ИСКУССТВО ДИЗАССЕМБЛИРОВАНИЯ

Санкт-Петербург

Оглавление

Введение

Книга, которую вы сейчас держите в руках, открывает двери в удивительный мир реверсинга (обратной разработки) защитных механизмов. Она адресована всем, кто любит головоломки и готов сделать свои первые шаги на пути к тому, чтобы стать настоящим хакером. Всем, кто проводит свободное (и несвободное) время за копанием в недрах программ и операционной системы. Наконец, всем, кто по роду своей деятельности занимается (постоянно или эпизодически) написанием защит и хочет узнать, как грамотно и гарантированно противостоять вездесущим хакерам.

Книга посвящается обратной разработке — пожалуй, наиболее сложному из аспектов хакерства, ведь дизассемблирование — это искусство. Однако авторы приложили максимум усилий к структурированию материала таким образом, чтобы изложение было выстроено логично, а читатель, овладевая искусством дизассемблирования, двигался "от простого к сложному". В начале книги излагаются базовые основы хакерства — техника работы с отладчиками, дизассемблерами, шестнадцатеричными редакторами, API- и RPC-шпионами, эмуляторами. Приводится широкий обзор популярного хакерского инструментария для Windows, UNIX и Linux, а также демонстрируются практические приемы работы с популярными отладчиками (SoftICE, OllyDbg, WinDbg), от широко известных до нетрадиционных. Подробно описаны приемы идентификации и реконструкции ключевых структур исходного языка — функций (в том числе виртуальных), локальных и глобальных переменных, ветвлений, циклов, объектов и их иерархий, математических операторов и т. д. Наряду с этим демонстрируются различные подходы к анализу алгоритма изучаемых программ и объясняется, как не заблудиться в мегабайтах дизассемблированного кода и избежать разнообразных хитрых ловушек. Значительное внимание уделено таким важным темам, как реконструкция алгоритмов работы защитных механизмов, идентификация ключевых структур языков высокого уровня, таких, как C/C++ и Pascal. Рассматриваются практические методы преодоления антиотладочных приемов, техника снятия дампа с защищенных приложений, преодоление упаковщиков и протекторов. На практических примерах продемонстрированы методы анализа кода вредоносных программ и эксплоитов. Не оставлены без внимания и такие важные темы, как противодействие антиотладочным приемам, исследование упакованного, зашифрованного и умышленно запутанного (Obfuscated) кода, а также другим технологиям, затрудняющим дизассемблирование и отравляющим хакерам жизнь.

ЧАСТЬ I

ОБЗОР

ХАКЕРСКИХ ПРОГРАММ

Глава 1

Инструментарий хакера

С чего начинается хакерство? Некоторые скажут — с изучения C/C++, языка ассемблера, обучения искусству отладки и дизассемблирования… И будут правы. Другие же добавят к этому "джентльменскому списку" изучение архитектуры разнообразных операционных систем, сетевых протоколов, поиск уязвимостей. И тоже будут правы. Но, с другой стороны — а за счет чего хакеры добиваются результатов? Правильно, за счет знаний и упорного труда. Но при этом они не только работают руками и головой, но и пользуются хакерским инструментарием. И правильный подбор программ очень важен, поскольку именно они формируют сознание, позволяя начинающему сделать свои первые шаги в дремучем лесу машинных кодов. Вот только этих программ настолько много, что новичок, впервые попавший на хакерский сайт, оказывается в полной растерянности — что качать, а что не стоит внимания. Основная цель данной главы как раз и состоит в том, чтобы дать предельно сжатый обзор хакерского софта, покрывающего практически любые потребности.

> ***Примечание***
>
> Помимо упорства и стремления к самостоятельному поиску ответов на возникающие вопросы, девизом любого хакера должна стать фраза: "Знай и люби свои инструменты". Иными словами, вы не должны довольствоваться лишь приведенным здесь кратким обзором, который призван лишь обратить ваше внимание на ту или иную программу. Как правило, в комплекте с любой из этих программ поставляется и руководство пользователя, и руководства эти настоятельно рекомендуется внимательно изучать.

Отладчики

Лучший отладчик всех времен и народов — это, конечно же, SoftICE, на котором выросло не одно поколение хакеров. Это — интерактивная программа с развитым командным интерфейсом, представляющим собой компромисс между легкостью освоения и удобством использования (рис. 1.1). Иными словами, без чтения руководства здесь не обойтись, тем более что никаких интуитивно-понятных меню SoftICE не предоставляет[1].

Изначально созданный фирмой NuMega, SoftICE был продан компании Compuware, долгое время распространявшей его в составе громоздкого пакета DriverStudio Framework. К глубокому разочарованию, 3 апреля 2006 по малопонятным причинам компания объявила о прекращении работы над продуктом, похоронив тем самым уникальнейший проект. Последняя версия DriverStudio 3.2 поддерживает всю линейку Windows вплоть до Server 2003, а также архитектуру

[1] Подробное "Руководство пользователя SoftICE" в русском переводе находится здесь: **http://www.podgoretsky.com/ftp/Docs/softice/siug.pdf**. Кроме того, достаточно краткое, но весьма полезное руководство по использованию SoftICE, позволяющее быстро начать работу с данным отладчиком, можно найти по адресу **http://www.reconstructer.org/papers/The%20big%20SoftICE%20howto.pdf**.

AMD x86-64. То есть лет на пять запаса прочности у SoftICE еще должно хватить, а там хакеры что-нибудь придумают [2].

