\\.\NTICE`. Проверки могут выполняться напрямую, через исключения или через побочные эффекты, чтобы точку принятия решения было труднее найти.

Препятствие отладке

Защита намеренно генерирует исключения, неоднозначно использует `int 3`, проверяет контрольные суммы кода и замечает вставленные программные breakpoints. Если код расшифровывается лишь перед выполнением, адрес для точки останова заранее неизвестен. Обработчик исключений может очищать debug registers и снимать аппаратные точки.

В Unix-подобной системе Shiva применяет взаимный `ptrace`: родитель присоединяется к ребенку, а ребенок - к родителю. Поскольку к процессу обычно может быть присоединен только один трассировщик, внешний отладчик не получает доступа. Если одна из взаимных операций не удалась, защита делает вывод о присутствии аналитика.

Статическая деобфускация бинарных файлов в IDA

Статический подход особенно полезен, когда нужной архитектуры или ОС нет под рукой, запуск образца слишком опасен либо защита активно распознает отладчик. В IDA есть два основных пути: написать сценарий, имитирующий алгоритм, или использовать эмулятор инструкций.

Деобфускация с помощью сценария

У Burneye стартовый decoder использует два вложенных цикла. Сценарий читает начальные значения из базы, повторяет регистровые операции и патчит каждый восстановленный байт. Схематически IDC-реализация выглядит так:

```
static main() {
  auto ecx, esi, edi, ebx, edx, eax, i, j, carry, value;

  ecx = Dword(0x05371000);
  esi = 0x05371087;
  edi = esi;
  ebx = Dword(0x05371004);
  edx = 0;

  for (i = 0; i < ebx; i = i + 1) {
    for (j = 0; j < 8; j = j + 1) {
      carry = edx & 1;
      edx = (edx >> 1) & 0x7fffffff;
      if (carry) edx = edx ^ 0xC0000057;
    }
    eax = Byte(esi);
    esi = esi + 1;
    value = (eax ^ edx) & 0xff;
    PatchByte(edi, value);
    edi = edi + 1;
  }
```

```
}
```

Реальный перевод должен точно повторять семантику инструкций. В IDC `>>` может быть знаковым, поэтому после логического сдвига нужен mask. Аналогично `lodsb` и `stosb` меняют только младшие восемь бит аккумулятора или памяти и корректируют индексные регистры. После патчинга следует обновить окно Strings и запустить анализ по адресу возврата, который decoder первоначально положил на стек.

Преимущество сценария фундаментально: враждебный бинарный код ни разу не исполняется. Инструкции можно переводить по одной, даже не понимая алгоритм на высоком уровне. Недостаток - хрупкость: адреса, размер блока и небольшое изменение версии packer делают сценарий непригодным. Универсального распаковщика из такого кода обычно не получается.

Деобфускация с помощью эмулятора

Эмулятор исполняет семантику инструкций программно, не передавая их настоящему процессору. Поэтому он работает на другой платформе, не требует ручного переписывания каждой операции и не запускает вредоносный код нативно. IDB выступает виртуальной памятью; plugin создает stack и heap, хранит регистры, а записи самомодифицирующегося кода сразу вносит в базу.

Основы x86emu

x86emu вызывается через `Alt-F8`, поддерживает x86 и может работать с PE, ELF и Mach-O. Главное окно показывает регистры и команды управления.

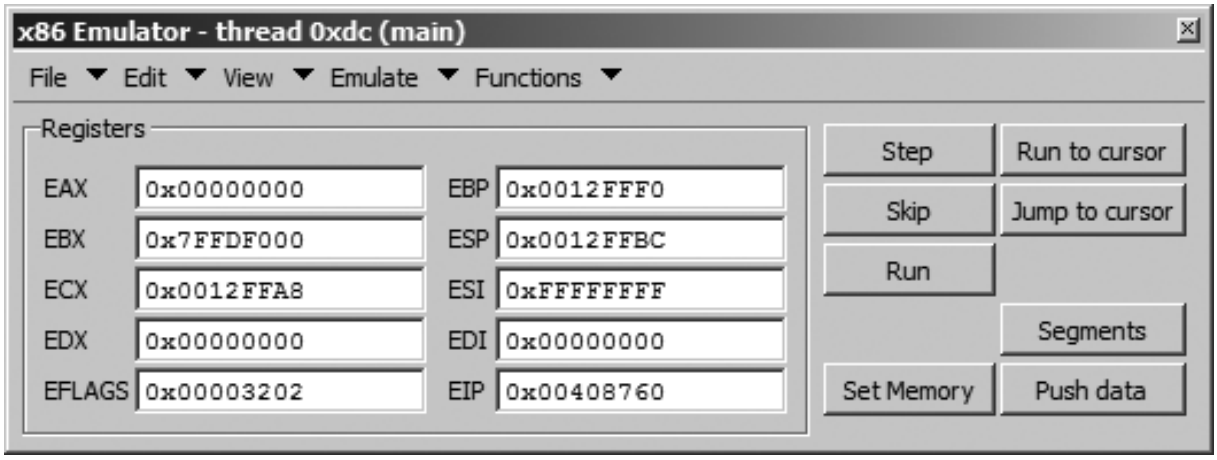

Рис. 21-4. Диалог управления x86emu

При активации создаются сегменты `.stack` и `.heap`, а начальный `EIP` берется с позиции курсора. Для PE эмулятор также моделирует заголовки, PEB, TEB, segment registers и IDT, загружает export directories библиотек без их машинного кода и заполняет импорты. Состояние сохраняется в netnode и созданных сегментах, поэтому не теряется вместе с закрытием диалога.

Основные команды:

• `Step` эмулирует одну инструкцию, при необходимости превращает байты по `EIP` в code и следует за новой позицией.
• `Run To Cursor` работает до внутренней точки останова или позиции курсора. Если управление туда не придет, выполнение может быть долгим.
• `Skip` лишь продвигает `EIP`. При пропуске библиотечного вызова аналитик обязан сам задать результат в `EAX`, заполнить выходной буфер и поправить `ESP`.

• `Jump To Cursor` немедленно задает `EIP`; состояние stack при этом должно соответствовать новой точке.
• `Run` продолжает работу до breakpoint и наиболее рискован при неизвестном control flow.

Диалог `Segments` позволяет изменить selectors и bases сегментных регистров. Это практическая модель окружения, а не полный редактор GDT.

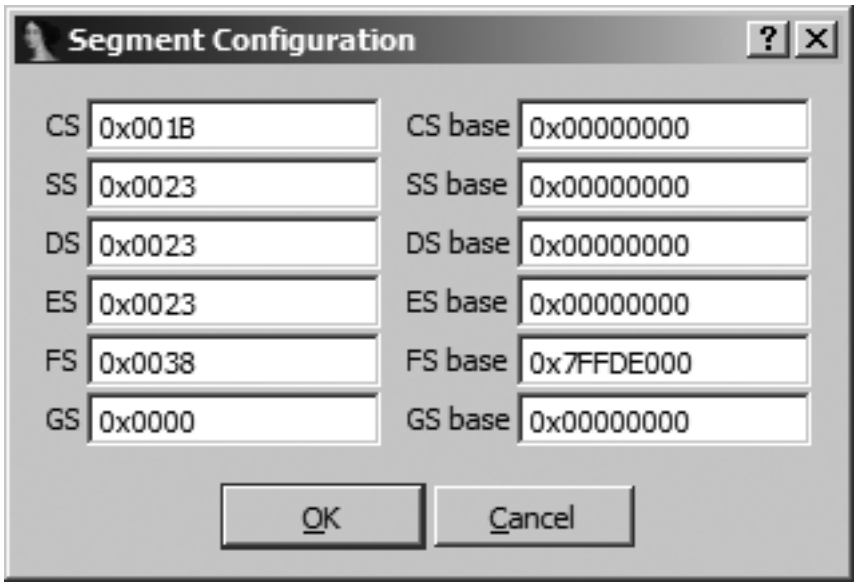

Рис. 21-5. Настройка сегментных регистров в x86eти

`Set Memory` записывает 8-, 16- или 32-битные значения, ASCII-строки либо содержимое файла по выбранному адресу. По сути это удобная оболочка над операциями `PatchXXX`.

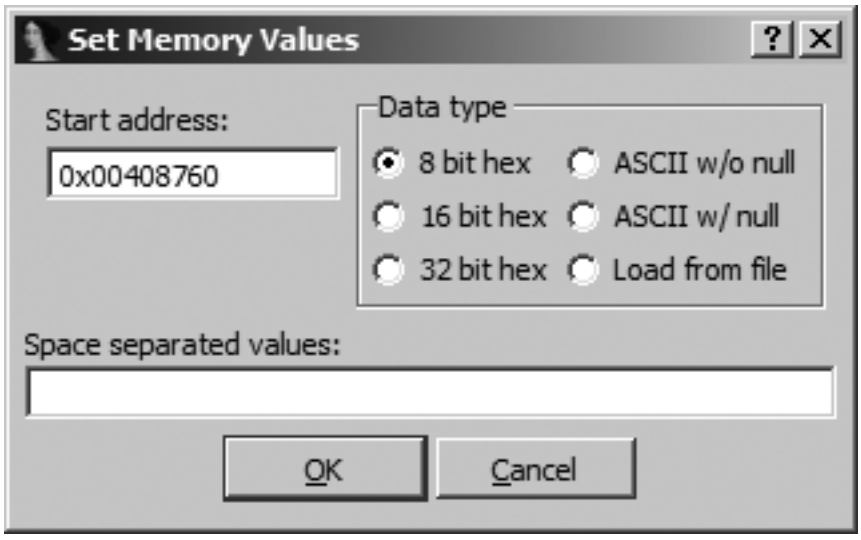

Рис. 21-6. Диалог изменения памяти x86eти

`Push Data` помещает числовые 4-байтовые значения на стек. Список обрабатывается справа налево, как при обычной передаче аргументов.

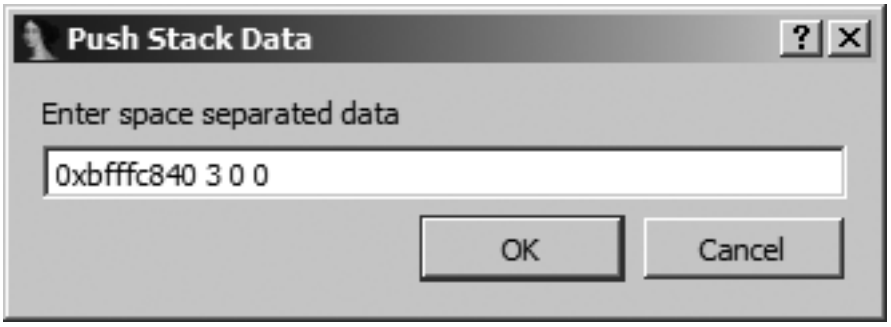

Рис. 21-7. Диалог помещения данных на стек x86emu

Breakpoints эмулятора хранятся во внутреннем списке. Он не вставляет `int 3` и не использует hardware debug registers, поэтому эмулируемая программа не может обнаружить такие точки стандартными способами.

Для Burneye достаточно пройти первые шесть инструкций, выполнить `Run To Cursor` до return decoder, а затем сделать еще два шага. Это заметно проще ручного IDC-перевода. Для UPX удобно дойти до циклов восстановления импортов и перехватывать `LoadLibrary` и `GetProcAddress`.

Эмуляция библиотечных функций

x86emu реализует ряд распространенных Windows API: проверки отладчика и версии, tick count, heap, TLS, `malloc`, `VirtualAlloc`, `LoadLibrary`, `GetProcAddress` и другие. Если реализация неизвестна, plugin использует доступную в IDA type information, извлекает со стека аргументы и показывает их пользователю.

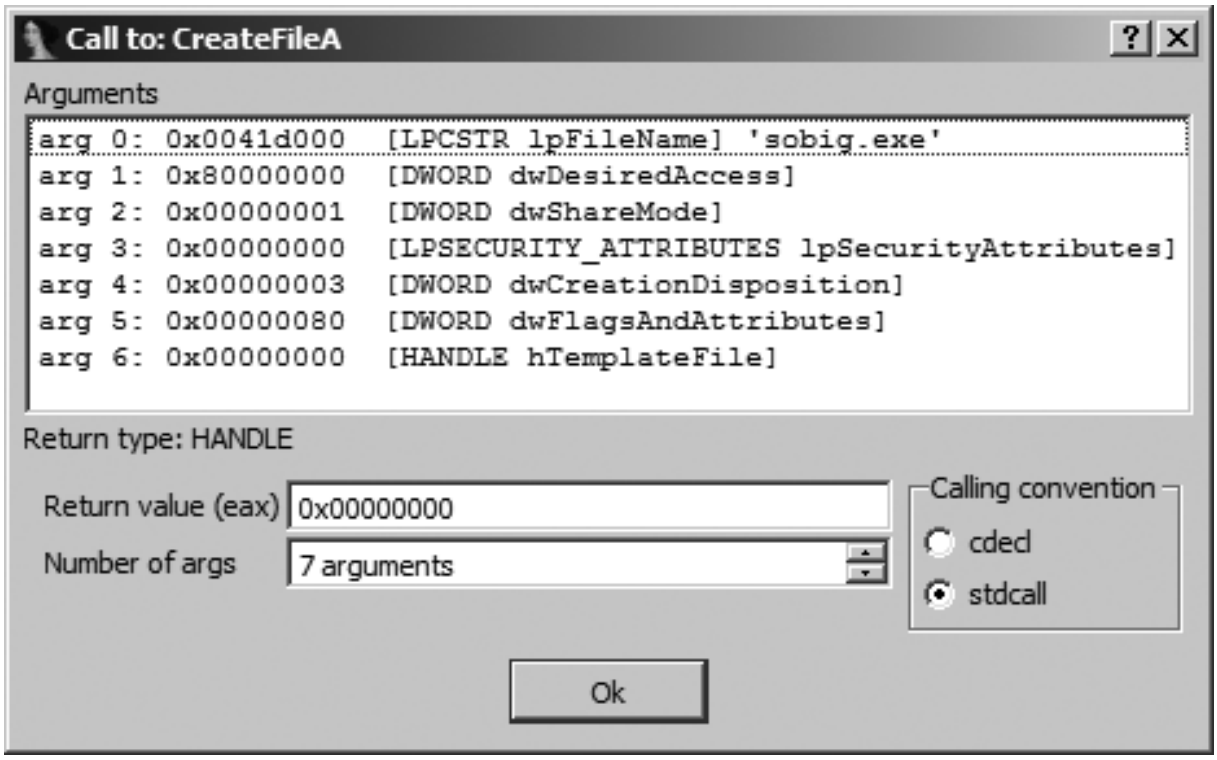

Рис. 21-8. Диалог библиотечной функции x86emu

Пользователь задает return value, calling convention и число аргументов. Для `stdcall` число аргументов необходимо, чтобы эмулятор правильно очистил стек. Типы и формальные имена выводятся только тогда, когда IDA располагает соответствующим прототипом.

При прохождении UPX в Output появляется журнал наподобие:

```
x86emu: LoadLibrary called: KERNEL32.DLL (7C800000)
x86emu: GetProcAddress called: ExitProcess (0x7C81CDDA)
x86emu: GetProcAddress called: ExitThread (0x7C80C058)
x86emu: GetProcAddress called: GetCurrentProcess (0x7C80DDF5)
x86emu: GetProcAddress called: GetCurrentThread (0x7C8098EB)
x86emu: GetProcAddress called: GetFileSize (0x7C810A77)
x86emu: GetProcAddress called: GetModuleHandleA (0x7C80B6A1)
x86emu: GetProcAddress called: CloseHandle (0x7C809B47)
```

Он фиксирует загруженные библиотеки и разрешенные функции. После обфускации symbol information обычно потеряна, поэтому простого восстановления адресов недостаточно. В цикле UPX адрес возвращенный `GetProcAddress` сохраняется так:

```
UPX1:00408897  call dword ptr [esi+8090h] ; GetProcAddress
UPX1:0040889D  or   eax, eax
UPX1:0040889F  jz   short loc_4088A8
UPX1:004088A1  mov  [ebx], eax            ; запись в import table
UPX1:004088A3  add  ebx, 4
```

Командой `Emulate > Windows > Set Import Address Save Point` адрес `004088A1` обозначается как точка сохранения импорта. Сделать это нужно до ее эмуляции. При каждой записи x86emu определяет функцию и присваивает ячейке имя. Вместо безымянных `dd 7C81CDDAh` получаются `ExitProcess`, `ExitThread`, `GetCurrentProcess`, `GetCurrentThread` с прототипами и xrefs. После этого IDA снова может аннотировать вызовы типами параметров.

Дополнительные возможности x86emu

• `File Dump` сохраняет выбранный диапазон адресов базы.
• `File Dump Embedded PE` распознает PE по позиции курсора, определяет размер из headers и извлекает встроенный executable.
• `View > Enumerate Heap` выводит все выделенные heap blocks.
• `Emulate > Switch Thread` переключает моделируемые потоки. x86emu перехватывает `CreateThread`, но собственного scheduler у него нет.
• `Functions > Allocate Heap Block` и `Allocate Stack Block` резервируют scratch memory и сообщают ее адрес.

x86emu и anti-debugging

Хотя x86emu не является обычным debugger, ему необходимо правдоподобно моделировать runtime. Для защиты от временных проверок он ведет собственный TSC и увеличивает его на каждую эмулированную инструкцию. Сколько реального времени аналитик провел между двумя `rdtsc`, неважно: разность остается пропорциональной лишь числу виртуальных команд.

Эмулятор поддерживает базовый Windows SEH. При exception или software interrupt он строит `CONTEXT`, проходит цепочку через `fs:[0]` и передает управление обработчику. После возврата состояние восстанавливается из `CONTEXT`, включая изменения, сделанные защищаемой программой.

Поведение `Dr0`-`Dr7` также эмулируется, но внутренние breakpoints с этими регистрами не связаны. Поэтому очистка debug registers обработчиком исключения не нарушает работу x86emu.

Обфускация на основе виртуальной машины

Самые сложные obfuscators переписывают программу в собственный bytecode и добавляют native virtual machine, исполняющую его. Анализ одного лишь native interpreter не раскрывает назначение программы: логика остается в embedded bytecode. Нужно найти все его блоки и восстановить instruction set виртуальной машины.

VMProtect является коммерческим примером такого подхода. Учебный HyperUnpackMe2 также использует VM: основной вызов состоит в поиске встроенной программы и определении смысла каждого opcode.