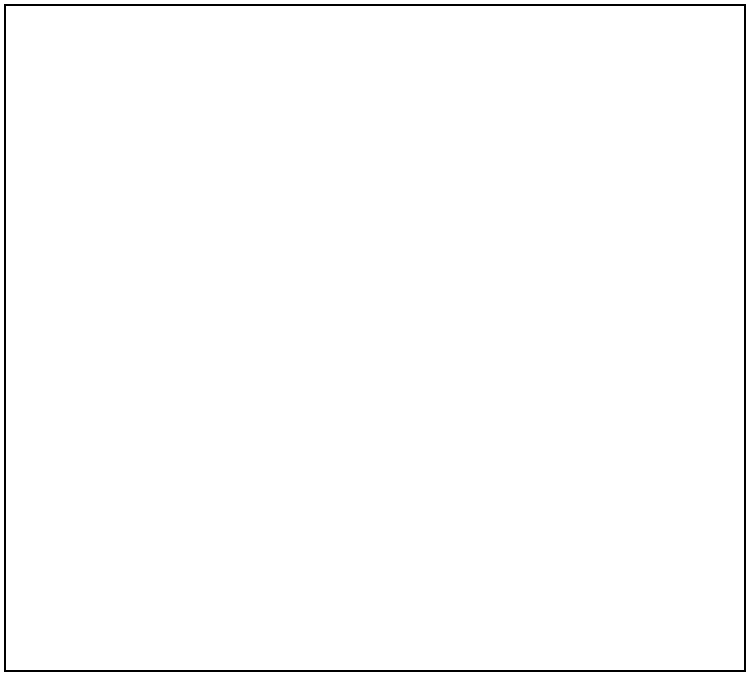

Рис. 1.1. SoftICE — профессионально ориентированный отладчик, на котором выросло не одно поколение хакеров

Найти SoftICE можно практически на любом хакерском сайте (например, здесь: **http://www.woodmann.com/crackz/Tools.htm**). Чтобы не качать целиком весь пакет DriverStudio, можно воспользоваться пакетом DeMoNiX (**http://reversing.kulichki.net**), содержащим только SoftICE, выдернутый из DriverStudio v. 2.7 build 562. Этот пакет занимает всего 2,27 Мбайт. Однако инсталлятор содержит ошибки, а старая версия не поддерживает новых веяний Microsoft (хотя замечательно идет под Windows 2000).

Вместе с SoftICE желательно сразу же установить IceExt (**http://sourceforge.net/projects/iceext**) — неофициальное расширение, позволяющее скрывать присутствие отладчика от большинства защит, сохранять дампы памяти, задействовать кириллические кодировки 866/1251, приостанавливать потоки и выполнять множество других важных задач (рис. 1.2).

Если IceExt откажется запускаться, скорректируйте следующие ключи в данной ветви системного реестра: `HKLM\SYSTEM\CurrentControlSet\Services\NTice: KDHeapSize (DWORD): 0x8000`; `KDStackSize (DWORD): 0x8000`.

Другое неофициальное расширение для SoftICE, IceDump (**http://programmerstools.org/system/files?file=icedump6.026.zip**), также умеет делать много полезных вещей и удачно дополняет IceExt (рис. 1.3).

[2] На данный момент на роль адекватного преемника SoftICE претендует коммерческий отладчик Syser Debugger, о котором речь пойдет чуть далее. Если же вас интересует отладчик Open Source, то возможно, ваше внимание привлечет проект Rasta Ring 0 Debugger (**http://rr0d.droids-corp.org/**).

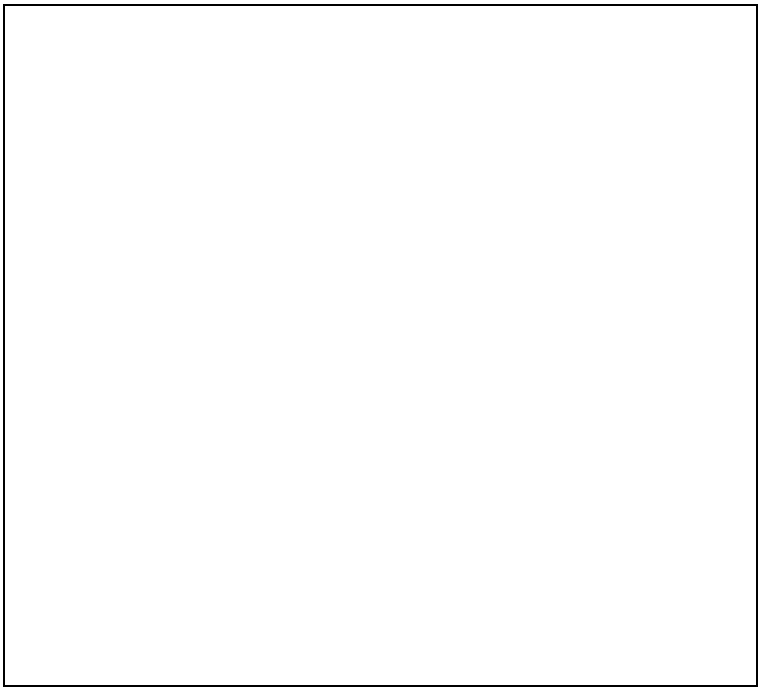

Рис. 1.2. Снятие дампа с помощью IceExt

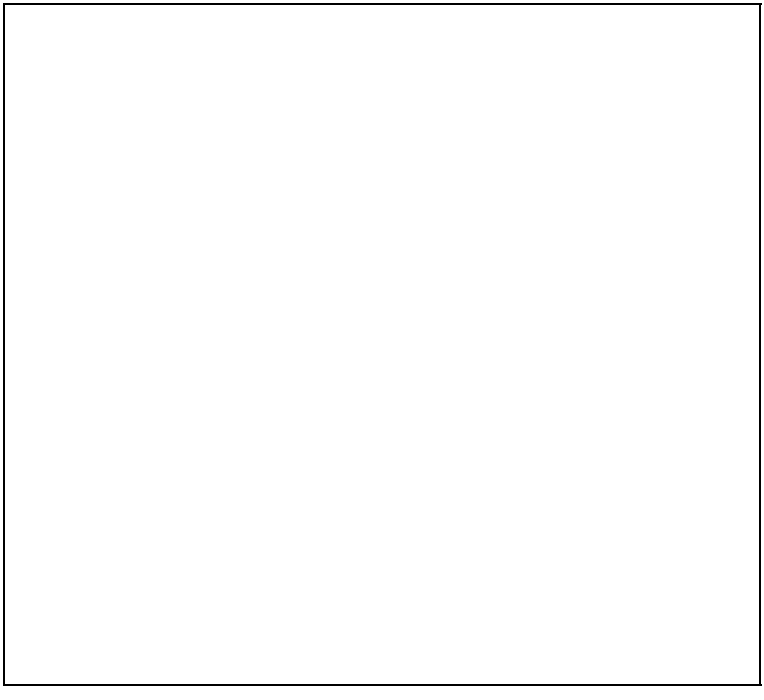

Рис. 1.3. Снятие дампа с помощью IceDump

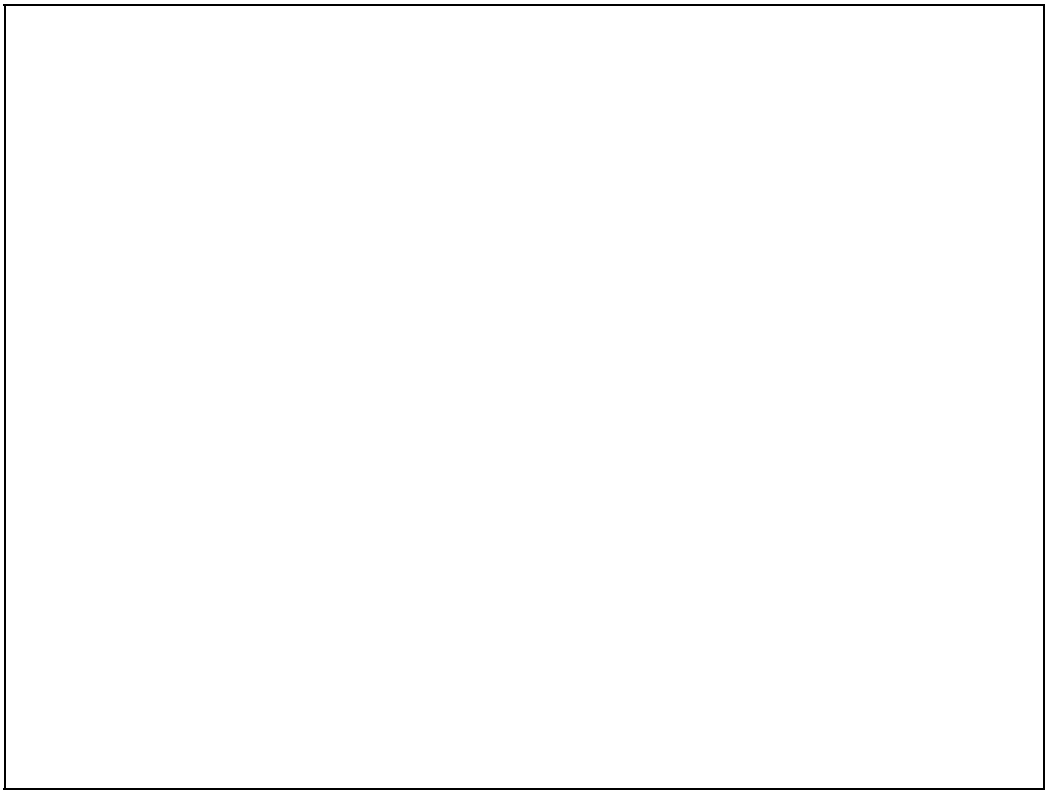

Рис. 1.4. Компактный и шустрый отладчик OllyDbg

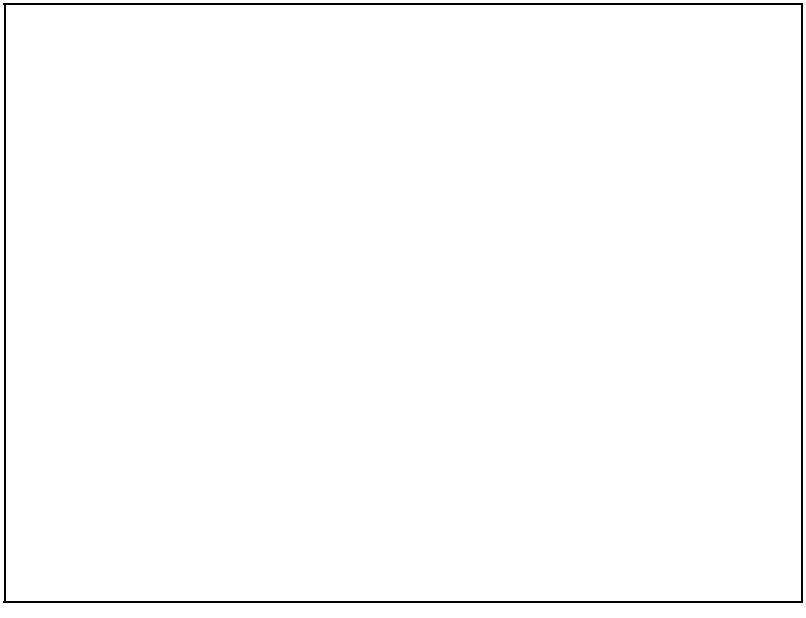

Рис. 1.5. Отладчик Syser Debugger за отладкой термоядерного драйвера

Кстати, сам SoftICE замечательно работает под виртуальной машиной VMware, для этого достаточно добавить в конфигурационный файл с расширением .vmx следующие две строки: `paevm = TRUE` и `processor1.use = FALSE`.