Можно написать отдельный processor module для нового bytecode. Так поступил Rolf Rolles при анализе HyperUnpackMe2. Недостаток - одна IDB использует один processor module, поэтому x86 и виртуальные команды приходится смотреть в разных базах. Более удобный путь - plugin, расширяющий существующий x86 processor через `custom_ana`, `custom_emu` и `custom_out`. Тогда оба набора команд видны вместе:

```
TheHyper:01013B2F  h_pop.l R9
TheHyper:01013B32  h_pop.l R7
TheHyper:01013B35  h_pop.l R5
TheHyper:01013B38  h_mov.l SP, R2
TheHyper:01013B3C  h_sub.l SP, 0Ch
TheHyper:01013B44  h_pop.l R2
TheHyper:01013B47  h_pop.l R1
TheHyper:01013B4A  h_retn 0Ch
TheHyper:01013B4D  dd 24242424h
TheHyper:01013B51  dd 0A9A4285Dh ; TAG VALUE
TheHyper:01013B55  push ebp
TheHyper:01013B56  mov  ebp, esp
```

IDA 5.7 добавила custom datatypes и custom data formats. Сценарий или plugin регистрирует имя menu и функцию форматирования; IDA вызывает ее при отображении помеченного объекта. Можно завести datatype для VM bytecode и показывать каждое значение как команду. Однако каждую виртуальную инструкцию придется найти и пометить вручную. Расширение processor лучше интегрировано с recursive analysis: достаточно определить одну инструкцию, после чего `custom_emu` сообщает IDA достижимые цели и помогает автоматически открыть остальной bytecode.

Итоги

Для современного вредоносного ПО обфускация скорее правило, чем исключение. Анализ почти неизбежно начинается с деобфускации: динамической под debugger либо статической с помощью сценария или эмулятора. Конечная цель одна - получить восстановленный бинарный образ, который можно полноценно дизассемблировать и исследовать.

Если итоговый анализ все равно выполняется в IDA, логично использовать ее на всем пути. Интерактивное исправление границ, IDC, processor extensions и x86emu показывают, что IDA способна гораздо больше, чем просто вывести листинг. В главе 25 к обфусцированному коду мы вернемся уже со стороны встроенных средств отладки.

Глава 22. Анализ уязвимостей

Сразу уточним: IDA не является автоматическим средством поиска уязвимостей. Она не подсвечивает опасный код красным и не создает exploit по щелчку мыши. Без квалифицированного пользователя, а иногда и набора scripts и plugins, это прежде всего disassembler и debugger. IDA облегчает статический анализ, но результат определяется знаниями аналитика.

Для обнаружения совершенно новых ошибок fuzzing часто продуктивнее чистого статического анализа. Зато после обнаружения дефекта IDA в сочетании с debugger становится одним из лучших средств для понимания причины и разработки proof of concept. Еще одна важная роль - поиск уже исправленных, но не описанных vendor-ом уязвимостей путем сравнения старого и нового бинарных файлов. Patch Tuesday сделал такой differential analysis регулярной практикой: изменение сужает область поиска и показывает код, который раньше был уязвим.

Эта глава не заменяет систематический курс по buffer overflow, format string и другим классам ошибок. Ее задача - показать, где именно возможности IDA помогают найти подозрительные места, понять stack layout и выбрать полезные элементы для дальнейшей разработки exploit.

Поиск новых уязвимостей с помощью IDA

При наличии source code можно применять автоматические аудиторы и ручную проверку. Binary auditing дает сходные возможности без исходников и потому применим к закрытым продуктам. Однако ни один анализатор не гарантирует обнаружения всех дефектов: автоматизация хорошо находит типовые случаи, а сложные data-flow зависимости требуют человеческого понимания.

Базовый статический аудит ищет:

• опасные функции наподобие `strcpy` и `sprintf`;
• использование buffers, полученных через `malloc` и `VirtualAlloc`;
• обработку пользовательского ввода из `recv`, `read`, `fgets` и аналогичных источников;
• несоответствие реального размера destination возможному размеру source.

Поиск вызовов опасной функции

Для `strcpy` ручной процесс выглядит просто:

1. Найти саму функцию.
2. Открыть все xrefs через View > Open Subviews > Cross References.
3. Посетить каждый call site и восстановить значения параметров.

Третий шаг может потребовать серьезного data-flow analysis. Даже первый зависит от формата файла. В PE и статически связанном ELF команда `G` с именем `strcpy` часто сразу приводит к нужной функции. В dynamically linked ELF она может привести в `extern`:

```
extern:804DECC  extrn strcpy:near ; CODE XREF: _strcpy
extern:804DECC                    ; DATA XREF: .got:off_804D5E4
```

Code xref ведет в procedure linkage table:

```
.plt:08049E90 _strcpy proc near
.plt:08049E90         jmp ds:off_804D5E4
.plt:08049E90 _strcpy endp
```

Data xref ведет в global offset table:

```
.got:0804D5E4 off_804D5E4 dd offset strcpy
```

На практике переход по data xrefs из `extern` через `.got` в `.plt` наиболее надежно приводит к callable address. Точка в имени `.strcpy` может отображаться как underscore, поскольку старые версии IDA не считали точку допустимым символом identifier.

Автоматизация поиска

Когда список функций велик, рутинную навигацию стоит поручить IDC. Вспомогательная функция возвращает адрес, по которому следует перебирать code xrefs:

```
static getFuncAddr(fname) {
  auto func, seg;
  func = LocByName(fname);
  if (func == BADADDR) return BADADDR;
  seg = SegName(func);

  if (seg == "extern") {
    func = DfirstB(func);       // extern -> GOT
    if (func == BADADDR || SegName(func) != ".got") return BADADDR;
    func = DfirstB(func);       // GOT -> PLT
    if (func == BADADDR || SegName(func) != ".plt") return BADADDR;
  }
  return func;
}
```

Затем script перебирает `RfirstB`/`RnextB`, добавляет comments в прямые call sites и отдельно обрабатывает wrappers или thunk-функции. Поиск становится поиском по комментарию, но решение об опасности по-прежнему требует определить происхождение аргументов и размеры объектов.

Это принципиальная граница: найти `sprintf` легко, а доказать overflow трудно. Нужно восстановить assignments, propagation значений, stack frames, globals и размеры dynamic allocations. Инструменты вроде BugScam демонстрируют возможную автоматизацию: находят опасные calls и выполняют ограниченный анализ аргументов, например оценивают максимальный результат `sprintf` по размеру источников и format specifiers. Такая эвристика полезна, но не равна полному data-flow engine.

Поиск stack buffers

IDA Python позволяет перечислить members frame structure и выделить достаточно крупные локальные массивы:

```
def scan_stack_buffers(func_ea, min_size=16):
    frame = GetFrame(func_ea)
    if frame is None:
        return
    size = GetStrucSize(frame)
    off = 0
    while off < size:
        name = GetMemberName(frame, off)
        if name:
            item_size = GetMemberSize(frame, off)
            if item_size >= min_size:
                print("%08x %-24s %d" % (func_ea, name, item_size))
            off += max(item_size, 1)
        else:
            off += 1
```

Такой script ищет candidates, а не уязвимости. Большой buffer может использоваться безопасно, а маленький объект может переполняться. Точность также зависит от того, правильно ли IDA

построила frame и типы variables.

Поиск уже исправленных уязвимостей

Vendor часто выпускает binary patch без точного описания причины. Сравнение unpatched и patched versions выделяет измененные functions. Если обновление связано с security, эти изменения становятся картой для восстановления исходной ошибки.

Перед сравнением обе базы нужно анализировать одинаково: одинаковые loader options, signatures, compiler settings и scripts. Иначе различия конфигурации будут выглядеть как изменения программы. Даже пересборка тем же source способна изменить адреса, register allocation и порядок blocks, поэтому сравнение по raw bytes недостаточно.

Инструменты binary diffing

BinDiff, Turbodiff и PatchDiff2 сопоставляют functions по совокупности признаков: control-flow graph, calls, constants, strings, число blocks и другие характеристики. Они создают backend-данные и позволяют визуально сравнивать графы. PatchDiff2 представляет свободно доступный IDA plugin.

Рабочий процесс PatchDiff2:

1. Создать отдельные IDB для старого и нового файла и дождаться завершения анализа.
2. Открыть базу оригинала и запустить plugin через Edit > Plugins или `Ctrl-8`.
3. Выбрать вторую IDB.
4. Изучить списки `Identical Functions`, `Matched Functions` и `Unmatched Functions`.

Identical functions можно отложить. Matched functions похожи, но не идентичны, и часто содержат именно security fix. Unmatched functions могли быть добавлены, удалены или изменены слишком сильно для автоматического сопоставления.

Когда plugin не смог связать очевидную пару, команда `Set Match` позволяет вручную указать адрес соответствующей функции. Флаг `Propagate` просит продолжить matching на основе новой опорной пары.

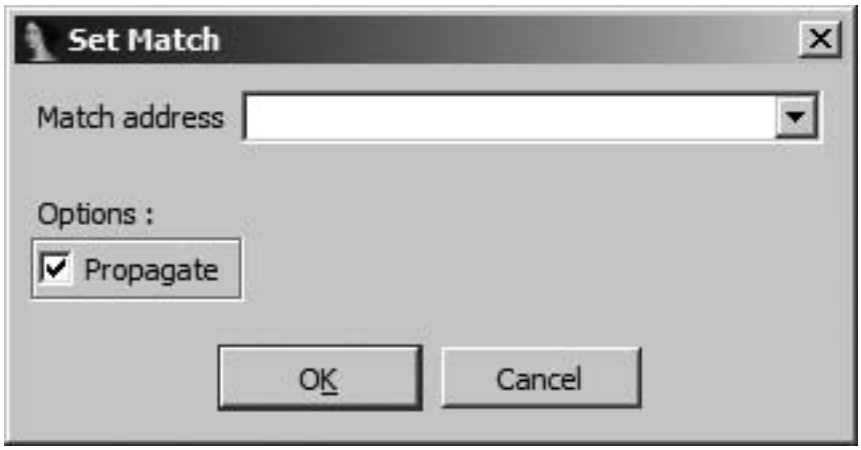

Рис. 22-1. Ручное сопоставление функций с помощью PatchDiff2

Для matched pair команда `Display Graphs` открывает side-by-side control-flow graphs и отмечает цветом blocks, присутствующие только в одной версии.

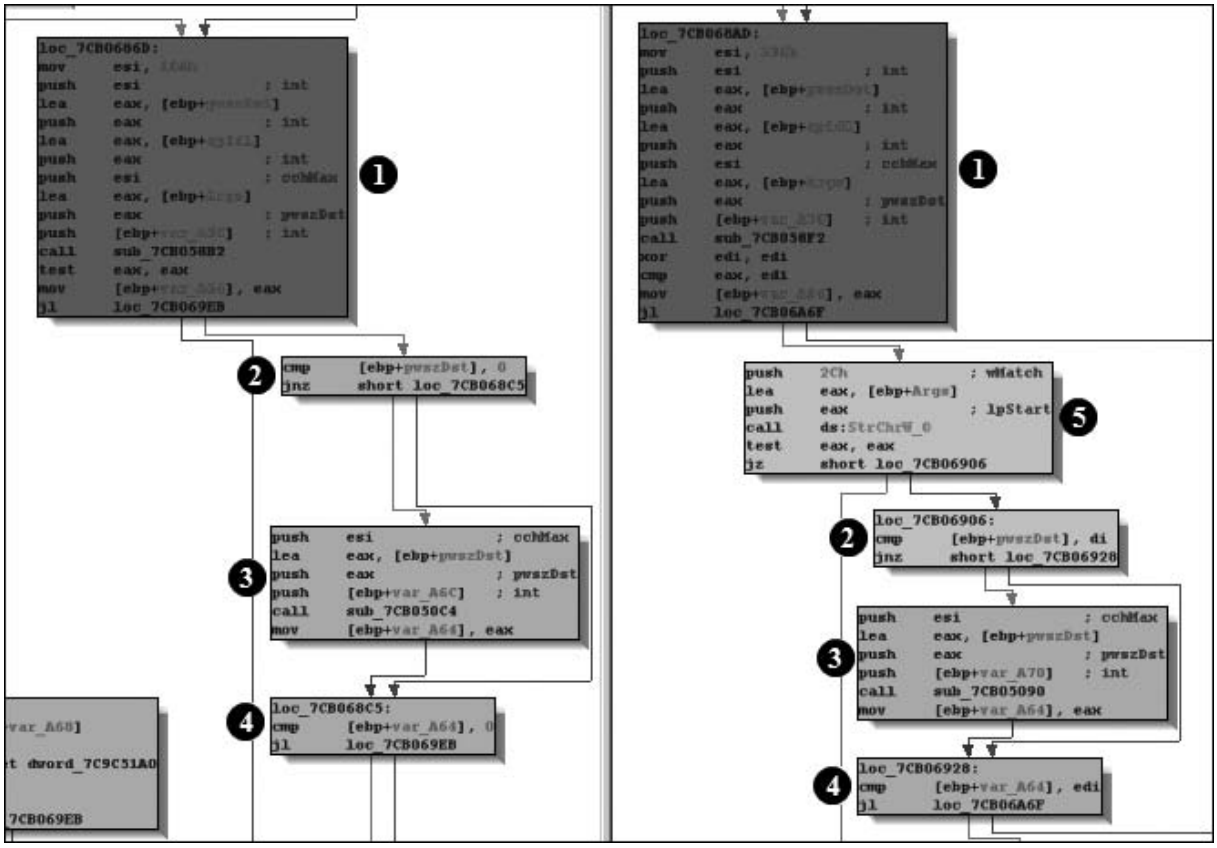

Рис. 22-2. Графическое сравнение функций в PatchDiff2

На рисунке blocks 1-4 присутствуют в обеих функциях, а block 5 добавлен в patched version. Именно его следует проверить первым: это может быть новый bounds check, validation length или безопасный выход. После matched functions нужно изучить unmatched: исправление иногда заменяет функцию целиком или добавляет новый helper.

Diff показывает где изменился код, но не объясняет почему. Аналитик обязан восстановить входные условия, data flow и достижимость нового path, а затем проверить, существует ли безопасный способ подтвердить ошибку в старой версии.

IDA в процессе разработки exploit

После обнаружения уязвимости IDA помогает ответить на практические вопросы:

• сколько байтов отделяет buffer от saved return address;
• какие local variables и saved registers будут перезаписаны раньше;
• где найти `jmp esp`, `pop/pop/ret` или другую подходящую sequence;
• какие writable function pointers, GOT entries или destructors доступны для перезаписи;
• как интерпретировать перехваченный shellcode.

Разбор stack frame

Допустим, frame содержит переполняемый `buffer`, за ним `filedes`, saved `EBP` и return address. Разность offsets сразу дает расстояние до каждой цели. Важно учитывать не только return address: если до него находится socket descriptor и overflow портит его случайным значением, программа может аварийно завершиться раньше передачи управления. Payload должен записать

промежуточным variables осмысленные значения.

Улучшенный Python script может перечислить каждый frame member, apparent size и расстояние до return address. Поля `r` и `s` в frame structure обозначают области saved registers и return address. Такая таблица удобнее визуального подсчета по листингу, но отражает модель IDA; при unusual prologue ее нужно проверить по инструкциям.

Format string vulnerabilities

Рассмотрим код, передающий пользовательский buffer напрямую как format string:

```
char buf[256];
recv(sock, buf, sizeof(buf), 0);
fprintf(stream, buf);
```

Compiler заранее резервирует в frame место для аргументов вызываемых функций. IDA показывает, где окажутся `stream`, pointer на format string и сам `buf`. Если pointer и начало buffer разделяют 16 bytes, первые четыре ожидаемых variadic arguments занимают этот промежуток, а пятый перекрывается первыми четырьмя байтами пользовательского ввода.

Формат вроде `%5$08x` читает fifth argument как hexadecimal. Управляемое значение в начале buffer тем самым становится argument. Последующие positional specifiers обращаются к следующим четырехбайтовым частям. Зная layout, можно строить precise reads/writes через `%n` и выбрать цель в `.got` или `.dtor`. IDA дает расстояния и addresses, но корректность payload зависит от ABI, alignment, architecture и реализации libc.

Поиск команд передачи управления

Если shellcode лежит на stack или heap с непредсказуемым адресом, полезен register, уже указывающий на него. Например, при `ESI` на buffer подойдет `jmp esi`; при payload сразу после overwritten return address часто ищут `jmp esp`.

Искать следует не текст `jmp esp`, а bytes `FF E4`. Последовательность может находиться внутри другой инструкции и не отображаться как самостоятельная команда. Search > Sequence of Bytes находит exact bytes по всей базе. Text search зависит от syntax и spacing и потому ненадежен.

IDC может перебирать все варианты transfer instructions к выбранному register. Сначала ищутся opcodes прямого `jmp/call reg`, затем проверяется, находится ли адрес в executable segment и не содержит ли он запрещенных payload-ом bytes. Внешние инструменты наподобие `msfpescan` решают похожую задачу, но IDA удобна тем, что найденную sequence можно немедленно исследовать в контексте функции и xrefs.

Адрес внутри существующей инструкции допустим для CPU: декодирование начинается с переданного адреса. Поэтому embedded `FF E4` может быть полезным gadget, даже если основной control flow никогда не распознает его как `jmp esp`.

NOP slides

Классический x86 payload начинается длинной цепочкой `90h`. Переход в любую ее точку скользит к shellcode и расширяет допустимый диапазон адресов. Но монотонная последовательность легко распознается IDS. Polymorphic NOP slide использует множество безопасных инструкций и их комбинаций, сохраняющих нужное состояние.

IDA помогает проверить, что каждый возможный entry point проходит через slide без destructive side effects и приходит к decoder. Это особенно важно для multi-byte псевдо-NOP: попадание в середину меняет границы декодирования.

Перезапись GOT

При наличии arbitrary write необязательно атаковать return address. В dynamically linked ELF global offset table хранит resolved addresses library functions. Если записать в entry адрес shellcode, следующий call соответствующей функции передаст управление атакующему.

Адрес можно найти вручную в `.got` либо пройти цепочку xrefs:

```
def find_got_entry(name):
    ext = LocByName(name)
    if ext == BADADDR:
        return BADADDR
    got = DfirstB(ext)
    if got != BADADDR and SegName(got) == ".got":
        Message("GOT entry for %s is at 0x%08x\n" % (name, got))
        return got
    return BADADDR
```

В некоторых базах первый xref приводит к PLT code, а следующий - к GOT data; script должен проверять segment names, а не полагаться на порядок. Цель выбирают с учетом того, будет ли функция вызвана после перезаписи, writable ли таблица при включенном RELRO и не содержит ли адрес запрещенных bytes.