> ***Примечание***
>
> При работе с SoftICE на многоядерных процессорах и процессорах с поддержкой HyperThreading также были отмечены некоторые проблемы (хотя возникают они нерегулярно). Устранить эти проблемы в случае их возникновения можно, добавив ключ `/ONECPU` в файл boot.ini.

Помимо SoftICE существуют и другие отладчики, из которых в первую очередь хотелось бы отметить бесплатный Olly Debugger (**http://www.ollydbg.de**). Это — удобный инструмент прикладного уровня (рис. 1.4), ориентированный на хакерские потребности, поддерживающий механизм плагинов (plug-ins) и собравший вокруг себя целое сообщество, написавшее множество замечательных расширений и дополнений, прячущих OllyDbg от защит, автоматически определяющих оригинальную точку входа в упакованной программе, облегчающих снятие протекторов и т. д.

Неплохие коллекции плагинов можно найти на сайтах **http://www.wasm.ru** и **http://www.openrce.org**.

Самый свежий (и пока еще во многом экспериментальный) ядерный отладчик — это, бесспорно, Syser (**http://www.sysersoft.com**). Данный отладчик выпущен китайскими разработчиками и в настоящее время переживает стадию активного развития и становления (рис. 1.5). Как уже говорилось чуть ранее, этот отладчик претендует на роль преемника SoftICE. Его последняя версия (v. 1.9, датированная 17 мая 2007 года) работает с 32-разрядными версиями Windows 2000/XP/2003/Vista, а также обеспечивает поддержку SMP, HyperThreading и многоядерных процессоров.

Довольно многие программисты пользуются отладчиком уровня ядра Microsoft WinDbg, входящим в состав бесплатного набора Debugging Tools (**http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx**). Он вполне пригоден для взлома (рис. 1.6). Тем не менее, большинство хакеров, привыкших к черному экрану SoftICE, считают его неудобным.

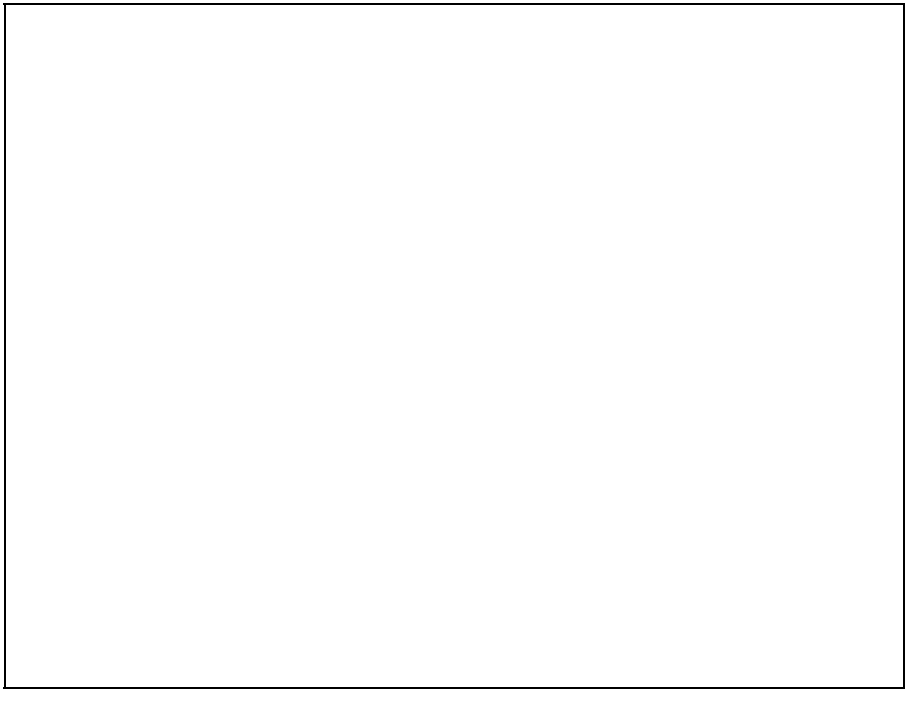

Рис. 1.6. Отладчик WinDbg

Дизассемблеры

Существует всего лишь один дизассемблер, пригодный для по-настоящему профессиональной работы — IDA Pro (**http://www.idapro.com**). Этот дизассемблер имеет консольную (рис. 1.7) и графическую (рис. 1.8) версии. IDA Pro воспринимает огромное количество форматов файлов и множество типов процессоров, легко справляясь с байт-кодом виртуальных машин Java и .NET, поддерживает макросы, плагины и скрипты, содержит интегрированный отладчик, работает под MS-DOS, Windows, Linux (рис. 1.9) и обладает уникальной способностью распознавать имена стандартных библиотечных функций по их сигнатурам.

Существует несколько версий IDA Pro — бесплатная (freeware), стандартная (standard) и расширенная (advanced). Бесплатную версию IDA Pro можно скачать по адресу **http://www.dirfile.com/ida_pro_freeware_version.htm**. Стоит, правда, отметить, что бесплатная версия, по сравнению со стандартной и расширенной, обладает ограниченными возможностями. Из всех процессорных архитектур поддерживается только x86, а функция поддержки подключаемых модулей попросту отсутствует. Что касается стандартной и расширенной версий, то они, как и любое хорошее программное обеспечение, стоят дорого (хотя, если хорошо поискать в файлообменных сетях, их можно стянуть оттуда бесплатно).

Основное достоинство IDA Pro состоит в том, что это — интерактивный дизассемблер, то есть интеллектуальный инструмент, позволяющий работать с двоичным файлом, мыслить и творить, а не тупой автомат, заглатывающий исследуемую программу и выплевывающий "готовый" дизассемблированный листинг, в котором все дизассемблировано неправильно.