Анализ shellcode

Shellcode встречается в proof-of-concept, Metasploit payloads, memory dumps и network captures. IDA способна его дизассемблировать, если правильно задать processor, bitness и начальный address. Поскольку header отсутствует, shellcode обычно загружается как raw binary.

В packet capture сначала нужно отделить protocol data от executable payload. Для HTTP это может означать пропуск headers и encoding; для бинарного protocol - учет length fields. NOP slide помогает найти начало, но encoded и polymorphic payload намеренно избегает очевидных `90h`.

Последовательность данных нельзя целиком объявлять code. Первые bytes могут быть protocol fields или адресами, которые exploit использует до shellcode. Правильный порядок:

1. Извлечь exact application payload без link/network headers.
2. Найти настоящий entry point, учитывая overwritten addresses и landing point.
3. Загрузить bytes как binary с правильной architecture.
4. Создать code с точки входа и вручную исправить areas data/code.
5. Если есть decoder, понять его границы и после эмуляции или запуска анализировать восстановленный body.

Encoders меняют byte pattern для обхода signature detection и могут быть self-modifying. Тогда статический listing показывает только decoder. Методы главы 21 - scripts, emulation и debugger-assisted unpacking - применимы и здесь.

Итоги

IDA не является волшебным сканером уязвимостей. Она предоставляет точную модель binary: functions, xrefs, stack frames, segments, bytes и control-flow graphs. На этой основе scripts ускоряют аудит опасных API, diff plugins выделяют security patches, а ручной анализ помогает рассчитать layout и найти пригодные transfer sequences или writable targets.

Любой результат остается гипотезой, пока аналитик не восстановит data flow и условия достижения. Наибольшая ценность IDA проявляется не в автоматическом ответе, а в возможности быстро задавать бинарному файлу конкретные проверяемые вопросы.

Глава 23. Реальные плагины IDA

Предыдущие главы показали API и архитектуру plugins. Теперь рассмотрим инструменты, которыми действительно пользуются аналитики. Набор отражает состояние экосистемы на момент выхода книги: версии, цены и ссылки исторические, но принципы интеграции остаются полезными.

Готовый plugin не всегда устанавливается одним копированием файла. Source distribution приходится собирать подходящим compiler и IDA SDK, а makefiles автора могут не учитывать конкретную систему. Binary distribution привязана к platform, разрядности и версии SDK. Plugin, собранный для несовместимой IDA, обычно нельзя использовать до новой сборки. Перед установкой следует проверить архитектуру, GUI variant, processor support и внешние runtime dependencies.

Hex-Rays Decompiler

Hex-Rays - коммерческий decompiler от создателей IDA. Рассматриваемая версия генерирует C-подобный pseudocode для functions из 32-битных x86 и ARM binaries. Plugin распространяется только в binary form и лицензируется отдельно от IDA.

Текущую function декомпилируют через View > Open Subviews > Pseudocode или `F5`. File > Produce File > Create C File, `Ctrl-F5`, создает pseudocode для всего файла. Результат открывается в отдельной вкладке и остается связанным с базой IDA.

Decompiler восстанавливает expressions, loops, conditionals и calls, используя те же type cues, что и IDA. Неизвестным locals присваиваются dummy names наподобие `v1`, `v2`. Результат становится гораздо понятнее после итеративного редактирования типов и имен.

Контекстное меню позволяет переименовать variable, задать type, перейти к xref, добавить comment, отметить участок как уже скомпилированный, скопировать assembly и показать casts.

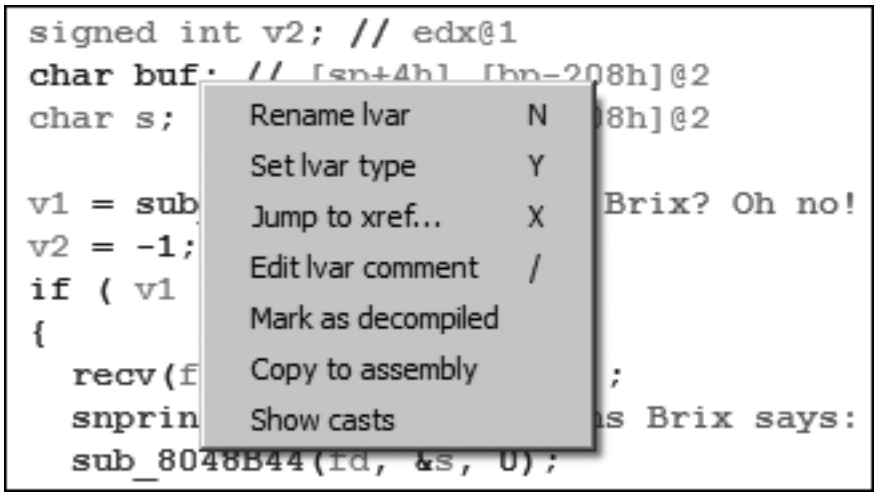

Рис. 23-1. Возможности редактирования pseudocode в Hex-Rays Decompiler

Команды `Rename variable` и `Set lvar type` изменяют не только отображение одной строки: тип распространяется по expressions и способен изменить весь decompilation. Если многочисленные принудительные conversions мешают чтению, `Hide Casts` временно скрывает casts. Имена functions, globals и comments синхронизируются с основной IDB.

Ограничение фундаментально: compiler удаляет comments, многие names, исходные types и высокоуровневые конструкции. Decompiler делает обоснованную реконструкцию, но не возвращает оригинальный source. Для C++ рассматриваемая версия выдает C-представление и не восстанавливает classes в исходной форме. Любой важный вывод из pseudocode следует подтверждать assembly, особенно при unusual optimization, undefined behavior и edge cases.

Иными словами, C читать быстрее, но доверие decompiler не должно заменять reverse engineering. Лучший результат получается в цикле: понять участок, уточнить type/name, повторно декомпилировать и проверить вывод по инструкциям.

IDAPython

IDAPython начинался как сторонний plugin Gergely Erdelyi, а с IDA 5.4 поставлялся вместе с IDA. Он встраивает Python scripting engine и предоставляет wrappers над IDA SDK. Историческая версия главы поддерживала Python 2.x, но не 3.x.

Основные преимущества по сравнению с IDC:

• богатая стандартная библиотека Python;
• classes, modules, exceptions и привычные data structures;
• удобная обработка files, binary formats и network data;
• большое сообщество и возможность переиспользовать сторонние packages;
• интерактивная console для коротких запросов к базе.

Source distribution содержит `BUILDING.txt`; сборка зависит от IDA SDK и Python development files той же разрядности, что IDA. На Windows использовались подходящие версии Visual C++, на Linux требовались соответствующие headers и libraries. Поскольку IDAPython уже входил в поставку IDA, самостоятельно собирать его имело смысл главным образом для новой версии или модификации.

IDL/SDK APIs доступны через modules, а sample scripts демонстрируют enumeration functions, xrefs, segments, operands и patching. Практически любой повторяющийся ручной анализ - кандидат на небольшой Python script.

collabREate

collabREate синхронизирует изменения нескольких удаленных IDA sessions. Он состоит из client plugin и Java server с SQL database. Идея развивает IDA Sync, но предлагает полноценные projects, permissions, subscriptions и checkpoints.

Plugin подписывается на database change notifications и публикует meaningful events: переименование, comments, patched bytes, изменение types и другие действия. Клиент также принимает события сервера и применяет их к локальной IDB. Поэтому edits, сделанные script-ом, распространяются так же, как ручные, если они вызывают те же уведомления.

Publish и subscribe настраиваются отдельно. Один участник может отправлять все изменения, но принимать только names и comments; другой - работать только как observer. Эта модель позволяет совместно использовать разные версии IDA, даже если старый клиент не способен открыть IDB, созданную новой версией: они обмениваются operations, а не целым файлом базы.

При подключении client отправляет MD5 исходного binary. Server группирует projects по этому значению, чтобы участники не смешали анализ разных файлов. Пользователь выбирает существующий project или создает новый с description.

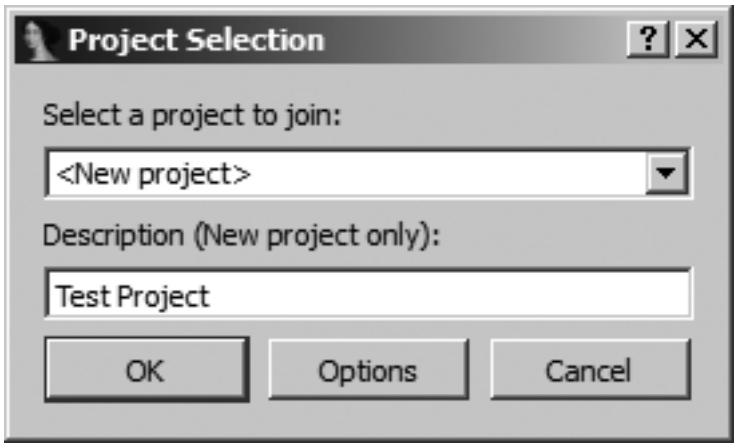

Рис. 23-2. Диалог выбора проекта collabREate

После выбора client сообщает, какие update types публиковать и получать. Server пересылает накопленные и новые события подписчикам.

Project можно fork-нуть перед крупным экспериментом. Server допускает несколько branches для одного binary и следит, чтобы client подключился к нужной. Полного rollback server не делает, но checkpoints фиксируют состояние последовательности updates. Чтобы вернуться к точке, можно открыть локальную IDB, соответствующую checkpoint, и продолжить от нее.

Повторная активация hotkey открывает команды управления: fork, checkpoint, requested permissions, project permissions и disconnect.

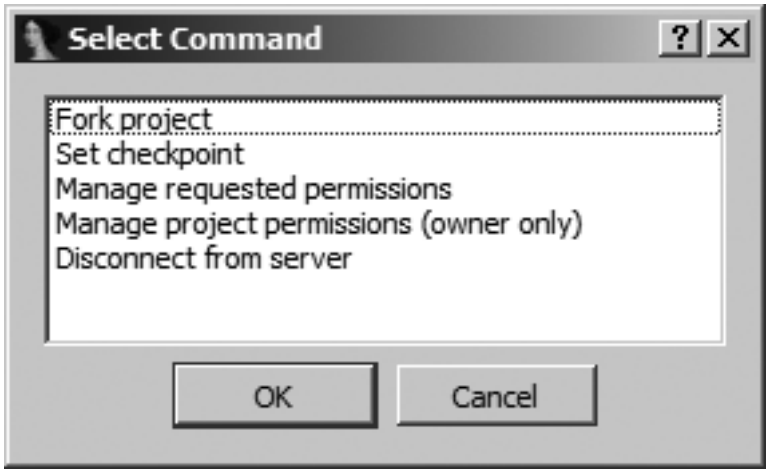

Рис. 23-3. Диалог Select Command в collabREate

Owner может ограничить publish/subscribe permissions пользователей. Это важно, когда широкая группа читает findings, а право патчить bytes или переименовывать symbols есть только у небольшой команды.

ida-x86emu

ida-x86emu - встроенный x86 instruction emulator, подробно разобранный в главе 21. Он эмулирует инструкции непосредственно над IDB, создает stack и heap, моделирует registers и часть OS environment, a self-modifying writes патчит в database.

Историческая distribution включала source для SDK 6.1 и binaries для IDA 5.0 и новее, включая freeware build. Plugin работал с PE, ELF и Mach-O в Qt и Windows GUI variants. Его ключевая

ценность - исследование decoder, packer и потенциально опасного кода без native execution.

Нужно помнить, что это не полный CPU/OS emulator. Неизвестные instructions, syscalls или APIs требуют вмешательства аналитика. Достоверность результата определяется тем, насколько правдоподобно смоделированы входы и окружение.

Class Informer

C++ Class Informer от Sirmabus автоматизирует восстановление информации о classes в binaries, собранных Microsoft Visual Studio. Он ищет vtables/vftables и MSVC RTTI, извлекает class names и inheritance и применяет annotations к базе.

При запуске можно:

• строить type containers;
• находить и именовать static/global constructors и destructors;
• выводить все найденные vftables;
• перезаписывать существующие comments;
• включать подробный debug output и audio notification;
• выбрать CODE и RDATA segments для сканирования.

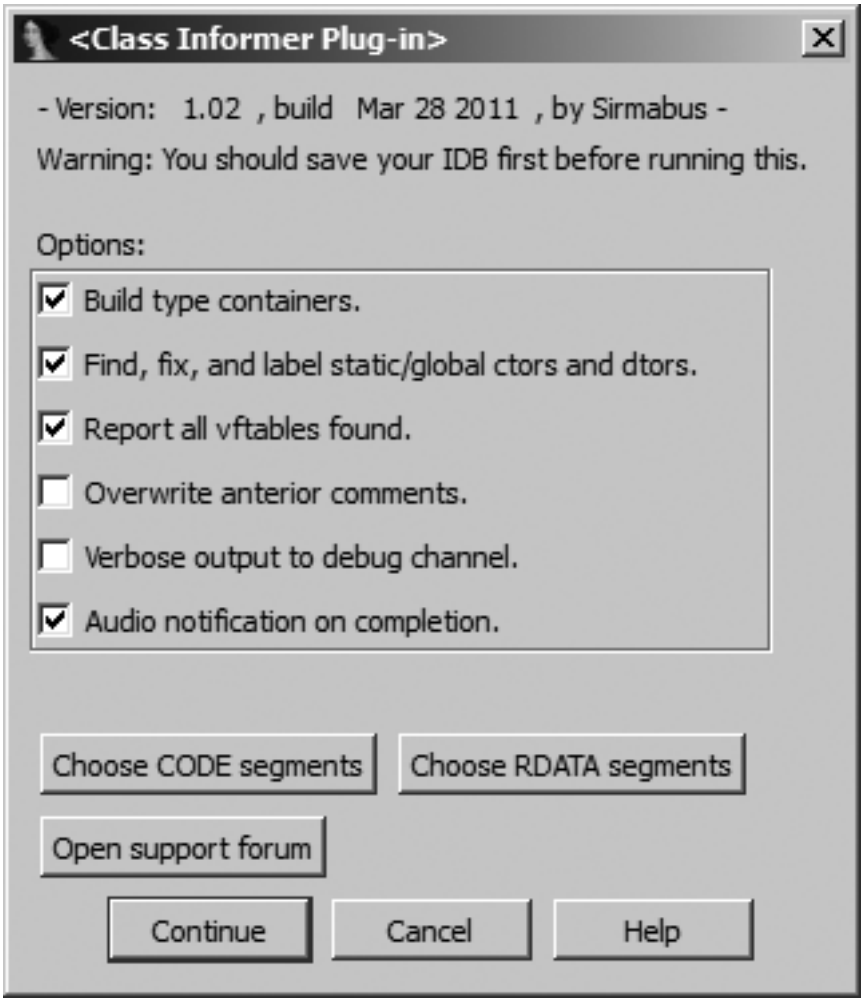

Рис. 23-4. Диалог параметров Class Informer

После `Continue` scan может занять заметное время. Итог открывается в новой tabbed view. Для каждой vtable показываются address, число function pointers и сведения RTTI: class name, base

classes и признаки multiple inheritance. Plugin также присваивает понятные names таблицам и comments в disassembly.

Автоматическое распознавание зависит от compiler ABI и сохранности RTTI. Оно рассчитано именно на MSVC patterns; другой compiler, stripped metadata или obfuscation снижают качество. Найденные связи нужно подтверждать constructors, assignments vptr и calls through table.

MyNav

MyNav - IDAPython extension Joxean Koret, победившее в конкурсе plugins Hex-Rays 2010 года. Script запускают после окончания initial autoanalysis; он добавляет около двадцати команд в Edit > Plugins.

MyNav сочетает navigation, debugger tracing и code coverage. В отличие от обычной instruction-level trace, он умеет строить function-level representation, что удобнее для больших приложений. Результаты одной debugging session можно использовать как filter для следующей.

После session plugin показывает visited и unvisited functions/blocks. Дальнейший анализ можно сузить до кода, который:

• исполнялся при обработке интересующего input;
• не исполнялся в baseline session;
• появился только при определенном action;
• лежит на пути между заданными functions;
• обращается к интересующим strings или APIs.

Например, сначала записывается startup trace, затем trace при открытии специального файла. Вычитание baseline оставляет обработчик формата и резко сокращает область ручного анализа. Возможность комбинировать traces превращает MyNav в практический инструмент differential dynamic analysis.

IdaPdf

PDF является сложным container format и исторически неоднократно использовался для доставки exploits. IdaPdf состоит из loader module и plugin, предназначенных для разбора структуры PDF внутри IDA.

Loader распознает PDF, создает новую database и раскладывает file objects в адресное пространство. После загрузки plugin анализирует cross-references, streams, filters и embedded content. Таблица объектов содержит номер, location, type, offset и size данных, цепочку filters, адрес/размер filtered stream и признак ASCII.

Context menus позволяют перейти к raw или filtered data, извлечь stream, применить filters и исследовать ссылки между objects. Если stream использует Flate, ASCIIHex, ASCII85 и другие стандартные filters, plugin сохраняет и необработанное, и декодированное представление. Это важно для JavaScript, images и embedded executables.

Дополнительные команды Edit > Other позволяют выделенный block преобразовать или исследовать как PDF data. Analyst получает привычные IDA navigation и scripting APIs для формата, который обычно не является executable. Подход показывает, что loader/plugin architecture применима не только к машинному коду, но и к структурированным attack containers.

IdaPdf не определяет автоматически, является ли JavaScript вредоносным, и не заменяет sandbox. Он облегчает извлечение и связь объектов; дальнейший анализ embedded code выполняется соответствующими tools.

Итоги

Экосистема plugins расширяет IDA в разных направлениях: Hex-Rays поднимает представление до pseudocode, IDAPython автоматизирует запросы, collabREate соединяет команду, x86emu моделирует исполнение, Class Informer восстанавливает C++ classes, MyNav фильтрует dynamic traces, а IdaPdf превращает PDF в навигируемую database.

При выборе стороннего plugin следует оценивать совместимость с IDA SDK, platform, bitness, processor и доверие к автору. Plugin выполняется внутри процесса IDA и получает полный доступ к базе и системе, поэтому его исходники и binaries являются частью attack surface. После установки лучший результат дает не пассивное ожидание автоматики, а сочетание вывода plugin с ручной проверкой в disassembly.