Рис. 1.7. Консольная версия IDA Pro

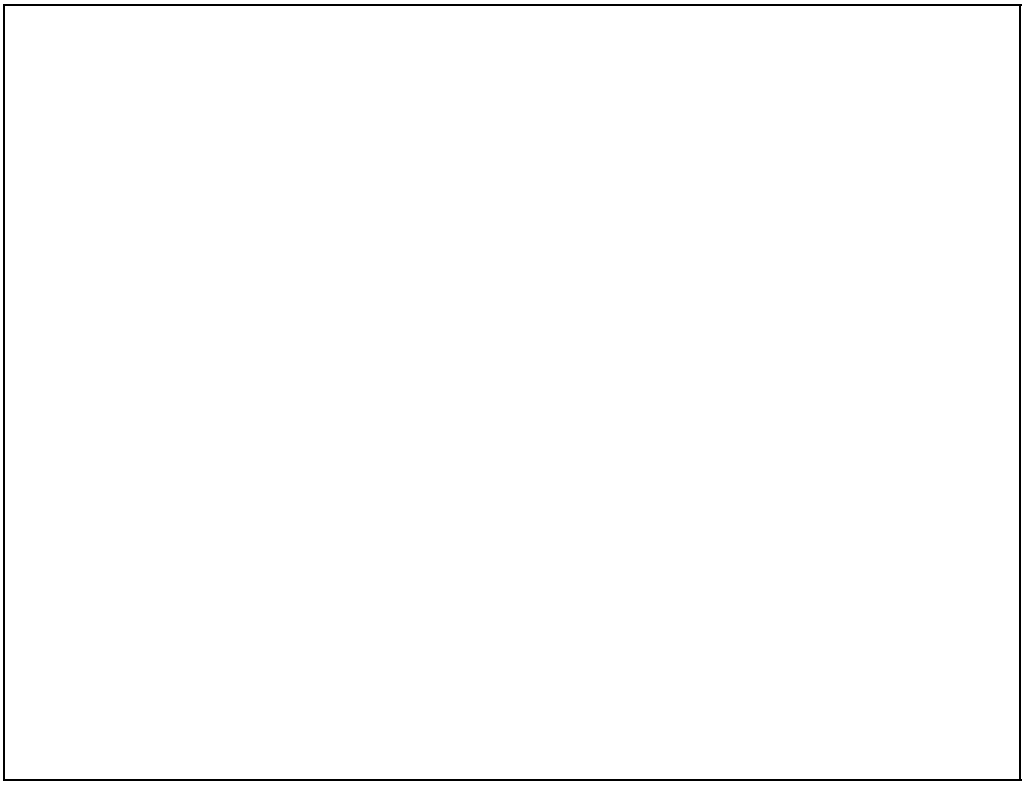

Рис. 1.8. Графическая версия IDA Pro

Рис. 1.9. IDA Pro под Linux

В последних версиях IDA Pro сделаны определенные подвижки в сторону автоматической распаковки файлов и снятия обфускаторов[3]. Внушительную коллекцию плагинов и скриптов можно найти как на официальном сайте, так и на сайте **http://www.openrce.org**.

Конкурирующие продукты не выдерживают никакого сравнения с IDA Pro. Тем не менее, народ активно качает бесплатный (ныне заброшенный) дизассемблер с функциями отладчика — W32Dasm (**http://www.wasm.ru/baixado.php?mode=tool&id=178**) и, судя по всему, остается доволен (рис. 1.10).

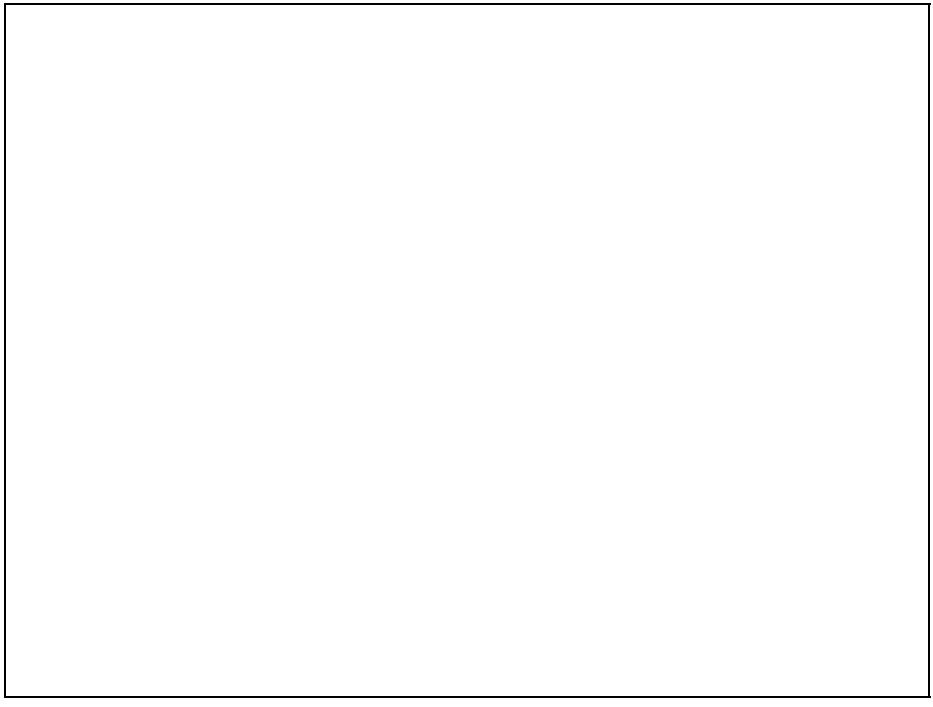

Рис. 1.10. Дизассемблер WDASM

Остальные дизассемблеры обладают еще более ограниченными возможностями, и поэтому здесь не рассматриваются. Единственный из такого рода продуктов, заслуживающий упоминания — это Hacker Disassembler Engine (**http://patkov-site.narod.ru/lib.html**), представляющий собой дизассемблер длин, распространяющийся в исходных текстах и предназначенный для встраивания в различные хакерские программы, занимающиеся перехватом функций, автоматической распаковкой, генерацией полиморфного кода и т. д.