Глава 24. Отладчик IDA

IDA известна прежде всего как disassembler, но включает интегрированный debugger. Его главное преимущество - единая database: names, comments, types, functions и graphs, созданные статическим анализом, доступны во время исполнения, а runtime observations можно сразу переносить обратно в IDB.

Глава предполагает знакомство с обычными понятиями debugging: process, thread, breakpoint, registers, step и call stack. Основное внимание уделяется тому, как эти действия представлены именно в IDA.

Запуск отладчика

IDA может присоединиться к существующему process или запустить новый под контролем debugger. Доступные backends зависят от platform и installation: local Windows, local Bochs, remote Windows/Linux/Mac OS X, GDB, ARM/Android, Symbian и WinCE.

Если никакая database не открыта, меню Debugger > Attach предлагает выбрать backend напрямую.

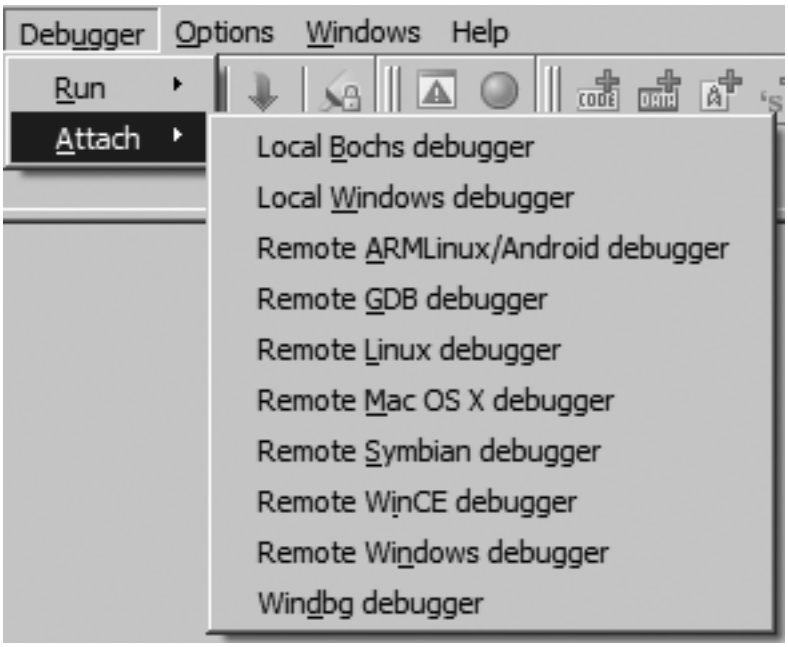

Рис. 24-1. Выбор backend для присоединения к произвольному процессу

После выбора локального debugger IDA запрашивает running process. Список содержит PID и имя, поддерживает поиск и прокрутку.

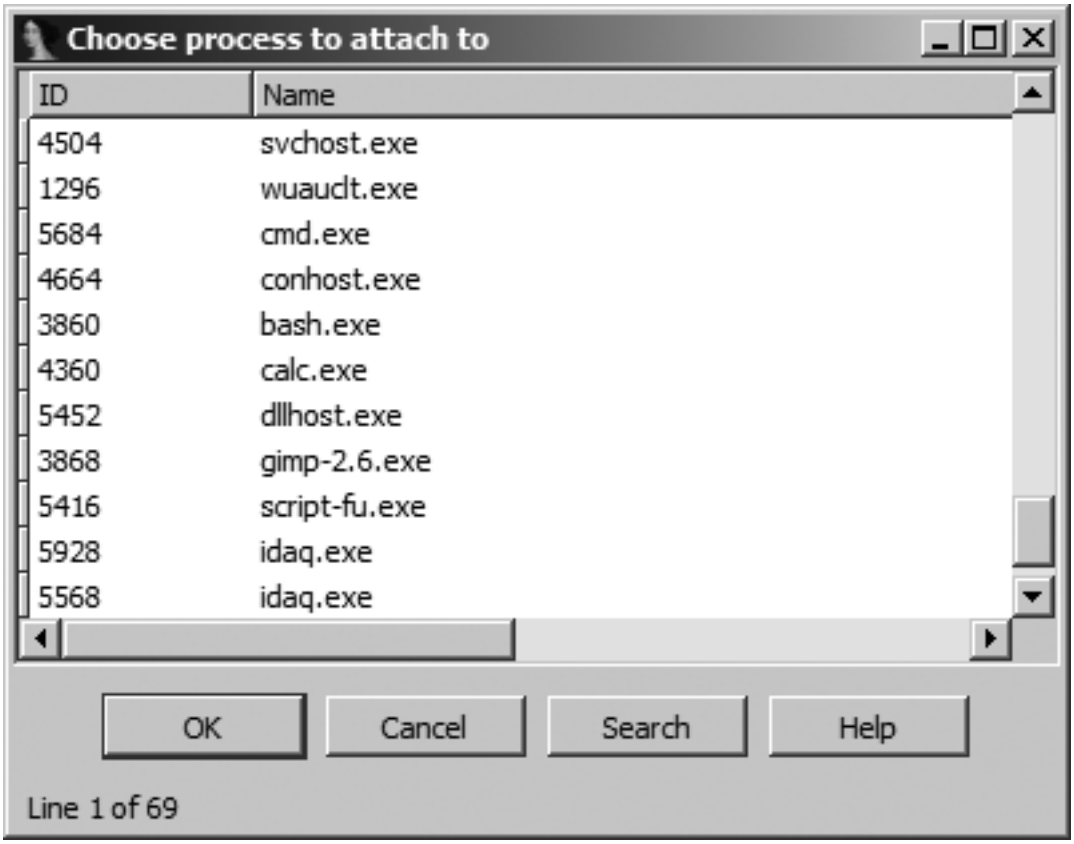

Рис. 24-2. Диалог выбора процесса debugger

При attach без заранее открытого executable создается temporary database по memory image. В нее входят segments всех загруженных shared libraries, поэтому она крупнее и шумнее обычной IDB. Поскольку до attach не было полного file image и autoanalysis, IDA первоначально разбирает лишь instruction по текущему IP и достижимый код. Process сразу приостанавливается, что позволяет поставить breakpoints перед продолжением.

Чаще удобнее сначала открыть executable и дождаться autoanalysis. Тогда меню Debugger содержит полный набор operations.

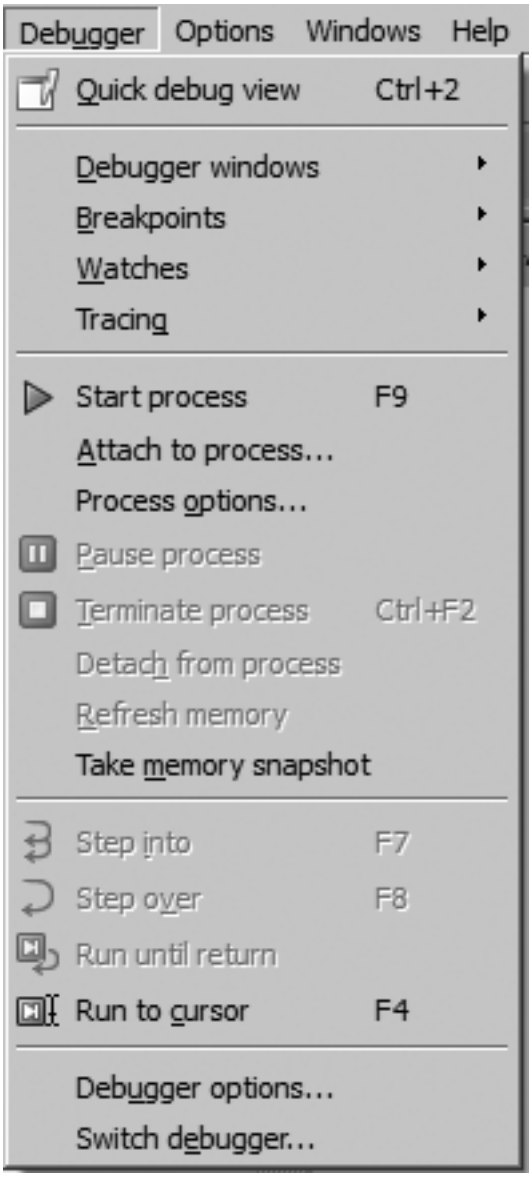

Рис. 24-3. Меню Debugger для открытого executable

Если backend еще не назначен, Debugger > Switch Debugger открывает selection dialog. Опция `Set as default debugger` запоминает выбор для новых databases данного типа.

Рис. 24-4. Диалог выбора отладчика

Debugger > Process Options задает executable path, input file, command-line arguments, working directory, hostname/port remote server и другие параметры. Значения path в IDB и на target могут отличаться.

Attach и Start Process

Attach позволяет наблюдать только с текущего момента. Loader initialization, ранние constructors и уже выполненный startup недоступны. Start Process, `F9`, запускает executable с самого начала и дает поставить breakpoint до критической sequence. Для packed/obfuscated code это особенно важно: можно остановиться сразу после unpacking и до передачи управления основной логике.

При запуске из готовой IDB статический analysis уже описал executable content. Shared libraries, загруженные позже, обычно имеют более ограниченный analysis, поскольку debugger видит memory mapping, а не всегда полный исходный file.

IDA не эмулирует чужую architecture. Local backend исполняет binary на совместимом host. Для другой OS/CPU нужен remote debugger server или GDB-compatible target. IDA поставляется с несколькими servers; client передает команды по сети, а actual process работает на target. Для WinCE исторически использовался ActiveSync, для Unix-like targets - `gdbserver`.

Remote debugging требует аккуратной проверки paths и доверия. Команда Start Process действительно запускает original executable на debugging host. Если исследуется malware, сам факт открытия IDB не создает sandbox: target должен быть изолирован, а network и shared folders - ограничены.

Основные окна debugger

После запуска IDA переключается на debugger desktop.

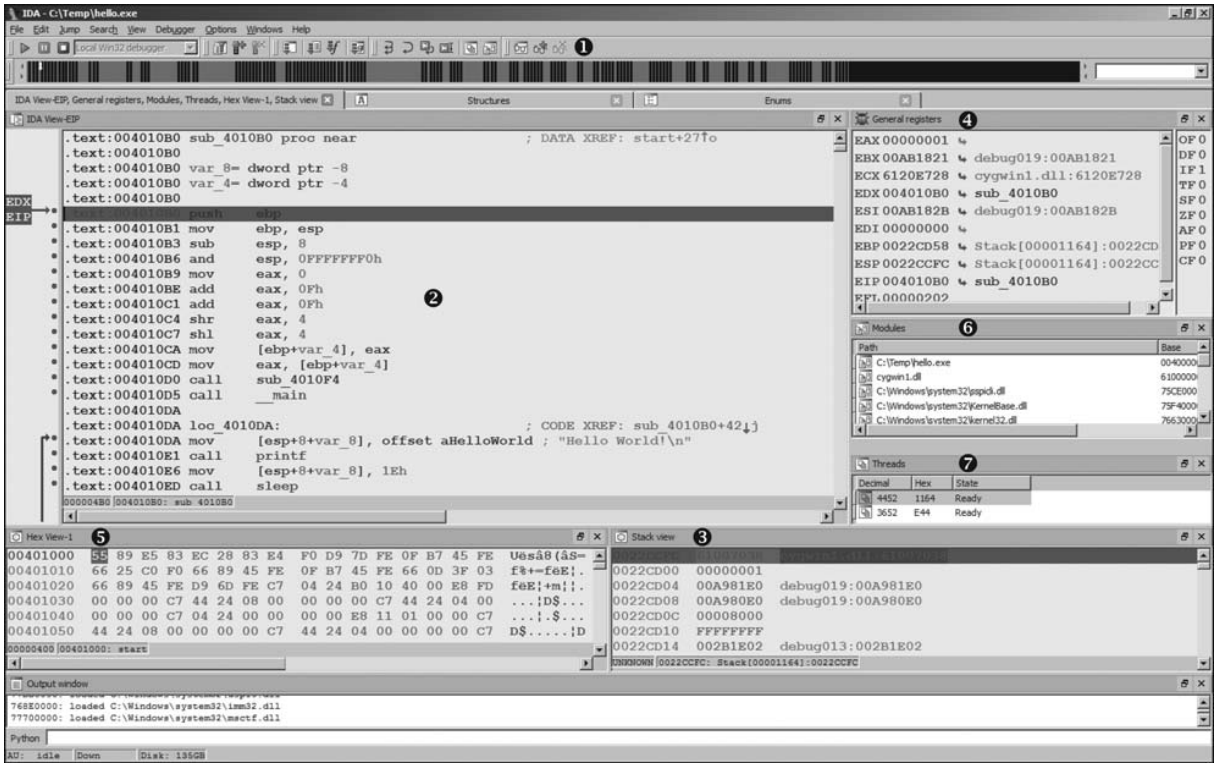

Рис. 24-5. Стандартное рабочее пространство debugger IDA

Основные элементы:

1. Debugger toolbar с resume, pause, stop, step и breakpoints.
2. `IDA View-EIP` с disassembly у текущего instruction pointer.
3. `Stack view` с содержимым stack.
4. `General registers` и flags.
5. `Hex View` для raw memory.
6. `Modules` со всеми loaded images.
7. `Threads` со списком process threads.

Desktop можно перестраивать, менять fonts и сохранять через Windows > Save Desktop. В отличие от отдельных debuggers, IDA использует те же disassembly engine и annotations, что и статическая база.

IDA View-EIP и Stack View

Текущий `EIP` отмечен в margin. Если другой register указывает на видимый address, его имя также появляется слева. При step окно следует за IP, но обычные navigation commands остаются доступными.

Stack View является специализированным disassembly view. IDA форматирует DWORDs, разрешает pointers в function names и показывает probable return addresses. Double-click переходит к цели. Формат можно менять как для обычных data items.

Registers

General Registers показывает integer registers и flags. Значения, изменившиеся после последнего event, выделяются. Arrow рядом с pointer-like value позволяет перейти к memory location.

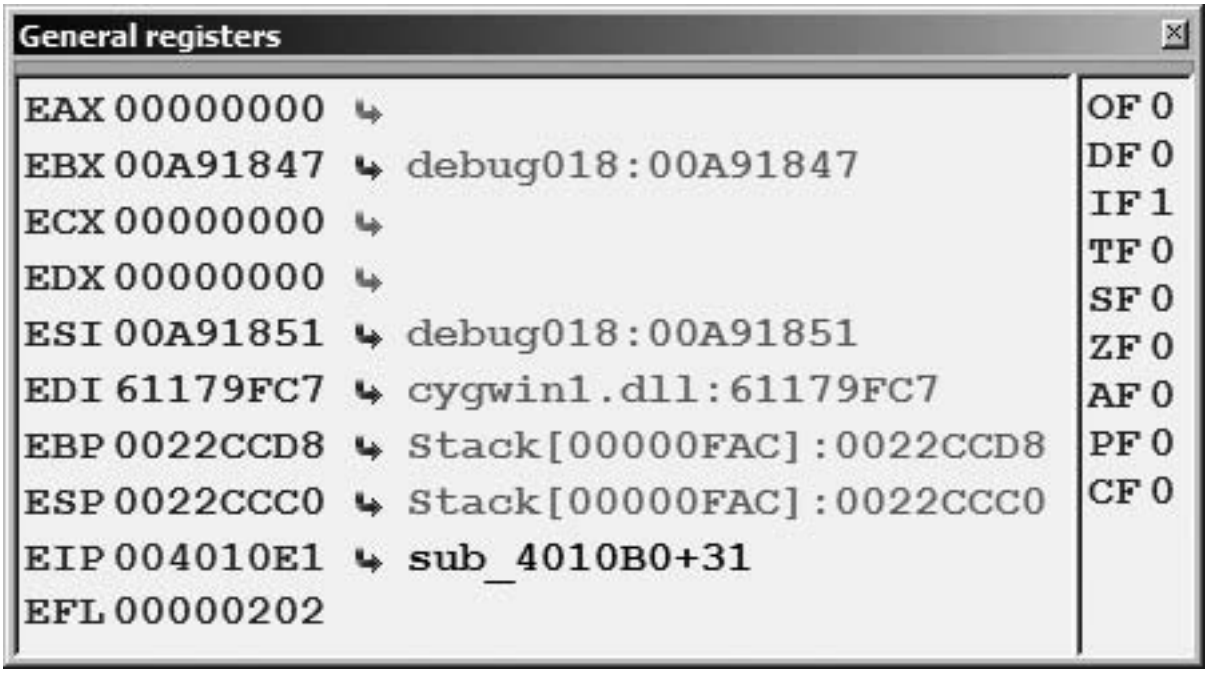

Рис. 24-6. Окно General Registers

Register можно изменить через context menu или double-click. Это позволяет моделировать branch outcome, подменять return value и исправлять state после пропущенного call. Однако ручная правка должна учитывать flags и связанные значения: изменение только `EAX` не всегда эквивалентно выполнению нужной instruction sequence.

Modules и Threads

Modules перечисляет main image и shared libraries с base addresses. Double-click module открывает exported symbols.

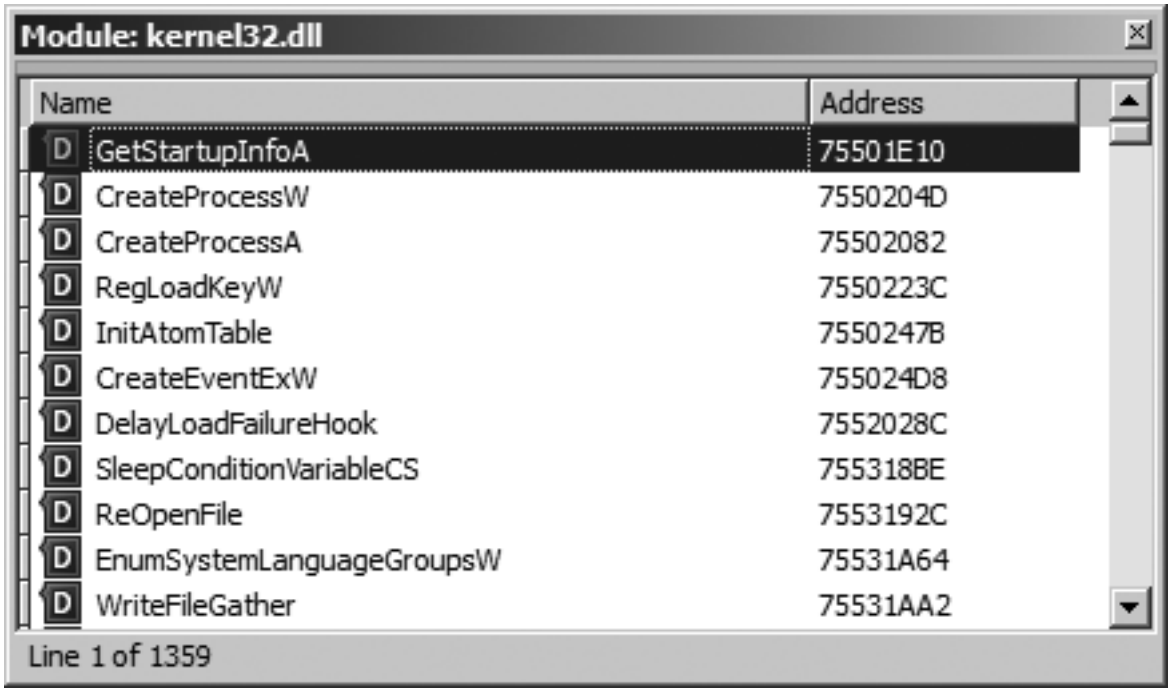

Рис. 24-7. Окно Modules и список exports выбранного модуля

Это быстрый способ найти runtime address API при ASLR. Threads показывает identifier, state и текущий IP. Выбор thread меняет register context и disassembly; context menu позволяет перейти к instruction выбранного потока. При multi-threaded debugging всегда проверяйте, какой thread остановил process и какой сейчас выбран.