Декомпиляторы

Декомпиляцией называется процесс получения исходного текста программы (или чего-то очень на него похожего) из двоичного файла. В полном объеме декомпиляция принципиально невозможна, поскольку компиляция — однонаправленный процесс, причем с потерей данных. Однако декомпиляторы все-таки существуют и со своей задачей достойно справляются.

[3] Более подробную информацию о распаковке файлов, снятии протекторов и обфускаторов можно найти в *части V* данной книги.

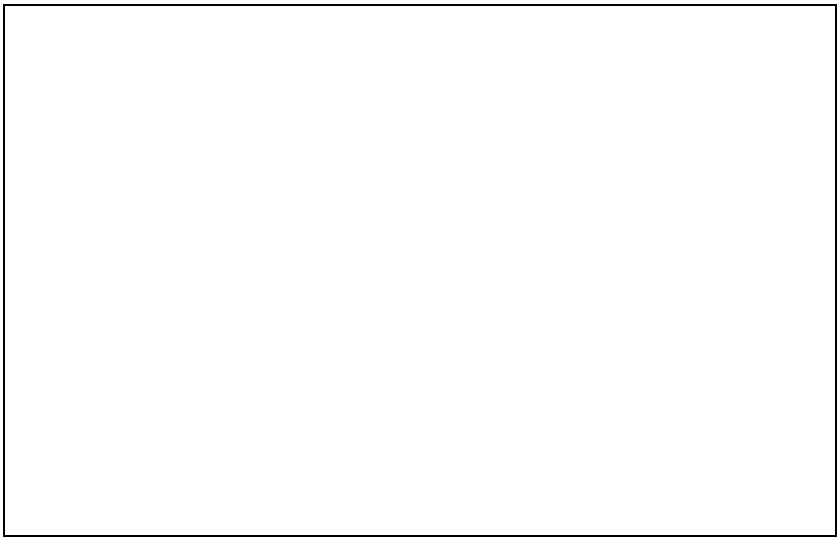

Рис. 1.11. Декомпилятор DeDe ломает HTTPProxy

Для программ, написанных на Delphi и Borland Builder с использованием RTTI, возможно восстановить исходную структуру классов вплоть до имен функций-членов, а также реконструировать формы и "вычислить" адреса обработчиков каждого из элементов. Допустим, у нас имеется диалоговое окно **Registration** с кнопкой **OK**, и мы хотим знать, какая процедура считывает серийный номер и что она с ним делает. Нет ничего проще! Берем бесплатный декомпилятор DeDe (**http://programmerstools.org/node/120**), декомпилируем программу и вперед (рис. 1.11)!

Для Visual Basic существуют свои декомпиляторы, лучшим из которых считается VB Decompiler от GPcH (**http://www.vb-decompiler.org/index.php?p=Products**). Другие бейсик-декомпиляторы, в том числе VB RezQ (**http://www.vbrezq.com/**), VBDE (**http://programmerstools.org/node/129**) и Spices.Decompiler (**http://programmerstools.org/node/635**) также полезно положить в свой хакерский чемоданчик.

Особый интерес представляют декомпиляторы программ-инсталляторов, поскольку многие проверки (на истечение срока работы демо-версии, на серийный номер или ключевой файл) производятся как раз на стадии инсталляции. Самый популярный инсталлятор — это InstallShield. Для него имеется множество удобных декомпиляторов. Вот только некоторые из них:

- InstallShield X Unpacker (**http://programmerstools.org/node/154**)
- Windows InstallShield Decompiler (**http://programmerstools.org/node/118**)
- InstallShield Decompiler (**http://programmerstools.org/node/114**)
- isDcc (**http://programmerstools.org/node/115**)

Что же касается Java и платформы .NET, то с ними замечательно справляется IDA Pro. Если же у вас нет под рукой IDA Pro, можно воспользоваться специализированными декомпиляторами, которые можно найти на сайтах **http://www.cracklab.ru** и **http://www.wasm.ru** вместе с декомпиляторами Fox Pro, Clipper и прочей экзотики.

Шестнадцатеричные редакторы

Давным-давно шестнадцатеричные редакторы (hex-редакторы) представали собой простые программы, умеющие лишь отображать двоичные файлы в шестнадцатеричном виде и править бай-

ты по указанным адресам[4]. Однако со временем они обросли встроенными дизассемблерами, ассемблерами, калькуляторами, функциями поиска регулярных выражений, научились работать с блоками, распознавать различные форматы файлов и даже расшифровывать/зашифровывать фрагменты кода или данных. В общем, превратились в эдакие швейцарские ножички с шестнадцатью лезвиями.

Наибольшую популярность завоевал HIEW (**http://webhost.kemtel.ru/~sen**). Вплоть до версии 6.11 (поддерживающей форматы MZ/PE/NE/LE/ELF) он распространялся на бесплатной основе (рис. 1.12). Стоит, правда, отметить, что старые и бесплатные версии обладают достаточными возможностями, и к тому же содержат гораздо меньше ошибок и багов.