Управление процессом

Toolbar объединяет наиболее частые команды.

Рис. 24-8. Инструменты управления debugger process

• `Start/Continue`, `F9`, запускает или продолжает process.
• `Pause` асинхронно останавливает выполнение.
• `Terminate` завершает process.
• `Step Into`, `F7`, выполняет одну instruction с входом в call.
• `Step Over`, `F8`, выполняет call как одну операцию, временно останавливаясь после него.
• `Run Until Return` продолжает до возврата текущей function.
• `Run to Cursor`, `F4`, выполняет до выбранного address.

Step Over и Run Until Return реализуются временными breakpoints и зависят от корректного control flow. Exception, thread switch или altered return address могут привести к остановке в неожиданном месте.

Memory view не обязано автоматически отражать external changes, особенно при remote debugging. `Refresh debugger memory` перечитывает данные target. `Take memory snapshot` переносит runtime bytes в database snapshot, что полезно после unpacking, но увеличивает IDB и не создает полноценный executable file.

Breakpoints

`F2` ставит или снимает breakpoint по позиции cursor. Адрес подсвечивается красной полосой. Полный список открывается через Debugger > Breakpoints > Breakpoint List.

По умолчанию IDA использует software breakpoint: первый opcode byte заменяется инструкцией trap, например x86 `int 3`. При срабатывании debugger восстанавливает byte, корректирует IP и управляет возобновлением. Проверка checksum или чтение собственного кода может обнаружить такую подмену.

Hardware breakpoints на x86 используют debug registers и не меняют code bytes. Они умеют останавливаться при execute, write или read/write access, но число registers мало, alignment и допустимые sizes ограничены CPU. Каждый thread имеет собственный register context, что debugger должен учитывать.

Context menu существующего breakpoint > Edit Breakpoint открывает настройки.

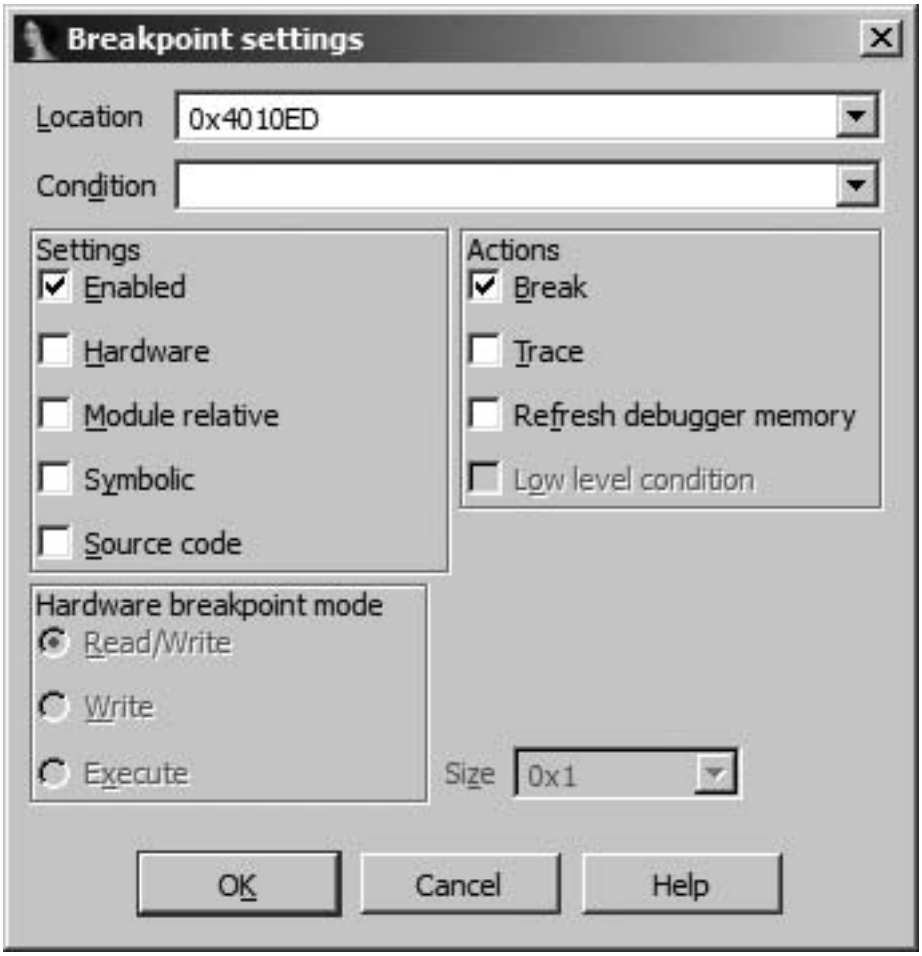

Рис. 24-9. Диалог Breakpoint Settings

Здесь задаются:

• enabled/disabled state;
• hardware или software implementation;
• module-relative и symbolic location;
• source-code binding;
• condition;
• actions `Break`, `Trace`, `Refresh debugger memory`;
• low-level condition, если backend ее поддерживает;
• access mode и size hardware breakpoint.

Module-relative breakpoint переживает relocation: location задается относительно module base. Symbolic breakpoint разрешается по имени после загрузки module. Это предпочтительнее hardcoded runtime address при ASLR.

Условные точки останова

Condition записывается выражением IDC, не Python. Ноль означает false, ненулевое значение - true. Доступны special register variables: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`, `EIP`, `EFL`, частичные registers и отдельные flags.

Примеры:

```
EAX == 0
ESP < 0x00120000
```

```
Byte(ESI) == 0x4D && Byte(ESI + 1) == 0x5A
strstr(GetString(ECX, -1, ASCSTR_C), "password") != -1
```

IDC functions можно вызывать, если они возвращают value и безопасно читают process state. Assignment также допустим и позволяет изменить поведение. Например, условие может присвоить `EAX=0` и вернуть false, фактически подменяя return value без остановки. Такой прием мощный, но side effects в condition затрудняют воспроизводимость.

Low-level condition вычисляется target/backend и работает быстрее, потому что не требует полной остановки и round trip к IDA. Набор поддерживаемых expressions ограничен.

Actions разделяют stop и logging. Breakpoint может не прерывать процесс, а только записывать trace event или обновлять memory. Это удобно для частых sites, но большой поток events существенно замедляет программу.

Tracing

Tracing записывает sequence events без ручного step. IDA поддерживает instruction tracing, function tracing и breakpoint trace actions. Настройки открываются через Debugger > Tracing > Tracing Options.

Рис. 24-10. Диалог Tracing Options

Параметры включают buffer size, trace file, output window, объединение consecutive events с одинаковым IP, проход через debugger segments и library functions, логирование instruction с тем же IP и return instructions.

Instruction trace наиболее подробный и дорогой: каждая instruction создает event с thread, address, instruction и result. Function trace записывает calls/returns и дает компактную картину. Исключение

libraries уменьшает шум, но может скрыть важную callback или API behavior.

Trace buffer имеет конечный размер и обычно работает как кольцо: старые events вытесняются. Для долгого запуска нужен file или узкий filter. Trace window позволяет переходить к address каждого event и анализировать значения.

Call stack

Debugger > Debugger Windows > Call Stack показывает цепочку active calls.

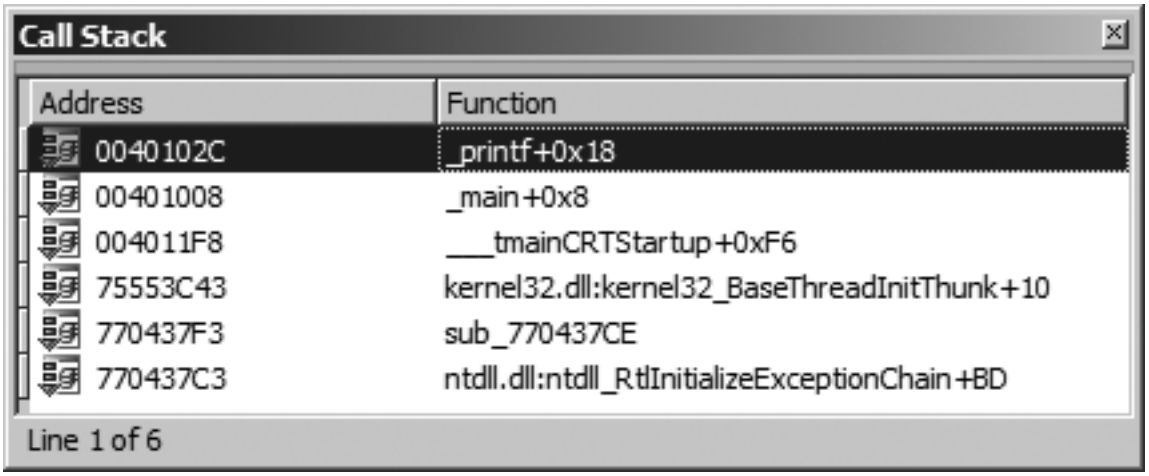

Рис. 24-11. Пример Call Stack

Верхняя строка - текущая function, ниже callers вплоть до thread startup. IDA идет по stack frames, используя saved frame pointer и return address. Если compiler опустил `EBP`, применил tail call, повредил stack либо unwind metadata недоступна, цепочка может быть неполной или неверной. Поэтому подозрительный frame проверяют по stack bytes и call instructions.

Double-click frame переносит disassembly к соответствующему return site. Это помогает понять контекст crash и быстро перейти от library routine к user code.

Watches

Watch постоянно вычисляет expression при каждой остановке. Это удобнее ручного перехода к variable. Globals имеют fixed database address, а local variables существуют только внутри active frame и зависят от `ESP`/`EBP`.

После входа в function IDA может вычислить runtime address local по frame model. При наведении на operand появляется resolved expression и фактический stack address.

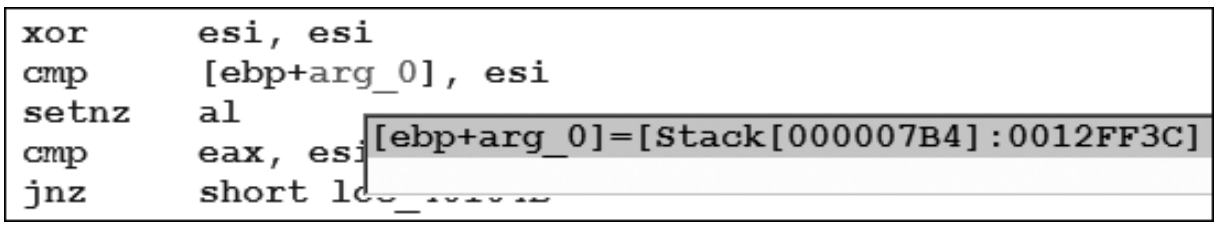

Рис. 24-12. Debugger вычисляет адрес local variable

Watch добавляется через Debugger > Watches. Выражение может читать registers, memory и IDC functions. Недоступный local после выхода из function уже не имеет прежнего смысла; сохраненный absolute address может указывать на переиспользованный stack slot.

Автоматизация debugger через IDC

Scripting API позволяет управлять registers и breakpoints:

```
SetRegValue(value, "EAX");
AddBpt(address);
AddBptEx(address, size, type);
DelBpt(address);
GetBptQty();
SetBptAttr(address, BPTATTR_FLAGS, value);
SetBptCnd(address, "EAX == 0");
CheckBpt(address);
```

Custom IDC breakpoint handler можно связать с address. Handler читает event context, логирует arguments, меняет state и возвращает решение об остановке. Для частых вызовов это превращает IDA в легкий instrumentation framework.

Синхронная модель событий

Команды `StepInto`, `StepOver`, `RunTo` и аналогичные только инициируют execution request. Перед следующим request нужно получить и очистить event через `GetDebuggerEvent`:

```
StepInto();
GetDebuggerEvent(WFNE_SUSP, -1);
StepInto();
GetDebuggerEvent(WFNE_SUSP, -1);
```

Если дважды вызвать step без получения event, второй request может не исполниться, потому что предыдущий event остается pending. Семейство `GetEventXXX()` возвращает PID, TID, address, exception и прочие сведения о последнем event.

Для resume используется request/continue API, а не бесконтрольный loop. Scripts также могут launch, attach, detach и terminate process, но должны проверять return codes и event type: breakpoint, exception, process exit и library load требуют разных действий.

Сбор статистики инструкций

Script может step-нуть процесс в loop и вести hash table по mnemonic:

```
static count_instruction(table) {
  auto mnem, old;
  mnem = GetMnem(EIP);
  old = GetHashLong(table, mnem);
  SetHashLong(table, mnem, old + 1);
}
```

После каждого `StepInto` вызывается `GetDebuggerEvent`, затем увеличивается counter для `mov`, `push`, `call`, `jmp` и других mnemonics. Это учебный пример: single stepping крайне медленный, thread scheduling влияет на выборку, а self-modifying code требует refresh. Для production instrumentation лучше tracing или специализированный plugin.

Автоматизация через plugins

SDK предоставляет asynchronous request queue и debugger notifications. Plugin может подписаться на events, включить tracing, собрать статистику текущей function и показать результат после достижения return.

Главное правило асинхронной модели: callback notification не должен вызывать synchronous debugger function, которая ожидает нового event. Пока IDA обрабатывает текущий event, такое ожидание может deadlock. Вместо этого нужно ставить asynchronous request в очередь; IDA выполнит его после возврата callback.

Типичный plugin:

1. Регистрирует notification hook.
2. Создает netnode или другую структуру counters.
3. Включает trace и ставит временную stop condition на выходе function.
4. На каждом trace event получает address из arguments и обновляет counter.
5. На stop event отключает tracing через queued request.
6. Перебирает netnode, выводит статистику и удаляет temporary state.

Имена SDK debugger functions не всегда совпадают с IDC names. При переносе script в C++ нужно сверяться с headers текущего SDK, учитывать synchronous/asynchronous variants и lifetime event arguments.

Итоги

Debugger IDA соединяет runtime state со статической IDB. Attach удобен для уже работающего process, Start Process - для полного наблюдения startup. Registers, stack, modules, threads, call stack и watches дают стандартные возможности, а names, types и xrefs делают их контекст богаче.

Breakpoints могут быть software, hardware, conditional, symbolic и trace-only. Tracing и scripting автоматизируют повторяющиеся наблюдения, но требуют строгой event discipline. И, наконец, debugger не является sandbox: local или remote backend исполняет настоящий binary на target, поэтому потенциально опасный код необходимо запускать только в специально подготовленной изолированной среде.

Глава 25. Интеграция дизассемблера и отладчика

Главная сила debugger IDA проявляется не в отдельной команде step, а в тесной связи с IDB. Static analysis дает names, functions и xrefs до запуска; dynamic analysis показывает настоящие bytes, addresses и paths; результаты можно переносить между этими представлениями. Это особенно полезно для packed code, runtime imports, anti-debugging и exception-driven control flow.

Предпосылки

Исполняемый file image и process memory image не идентичны. Loader создает sections, применяет relocations, разрешает imports, выделяет zero-filled regions и загружает libraries. Process также создает stack, heap, PEB/TEB и другие runtime objects. Самомодифицирующийся код меняет bytes уже после загрузки.

IDA database первоначально описывает file image в выбранной virtual layout. Debugger читает process memory. Если program загружена по preferred base и не менялась, addresses совпадают. При ASLR или manual mapping нужен relocation delta. IDA обычно сопоставляет modules автоматически, но analyst должен понимать, относится address к database, module-relative offset или runtime memory.

Базы IDA и debugger

При debugging IDA добавляет segments, соответствующие stack, heap, libraries и другим mappings. Не все из них должны навсегда становиться частью IDB. Debugger segment является runtime view; loader segment происходит из исходного file и участвует в сохранении/rebasing.

Memory snapshots

Debugger > Take Memory Snapshot копирует bytes из process в database. IDA спрашивает, сохранять все segments или только loader segments.

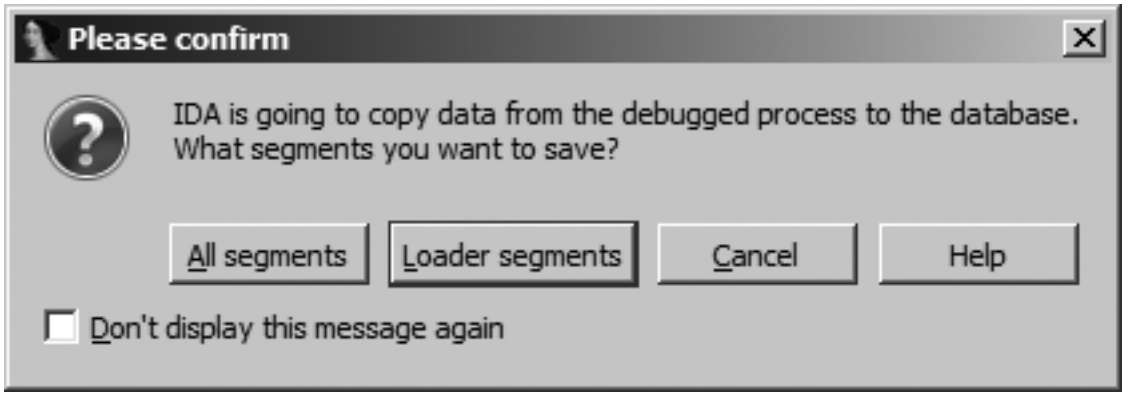

Рис. 25-1. Подтверждение копирования process memory в IDB

`Loader segments` обычно достаточно для распакованного main image: runtime stack и системные libraries не раздувают IDB. `All segments` полезно, когда анализ зависит от dynamically allocated code, heap payload или manual-mapped module. Snapshot не замораживает process и не создает

корректный PE/ELF на disk; он лишь сохраняет текущее memory content в database.