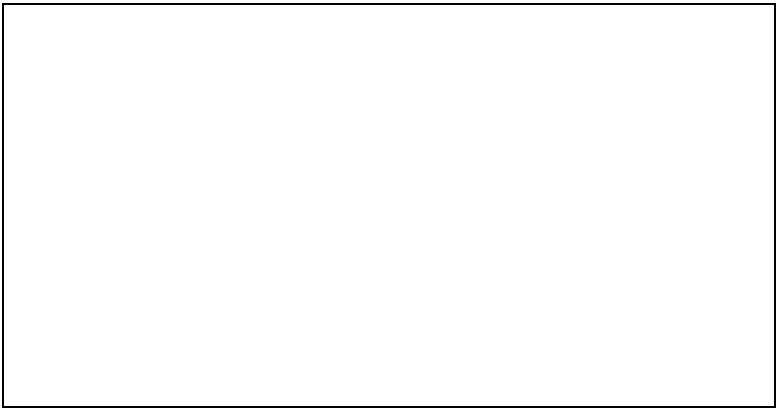

Рис. 1.12. Старый добрый hex-редактор HIEW

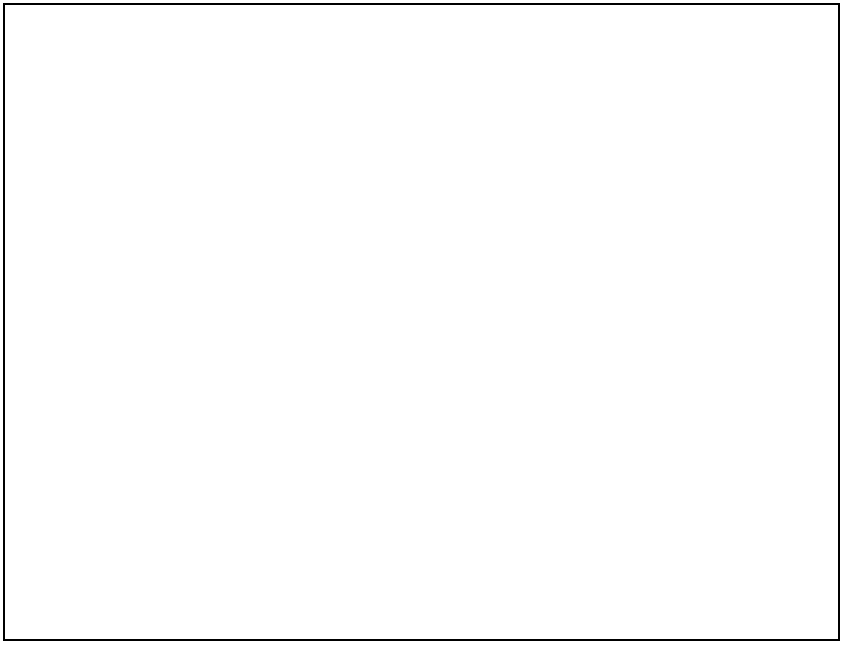

Рис. 1.13. Hex-редактор HTE — достойная замена HIEW

[4] Кстати, вместо них часто использовался редактор диска Norton Disk Editor.