После snapshot code и data, восстановленные packer, остаются доступны после завершения session. Следует повторно запустить analysis на новых bytes и проверить границы segments.

Атрибут loader segment

Edit > Segments > Edit Segment открывает attributes. Флаг `Loader segment` определяет, считать ли segment частью исходного loaded image.

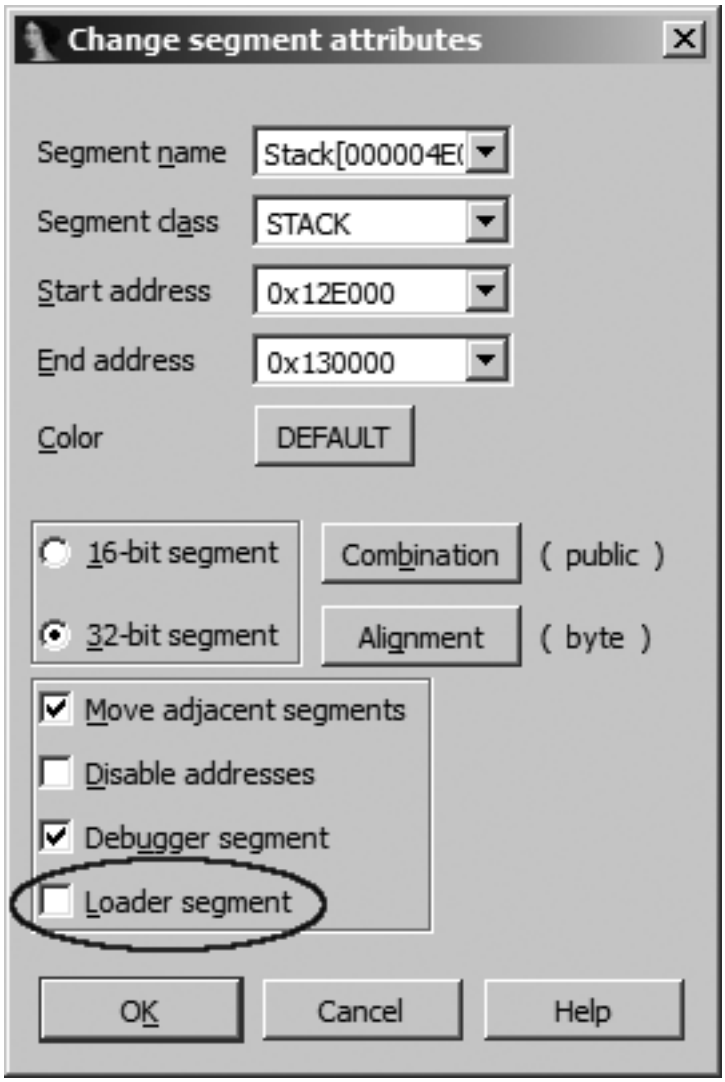

Рис. 25-2. Редактирование segment и флага Loader segment

Для snapshot heap с unpacked code иногда имеет смысл включить loader status, чтобы content сохранялся как существенная часть IDB. Для временного stack - обычно нет. Ошибочная классификация влияет на snapshot selection и дальнейшие database operations.

Отладка обфусцированного кода

Dynamic deobfuscation позволяет заставить сам decoder выполнить сложную работу. Задача analyst - получить контроль до защитного кода, определить момент восстановления original program,

сохранить memory и привести import/name information в пригодный вид.

Самая ранняя остановка

Breakpoint только на PE entry point не всегда является первым. Windows loader может выполнить TLS callbacks до `AddressOfEntryPoint`. Malware использует их для anti-debug checks, распаковки и изменения process state.

IDA parser показывает callbacks в Exports view рядом с `start`.

Рис. 25-3. Окно Exports с обнаруженной TLS callback function

Нужно поставить breakpoint на каждую TLS callback и entry point. Также Debugger Setup позволяет автоматически останавливаться при debugging start, process entry, thread start/exit, library load/unload и debugging message.

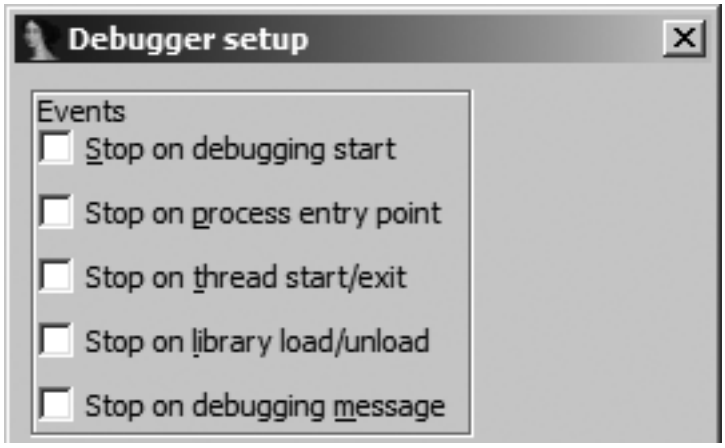

Рис. 25-4. События, при которых debugger может приостановить process

Stop on debugging start дает самый ранний контроль, но может остановить внутри system loader. Stop on process entry удобнее, но пропускает TLS callbacks. Library events полезны, если packer загружает API динамически.

Поиск конца распаковщика

Простейший UPX stub начинается `pusha`, распаковывает sections, восстанавливает imports, заканчивается `popa` и прыгает на OEP. Можно вручную найти `popa` и выполнить Run to Cursor. Но другие packers не сохраняют registers таким образом, а opcode может появляться в data.

IDC script способен просканировать code от entry point до первого подходящего `popa`, поставить temporary breakpoint и продолжить:

```
static run_to_popa(start, end) {
  auto ea;
```

```
    ea = FindBinary(start, SEARCH_DOWN, "61");
    while (ea != BADADDR && ea < end) {
      if (GetMnem(ea) == "popa") {
        AddBpt(ea);
        RunTo(ea);
        GetDebuggerEvent(WFNE_SUSP, -1);
        return ea;
      }
      ea = FindBinary(ea + 1, SEARCH_DOWN, "61");
    }
    return BADADDR;
  }
```

Поиск bytes сочетается с проверкой mnemonic, но не доказывает, что site достижим и действительно завершает decoder. Надежнее наблюдать изменение executable memory и переход из stub section в ранее пустой region.

Step tracing до выхода в новый код

Более общий script включает instruction trace и шагает, пока `EIP` не покинет диапазон, который IDA считала loader/obfuscated code. Этот подход предполагает, что до entry point нет malicious behavior и весь код до границы принадлежит stub. Для hostile sample такие assumptions нужно явно проверить.

При попадании IP в bytes, ранее определенные как data или как середина instruction, IDA предупреждает и предлагает создать instruction по текущему адресу.

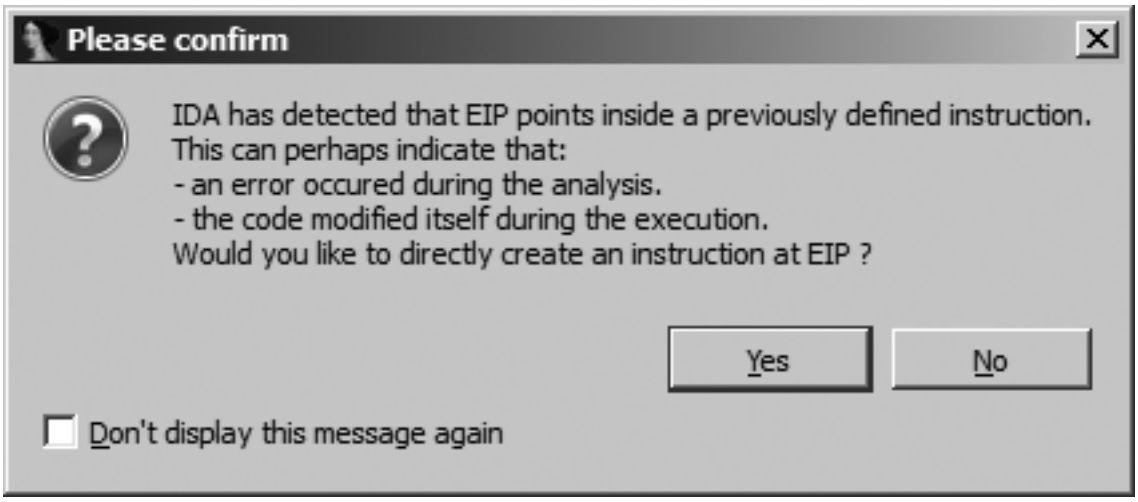

Рис. 25-5. IDA обнаружила EIP внутри ранее определенной инструкции

Для self-modifying и intentionally desynchronized code это ожидаемо. Выбор `Yes` создает instruction по runtime IP, но может undefine перекрывающийся item. После session следует проверить получившуюся database: автоматическое доверие каждому visited IP способно разрушить legitimate alternate representation или data.

Снимок распакованного образа

Когда control transfer достиг OEP:

1. Pause process до основной malicious behavior.
2. Refresh debugger memory.
3. Take snapshot loader segments; при manual allocation включить нужный segment отдельно.
4. Undefine ошибочный packed listing и создать code с OEP.

5. Запустить analysis по восстановленным ranges.
6. Исправить segment permissions и boundaries при необходимости.
7. Восстановить imports и symbols.

Snapshot сохраняет memory form. Чтобы получить запускаемый disk file, нужно дополнительно восстановить headers, raw offsets, section sizes, import directory и entry point специализированным dumper. Для продолжения анализа внутри IDA это не всегда требуется.

Восстановление import table

Packer часто строит IAT вызовами `LoadLibrary`/`GetProcAddress`. После распаковки cells содержат absolute addresses, но names потеряны. Их можно восстановить, перечислив loaded modules и exports, построив map `runtime address -> name`, а затем пройдя probable IAT range.

Псевдокод процесса:

```
exports = {}
for module in debugger_modules():
    for name, address in module_exports(module):
        exports[address] = name

for ea in range(iat_start, iat_end, 4):
    target = Dword(ea)
    if target in exports:
        MakeNameEx(ea, exports[target], SN_NOWARN)
```

Нужно учитывать forwarded exports, duplicate names, ordinal-only exports, ASLR и 64-bit pointer size. Имена с underscore иногда избегаются, поскольку берутся непосредственно из export table и должны оставаться допустимыми IDA identifiers.

После именования IAT изменения распространяются на call sites:

```
UPX0:00403C5B  call ds:RtlFreeHeap
UPX0:00403C61  test eax, eax
UPX0:00403C6C  call ds:RtlGetLastWin32Error
```

Теперь type libraries могут добавить prototypes и parameter comments. Это один из самых заметных способов повысить качество распакованного disassembly.

Сокрытие debugger

Защитный код проверяет PEB flags, heap behavior, system queries, timing и exception semantics. Вместо патчинга десятков sites можно в начале session поставить conditional breakpoints на relevant APIs и автоматически подменять результаты.

PEB и heap flags

`IsDebuggerPresent` читает `PEB.BeingDebugged`. Conditional breakpoint на return может установить `EAX=0`. Еще лучше до checks обнулить field непосредственно в process memory.

`NtGlobalFlag` в PEB часто содержит debug heap flags при запуске под debugger. Его можно очистить. Heap structures также имеют `Flags` и `ForceFlags`, отличающиеся в debug process. Изменять их следует до создания важных heaps; неправильное сочетание способно повредить allocator.

На современных systems offsets различаются по OS version и architecture. Нельзя без проверки применять constants из старого Windows example.

NtQueryInformationProcess

Эта native API имеет вид:

```
NTSTATUS WINAPI NtQueryInformationProcess(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength
);
```

При class `ProcessDebugPort` со значением 7 output может указать debugger port. Другие classes раскрывают debug object или flags. Breakpoint на function entry проверяет `ProcessInformationClass`; если query интересует debugger, handler записывает безопасное значение в output buffer, устанавливает success в `EAX`, снимает arguments по calling convention и меняет `EIP` на saved return address. Function body тем самым не выполняется.

При таком bypass особенно важно правильно очистить stack. Ошибка в числе arguments или convention приводит к crash позже, далеко от breakpoint.

NtSetInformationThread

Значение `ThreadHideFromDebugger` в `ThreadInformationClass` просит kernel скрыть thread от debugger. Последствия зависят от version системы и backend. Перехват строится аналогично: при нужном class вернуть success, не выполняя system call.

```
if (Dword(ESP + 8) == ThreadHideFromDebugger) {
    EAX = 0;
    EIP = Dword(ESP);
    ESP = ESP + 0x10;  // return address и три arguments, пример для stdcall
}
```

Конкретный stack adjustment нужно вывести из prototype, а не копировать вслепую.

OutputDebugString

Исторически `OutputDebugStringA` могла оставлять в `EAX` разные значения в зависимости от debugger. Хотя prototype возвращает `void`, caller способен проверить residual register value. Breakpoint на return нормализует `EAX`. Это напоминание: anti-debugging использует не только документированный API contract, но и наблюдаемые side effects.

IDAStealth

IDAStealth автоматизирует множество Windows-specific обходов: подменяет результаты `NtQueryObject`, `RtlGetNtGlobalFlags`, `NtQuerySystemInformation`, `NtQueryInformationProcess`, `GetTickCount`, version APIs и `NtClose`; исправляет PEB, heap flags и exception dispatch behavior.

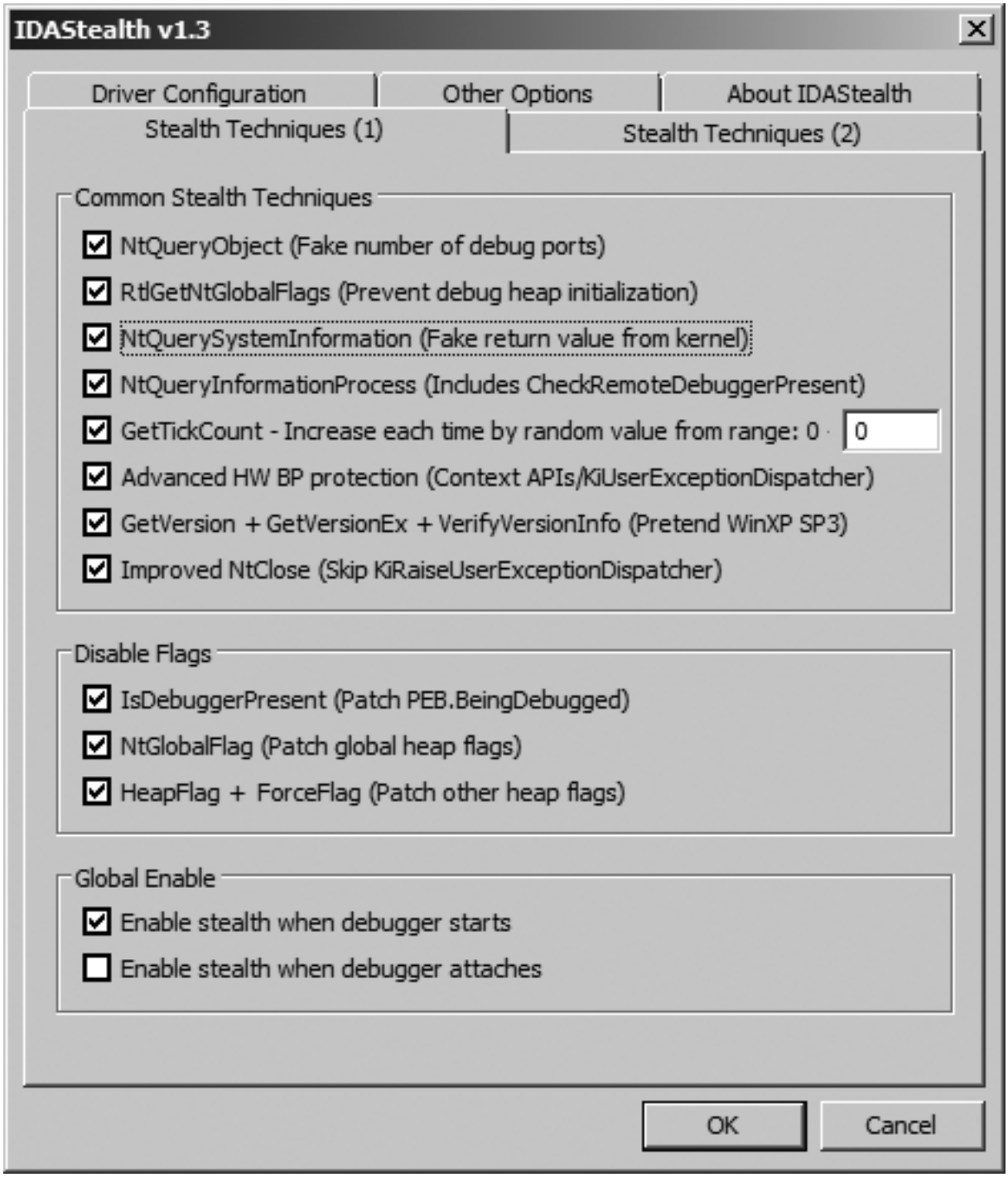

Рис. 25-6. Диалог конфигурации IDAStealth

Plugin можно включать при start и attach, выбирать отдельные techniques и настраивать driver. Это удобнее большого startup script, но не делает debugger невидимым абсолютно. Новая или sample-specific проверка может остаться. Кроме того, kernel driver расширяет attack surface лаборатории и требует доверия к plugin.

Работа с исключениями

Exception может означать bug, breakpoint, single step или намеренный control-flow mechanism. Debugger решает два независимых вопроса: остановиться ли для пользователя и передать ли

exception приложению.

Debugger > Debugger Options открывает общий setup с logging и кнопкой `Edit exceptions`.

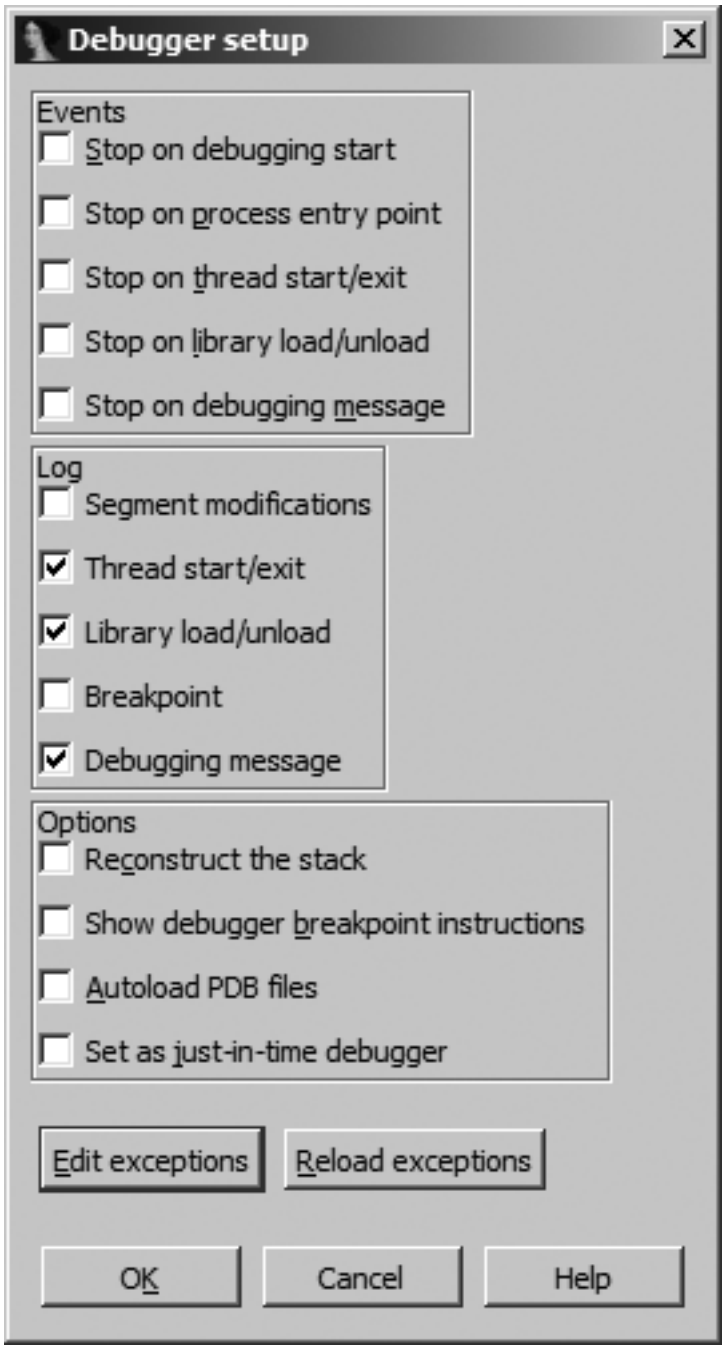

Рис. 25-7. Окно Debugger Setup

Таблица Exceptions содержит code, name, stop policy и получателя.

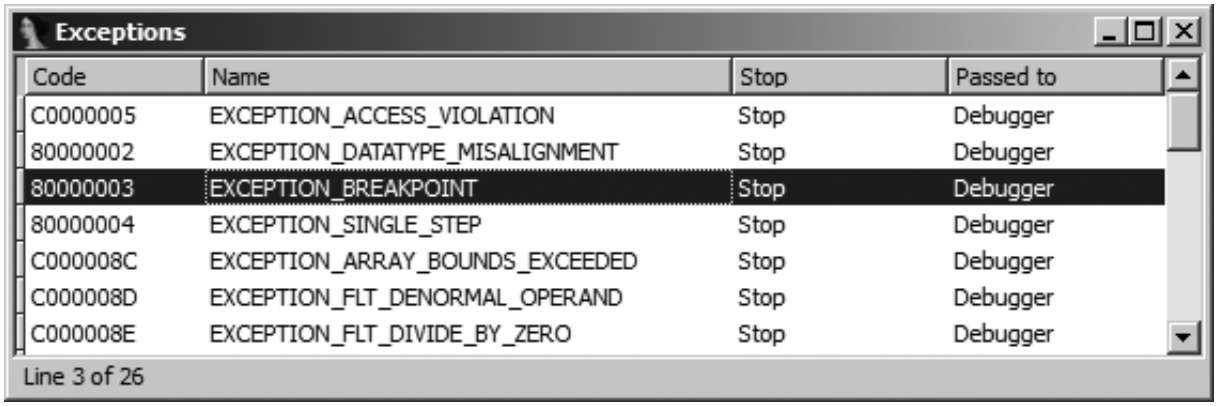

Рис. 25-8. Конфигурация exceptions

Глобальные defaults хранятся в configuration file, а отдельная IDB может переопределять их. Context menu > Edit открывает параметры конкретного code.

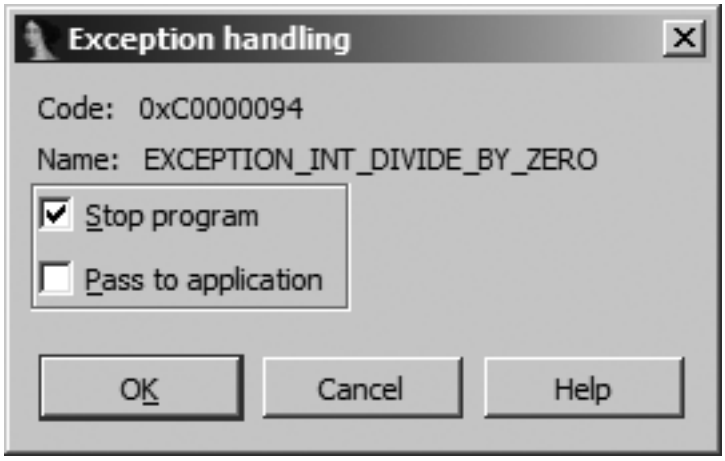

Рис. 25-9. Настройка обработки одного exception

`Stop program` заставляет debugger приостановиться. Если снять его для exception, которое повторяется на той же faulting instruction, process может войти в бесконечный цикл. `Pass to application` определяет, получит ли событие собственный handler. Если debugger поглотит deliberate exception, protected program пойдет по неправильному path.

При resume после остановки IDA может спросить, передавать ли exception приложению.

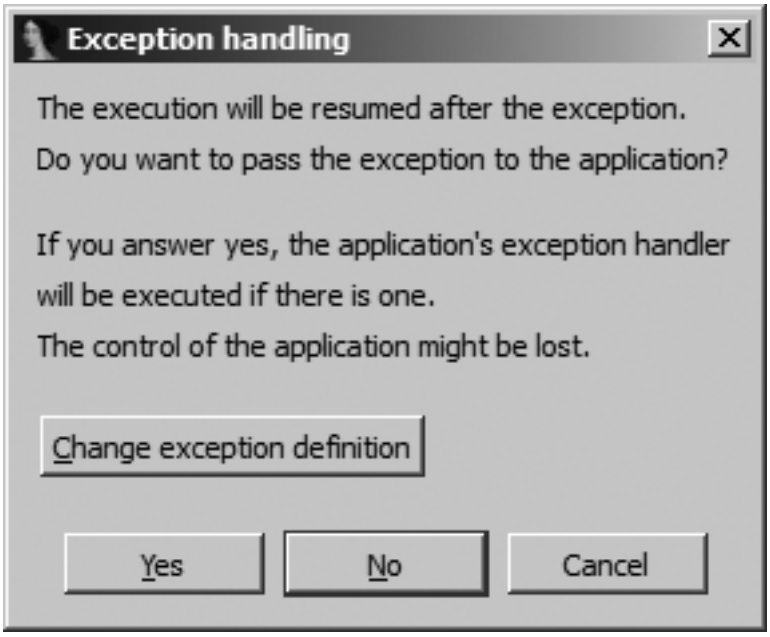

Рис. 25-10. Диалог Exception Handling

`Yes` запускает application handler, если он есть; `No` продолжает так, будто exception обработал debugger. Потерять control flow легко, поэтому решение должно основываться на назначении exception.

Некоторые instructions устанавливают trace flag или генерируют событие, различающее обычный run и single step. IDA показывает дополнительное предупреждение.

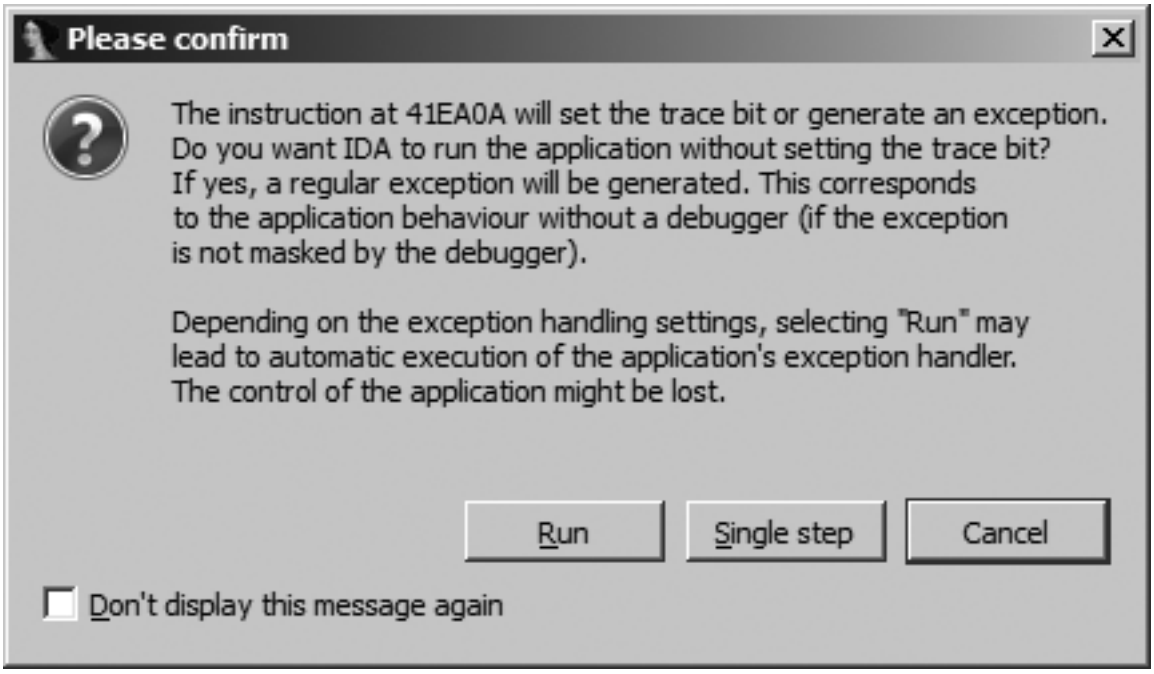

Рис. 25-11. Подтверждение поведения instruction, устанавливающей trace flag

Обычно `Run` лучше воспроизводит native behavior и позволяет application handler отработать согласно policy. `Single step` оставляет debugger в более жестком контроле, но может изменить наблюдаемый результат anti-debug check.

Windows SEH chain

Structured Exception Handling хранит для каждого thread цепочку records, доступную через `fs:[0]` на 32-bit x86. Debugger window SEH Chain показывает address record и handler.

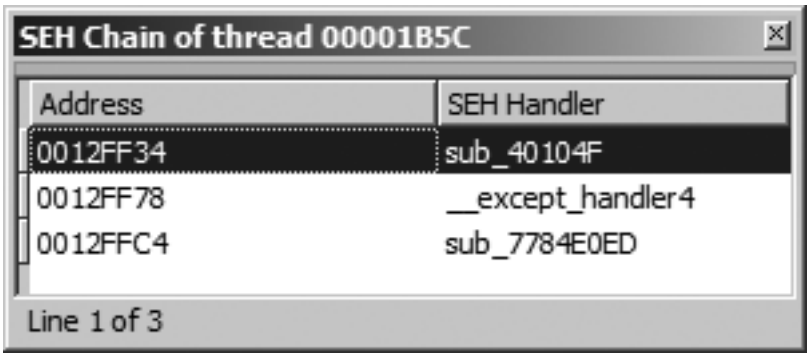

Рис. 25-12. Окно SEH Chain для выбранного thread

В цепочке могут быть user handler, compiler helper `__except_handler4` и OS frame. Double-click переходит к handler. Перед deliberate exception полезно снять snapshot chain и поставить breakpoint на первый application handler.

Handler получает `EXCEPTION_RECORD`, establisher frame, `CONTEXT` и dispatcher context. Изменяя `CONTEXT.Eip`, он выбирает address возобновления; также может менять registers и debug registers. Поэтому простой взгляд на faulting instruction не раскрывает next basic block.

Windows возвращает управление из exception через `NtContinue`, восстанавливающую `CONTEXT`. Чтобы не step-ать через kernel/user dispatcher, можно поставить breakpoint на `NtContinue`, прочитать pointer на `CONTEXT` из arguments, извлечь saved `Eip` и поставить следующий breakpoint на него. После продолжения IDA остановится в первом instruction реального resumed path.

Схема автоматизации:

```
// На входе NtContinue, 32-bit пример
ctx = Dword(ESP + 4);
resume_eip = Dword(ctx + offset_Eip);
AddBpt(resume_eip);
```

`offset_Eip` зависит от architecture и точного layout `CONTEXT`. На x64 используются другие registers, calling convention и fields.

Практический порядок анализа защищенного файла

1. Открыть executable статически, найти entry point и TLS callbacks.
2. Выбрать изолированный local/remote target и настроить earliest pause events.
3. Поставить breakpoints на callbacks, entry, anti-debug APIs и probable decoder exit.
4. Нормализовать PEB/heap/API results вручную или через IDAStealth.
5. Настроить exception policy так, чтобы deliberate exceptions дошли до приложения.
6. Следить за executable memory writes и transfer в новый region.
7. На OEP refresh memory и создать snapshot нужных segments.
8. Повторно определить code/data и восстановить IAT names.
9. Сохранить отдельную IDB до и после deobfuscation, чтобы можно было проверить преобразования.

Ни один heuristic - popa, резкий jump, RWX region или API pair - не универсален. Современный protector может распаковывать code по function, использовать VM или удалять восстановленные pages после выполнения. Тогда tracing и snapshots выполняются итеративно.

Итоги

Связка disassembler/debugger превращает IDA в рабочую среду для перехода от file image к process behavior. Snapshots сохраняют runtime modifications, segment attributes управляют их местом в IDB, а module exports позволяют вернуть имена динамическим imports.

При анализе защиты нужно получить контроль раньше TLS callbacks, распознать конец decoder, скрыть debugger достаточно для конкретного sample и корректно передавать deliberate exceptions. SEH и `NtContinue` показывают, что реальный control flow иногда существует только в runtime structures. Именно объединение static database, debugger events и scripting делает такие переходы наблюдаемыми и воспроизводимыми.

Глава 26. Дополнительные возможности отладчика

Заключительная глава рассматривает три функции, расширяющие обычный debugging: remote servers, эмуляцию Bochs и Appcall. Они позволяют отделить GUI от target, исполнять x86 в программной модели и вызывать functions активного process прямо из IDC или IDAPython.

Удаленная отладка с IDA

Все версии IDA поставлялись с server components для remote debugging. IDA также могла работать как frontend к `gdbserver` и встроенным GDB stubs. После настройки connection remote session почти не отличается от local: те же breakpoints, registers, memory views и stepping доступны через GUI.

Главные причины использовать remote target:

• target OS или CPU отличается от workstation;
• executable должен работать на embedded device;
• потенциально опасный process нужно изолировать;
• GUI IDA нежелательно устанавливать на production/test host;
• target имеет особое hardware или environment.

Remote debugging не является автоматическим sandbox. Server выполняет настоящий binary с правами своего пользователя, а незашифрованный protocol может раскрыть data и credentials.

Серверы Hex-Rays

На target запускается только подходящий server binary, полная IDA там не нужна. Историческая поставка включала:

• `win32_remote.exe` для 32-bit Windows;
• `win64_remotex64.exe` для 64-bit Windows;
• `wince_remote_arm.dll` для Windows CE через ActiveSync;
• `mac_server` и `mac_serverx64` для OS X;
• `linux_server` и `linux_serverx64` для Linux;
• `armlinux_server` для ARM Linux;
• `android_server` для Android.

64-bit variants требовали IDA Advanced. На Unix servers зависят от shared libraries; `ldd` или `otool -L` показывают dependencies.

Command-line options:

```
-p<port>      другой TCP port, по умолчанию 23946
-P<password>  password клиента
-v            verbose mode
```

Между option и value пробел не ставится. Server не имел option привязки к конкретному IP; доступ ограничивается host firewall. Один instance обслуживает одну active session. Для параллельных sessions нужно запустить несколько instances на разных ports.

Настройка клиента

Перед Start или Attach откройте Debugger > Process Options.

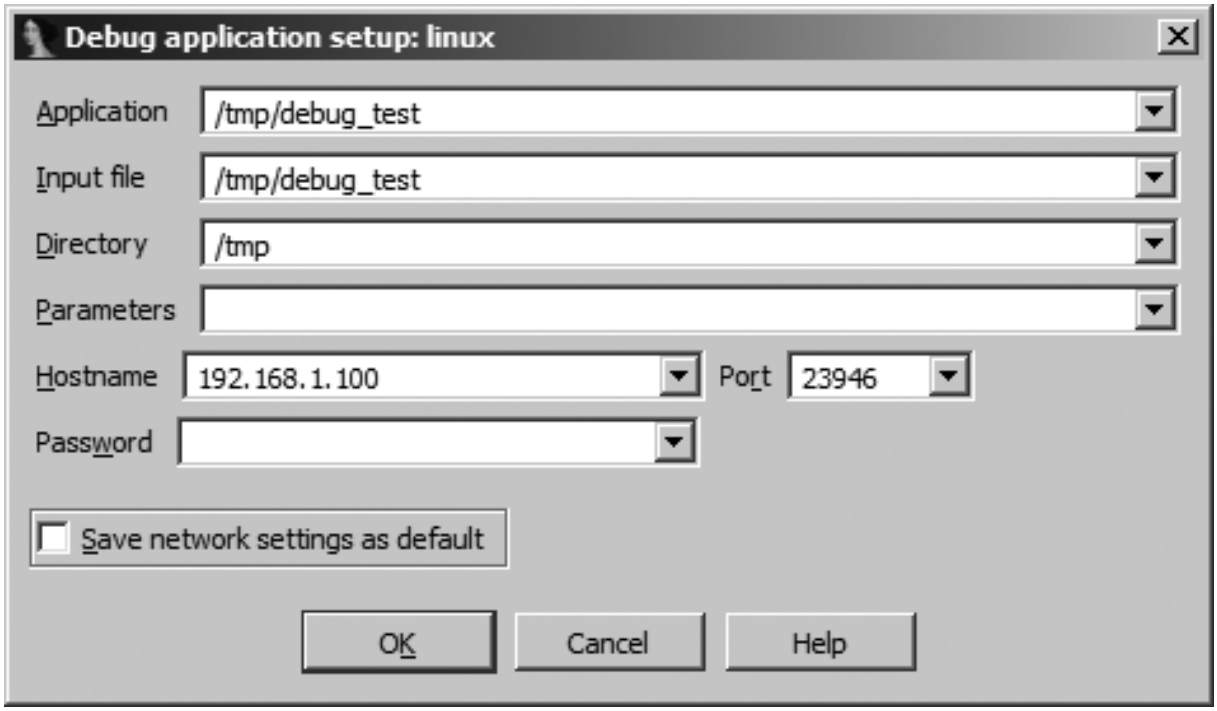

Рис. 26-1. Диалог Process Options для remote debugger

Поля:

• `Application` - executable, который будет запущен. Для remote session это path на server.
• `Input file` - file, из которого создана IDB, также path на server при remote debugging.
• `Directory` - working directory process на target.
• `Parameters` - command-line arguments. Shell metacharacters не обрабатываются и передаются буквально, поэтому redirection через <, > или | не работает.
• `Hostname` - имя или IP server; пустое значение означает local session.
• `Port` - listener port.
• `Password` - password, заданный через `-P`.

Output remote process появляется в console, где запущен server. Исторический password не маскировался в UI и передавался plain text. Поэтому такое соединение допустимо только в доверенной изолированной сети либо внутри защищенного tunnel.

`Application` и `Input file` часто совпадают, но не всегда. DLL нельзя запустить самостоятельно: Input file указывает DLL, открытую в IDA, а Application - host executable, который ее загружает. Такое разделение позволяет отлаживать library в реальном consumer process.

Remote GDB

Подключение к `gdbserver` похоже, но password не используется. Кнопка `Set specific options` открывает GDB configuration.

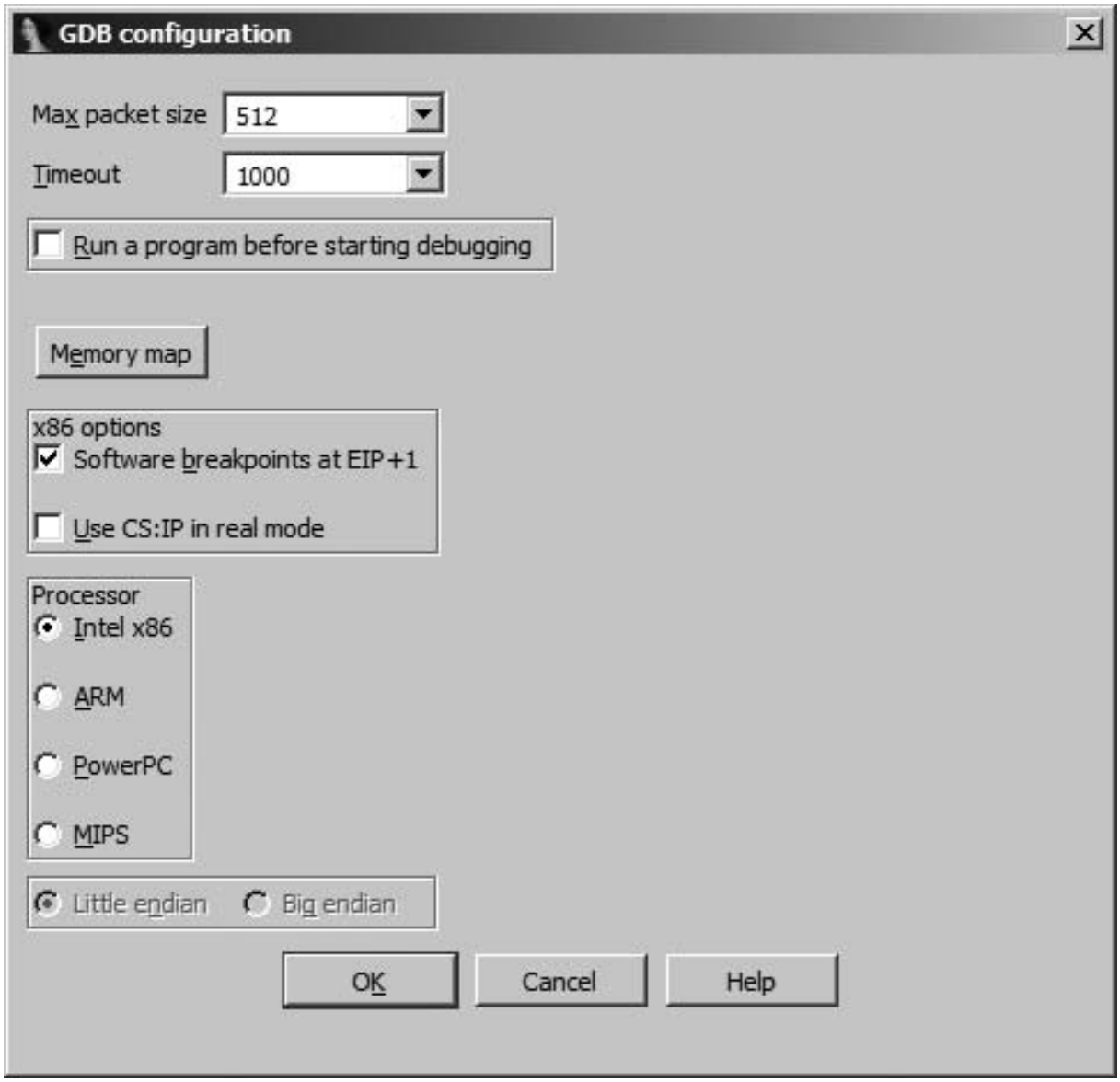

Рис. 26-2. Диалог GDB Configuration

Задаются maximum packet size, timeout, запуск вспомогательной программы, memory map, x86 breakpoint behavior, real-mode `CS:IP`, processor и endianness. IDA не может надежно вывести architecture target только из protocol, поэтому пользователь выбирает x86, ARM, PowerPC или MIPS и little/big endian. Неверный выбор приводит к неправильным registers и disassembly.

Memory map особенно важна для bare-metal stub, где OS loader не сообщает segments. Software breakpoint at `EIP+1` корректирует особенности x86 `int 3` reporting у некоторых stubs.

Присоединение к удаленному процессу

Если IDB не открыта, Debugger > Attach > Remote backend сначала запрашивает hostname, port и password.

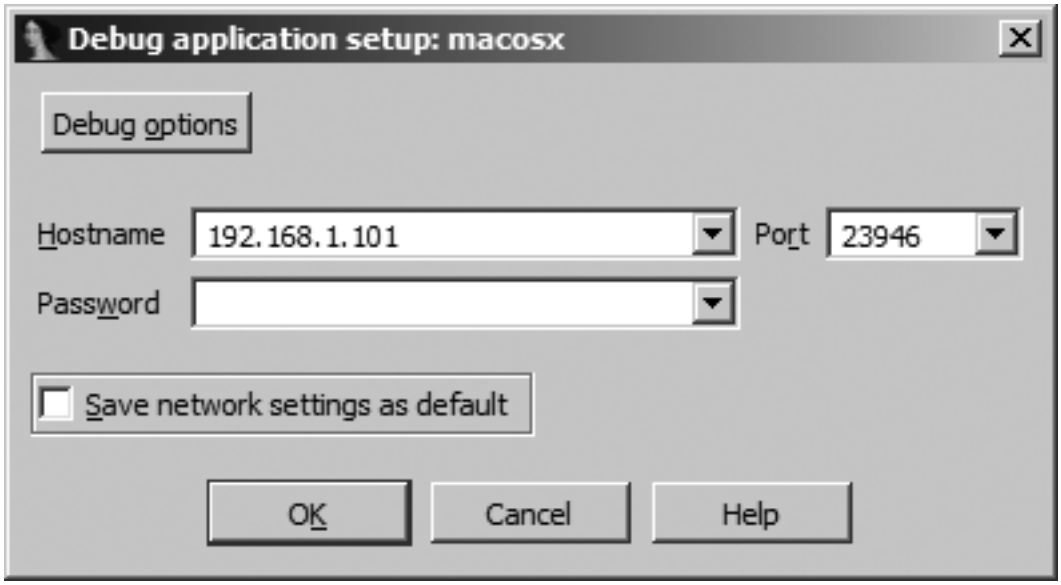

Рис. 26-3. Конфигурация remote debugger перед attach

После connection server возвращает process list. IDA создает temporary database из selected memory image, как при local attach. Если executable заранее открыт, connection parameters берутся из Process Options, a static database сохраняет свои annotations.

Практически предпочтительно заранее открыть тот же build и проверить hash. Иначе addresses и bytes IDB могут не соответствовать process. При ASLR IDA rebases module, но она не исправит semantic mismatch другой версии binary.

Отладка с Bochs

Bochs - open-source full-system x86 emulator с devices и BIOS. Интеграция с IDA дает emulation-based alternative native debugging. Windows builds IDA включали совместимый Bochs; на других systems требовалась версия 2.4.2 или новее.

Для x86 binary выбирается `Local Bochs debugger`. Поскольку guest instructions интерпретируются emulator-ом, Windows application можно исследовать на non-Windows host. Существуют три принципиально разных режима: IDB, PE и Disk image.

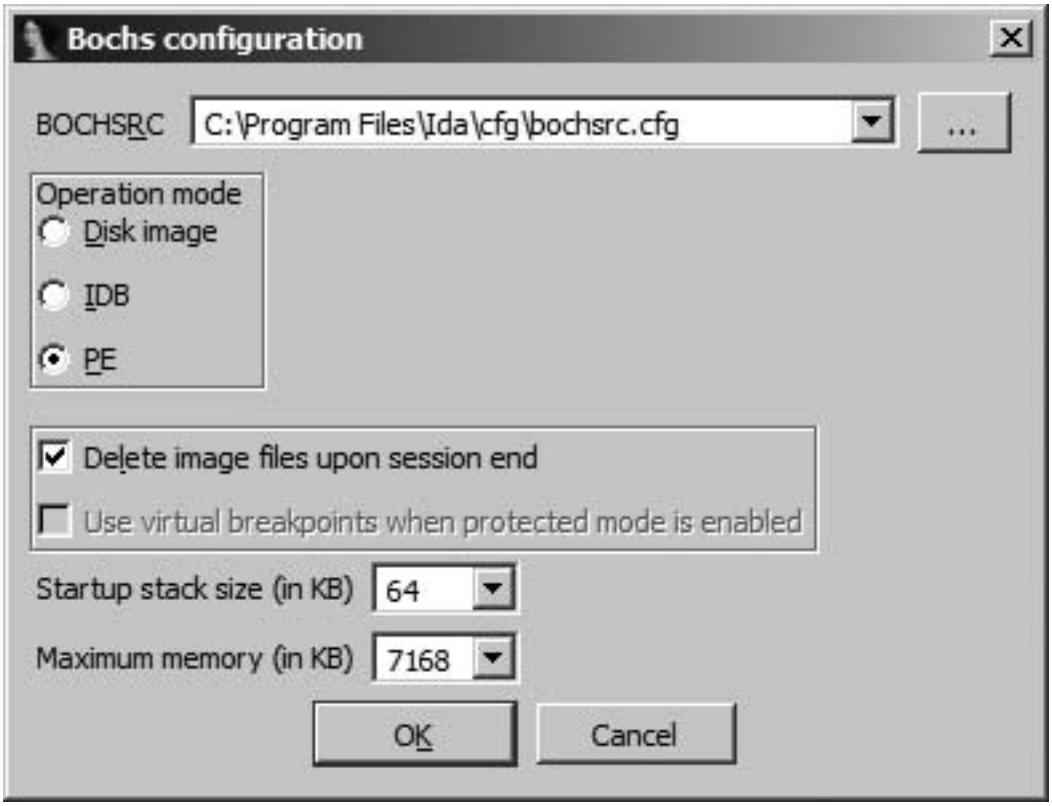

Рис. 26-4. Диалог конфигурации Bochs debugger

Диалог задает `bochsrc`, operation mode, удаление temporary image files, virtual breakpoints, startup stack size и maximum memory.

Режим IDB

Это минимальная модель. Bochs видит только bytes и segments текущей database. IDA копирует их в emulator memory и выделяет configurable stack. Начальный IP выбирается в порядке:

1. symbol `ENTRY`;
2. начало выделенного range;
3. позиция cursor.

Никакой OS support нет: shared libraries, syscalls, PEB/TEB и well-known process structures отсутствуют. Поэтому одинаково можно выполнить отдельную function из PE, ELF, Mach-O или raw shellcode, если она не обращается за пределы database либо такие зависимости смоделированы вручную.

IDB mode удобен как controlled instruction laboratory: задать registers/stack, выполнить algorithm и наблюдать результат. Он плохо подходит для кода, тесно связанного с OS или multi-threading.

Режим PE

PE mode приближается к process-level emulation. Control plugin играет роль Windows loader: создает PEB, TEB и stack, размещает main image и common DLLs. Code библиотек может быть загружен как data, но необязательно исполняется нативно.

Набор libraries задается в `<IDADIR>/plugins/bochs/startup.idc`. Library можно загрузить полностью или отметить stubbed. Для stubbed library plugin перехватывает каждый export и

направляет его в script implementation. Файлы ищутся как `api_foolib.idc` или `api_foolib.py`; `api_kernel32.idc` служит образцом.

Stubs заменяют отсутствующую OS. Например, реальный `VirtualAlloc` ожидает kernel memory manager, a scripted implementation может выделить emulator pages и вернуть правдоподобный pointer. Чем больше relevant API смоделировано, тем дальше пройдет program. Неверный stub способен незаметно направить execution по нереальному path.

На non-Windows host необходимые DLL можно скопировать из лицензированной Windows system и прописать mapping в `startup.idc`:

```
path /home/idauser/xp_dlls/=c:\winnt\system32\
```

В PE mode IDA не показывает обычное предупреждение о запуске malware, поскольку создается только Bochs process, а исследуемые bytes являются данными emulator. Это снижает риск native execution, но emulator, scripts, shared folders и network configuration все равно требуют защиты.

Режим Disk image

Bochs способен загрузить полноценный guest OS с virtual disk, BIOS и devices. Disk image создается `bximage`, затем на него устанавливается OS. Это максимальная fidelity, но наблюдать один user process внутри whole system сложно: нужно понимать scheduler, paging, process address spaces и kernel transitions.

Наиболее прямое применение IDA/Bochs disk mode - BIOS, MBR и ранний boot code, пока control flow еще прост. IDA loader открывает `bochsrc`, читает первый sector первого disk image и выбирает Bochs debugger. Default configuration находилась в `<IDADIR>/cfg/bochsrc.cfg` и задавала BIOS, video ROM и disks.

Для обычной application function IDB/PE modes проще. Disk image оправдан, когда behavior зависит от real OS kernel, boot chain, drivers или hardware model.

Appcall

Appcall делает любую function активного debugged process вызываемой из IDC или IDAPython. Возможные применения:

• вызвать `VirtualAlloc` и выделить scratch memory;
• загрузить library через `LoadLibrary`;
• попросить сам process декодировать block;
• вычислить proprietary hash;
• вызвать internal parser или helper с controlled arguments.

Function должна находиться в address space, иметь name и type prototype, достаточный для marshaling parameters и return value. Appcall выполняется в context текущего debugger thread. IDA сохраняет registers, организует call, ожидает completion и восстанавливает thread state, чтобы normal debugging мог продолжиться.

Пример VirtualAlloc

Исходный prototype:

```
LPVOID WINAPI VirtualAlloc(
```

```
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);
```

Нужный C-call:

```
VirtualAlloc(NULL, 4096, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

После разрешения constants:

```
VirtualAlloc(0, 4096, 0x3000, 4);
```

Во время debugging IDA исторически добавляла library prefix, поэтому symbol назывался `kernel32_VirtualAlloc`. Type library могла не узнать измененное имя. Через Set Function Type достаточно назначить compatible signature:

```
int __stdcall kernel32_VirtualAlloc(int, int, int, int);
```

Точная semantic type информация желательна, но для механики call критичны calling convention, число, sizes и порядок arguments и return type.

IDC вызывает function как встроенную:

```
IDC>Message("%x\n", kernel32_VirtualAlloc(0, 4096, 0x3000, 4));
3c0000
```

Результат - address нового 4096-byte block. Чтобы mapping появился в IDA, выполните Debugger > Refresh Memory или дождитесь следующего refresh event.

В IDAPython используется object `Appcall` из `idaapi`:

```
Message("%x\n" % Appcall.kernel32_VirtualAlloc(0, 4096, 0x3000, 4))
```

Ограничения и безопасность Appcall

Appcall не является чистой эмуляцией: function реально выполняется в process. Она может изменить global state, heap, files, network и threads. Восстановление registers не отменяет side effects. Нельзя вызывать неизвестную malware function на production target только потому, что IDA вернет thread context.

Необходимо учитывать:

• текущий thread должен быть в безопасном состоянии, не удерживать несовместимый lock;
• prototype и calling convention должны быть верны;
• pointers должны ссылаться на valid process memory;
• function может вызвать exception, never return или завершить process;
• callback, TLS и thread-local state могут влиять на результат;
• x64 и structure-by-value требуют особенно точных types.

Лучше сохранить snapshot/IDB, использовать disposable isolated target и начинать с well-understood API. Для internal algorithms полезно подготовить input buffer через debugger memory, вызвать function и извлечь output после refresh.

Итоги

Remote servers отделяют IDA GUI от platform, где реально работает process. Для надежной session нужно правильно задать target paths, architecture, endianness и защитить network channel. Один

server обслуживает одну session, а executable на target должен точно соответствовать IDB.

Bochs предлагает три уровня emulation: минимальный IDB для самостоятельных code fragments, PE с моделируемым Windows environment и Disk image для whole system. Fidelity растет вместе со сложностью настройки и наблюдения.

Appcall превращает type-aware database в активный инструмент: script может вызвать API или внутренний algorithm process. При этом вызов исполняется по-настоящему и сохраняет side effects. Вместе remote debugging, emulation и Appcall завершают картину IDA как расширяемой среды, объединяющей static representation, controlled execution и automation.