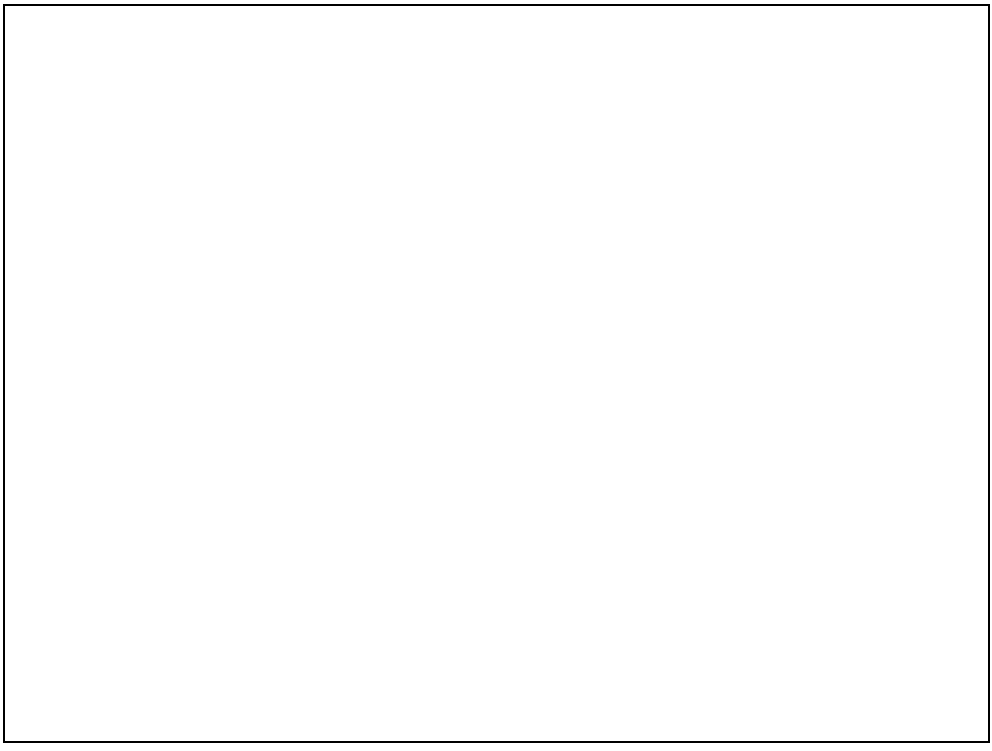

Рис. 1.14. Коммерческий hex-редактор WinHex

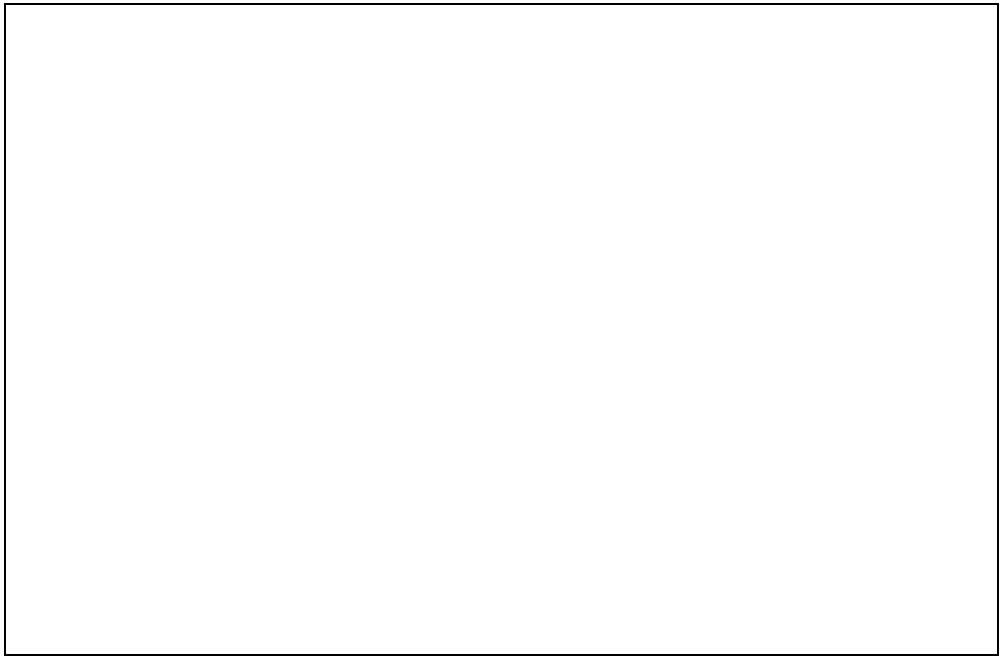

Рис. 1.15. Коммерческий hex-редактор Hex Workshop

Другой хороший редактор, по своим возможностям не только не уступающий HIEW, но даже превосходящий его, — это HTE (**http://hte.sourceforge.net**), распространяющийся в исходных кодах на бесплатной основе. В отличие от HIEW, HTE позволяет выбирать способ ассемблирования инструкции (если инструкция может быть ассемблирована несколькими путями), а также поддерживает мощную систему перекрестных ссылок, вплотную приближающую его к бесплатной версии IDA Pro (рис. 1.13).

Западные хакеры очень любят коммерческие шестнадцатеричные редакторы вроде WinHex (**http://www.winhex.com/winhex/index-m.html**) и Hex Workshop (**http://www.bpsoft.com**). Что привлекательного они в них нашли — непонятно. Ни WinHex (рис. 1.14), ни Hex Workshop (рис. 1.15) не содержат никаких ассемблеров или дизассемблеров, причем маловероятно, что эти функции появятся в дальнейшем. Единственное положительное качество этих редакторов, которое можно отметить, — это наличие калькулятора контрольных и хэш-сумм (например, CRC16, CRC32, MD5, SHA-1), что в некоторых случаях оказывается очень удобным.

Распаковщики

Все больше и больше программ распространяются в упакованном виде (или защищаются протекторами, что еще хуже). Как результат, непосредственное дизассемблирование таких программ становится невозможным. Наконец, поскольку многие упаковщики/протекторы содержат антиотладочные приемы, то страдает и отладка.

Попытки создать универсальный распаковщик многократно предпринимались еще со времен MS-DOS. Всякий раз эти попытки оканчивались полным провалом, поскольку разработчики защит придумывали новую гадость. Тем не менее, в состав большинства хакерских инструментов (IDA Pro, OllyDbg) входят универсальные распаковщики, справляющиеся с несложными защитами. Что касается сложных защит, то, столкнувшись с одной из них, хакер вынужден распаковывать защищенный файл вручную. В *части V* данной книги этот вопрос будет рассмотрен более подробно. Пока же отметим, что когда же один и тот же упаковщик встречается хакеру десятый раз кряду, он садится за написание автоматического или полуавтоматического распаковщика, чтобы облегчить себе работу. Коллекции таких утилит собраны на сайтах **http://www.exetools.com/unpackers.htm**, **http://programmerstools.org/taxonomy/term/16**, **http://www.woodmann.com/crackz/Packers.htm**. Основная проблема состоит в том, что каждый такой распаковщик рассчитан на строго определенную версию упаковщика/протектора и с другими работать просто не может! Чем чаще обновляется упаковщик/протектор, тем сложнее найти подходящий распаковщик, поэтому лучше полагаться только на самого себя.

Кстати, прежде чем искать распаковщик, неплохо бы для начала выяснить: чем же вообще защищена ломаемая программа? В этом поможет бесплатная утилита PEiD (**http://peid.has.it**), содержащая огромную базу сигнатур. Хотя эта программа довольно часто ошибается или дает расплывчатый результат, но, тем не менее, это все-таки лучше, чем совсем ничего.

Дамперы

Снятие дампа с работающей программы — универсальный способ распаковки, позволяющий справиться практически с любым упаковщиком и большинством протекторов. Правда, над полученным дампом еще предстоит как следует поработать, поэтому такие дампы рекомендуется использовать лишь для дизассемблирования. Сломанная таким путем программа может работать неустойчиво, периодически падая в самый ответственный момент.

Какие же дамперы имеются в вашем распоряжении? Первой из утилит этого класса (и самой неуклюжей) была программа ProcDump (**http://www.fortunecity.com/millenium/firemansam/962/html/procdump.html**). Затем появился дампер Lord PE (**http://www.softpedia.com/get/Programming/File-Editors/LordPE.shtml**), учитывающий горький опыт своего предшественника и способный

сохранять дамп даже в тех случаях, когда заголовок файла PE умышленно искажен защитой, а доступ к некоторым страницам памяти отсутствует (атрибут `PAGE_NOACCESS`). Венцом эволюции стал дампер PE Tools (рис. 1.16), базовый комплект поставки которого можно найти практически на любом хакерском сервере, например, на WASM (**http://www.wasm.ru/baixado.php?mode=tool&id=124**) или на CrackLab (**http://www.cracklab.ru/download.php?action=get&n=MTU1**), а свежие обновления лежат на "родном" сайте проекта **http://petools.org.ru/petools.shtml**.

Примечание

"Родной" сайт проекта часто меняет свой адрес.

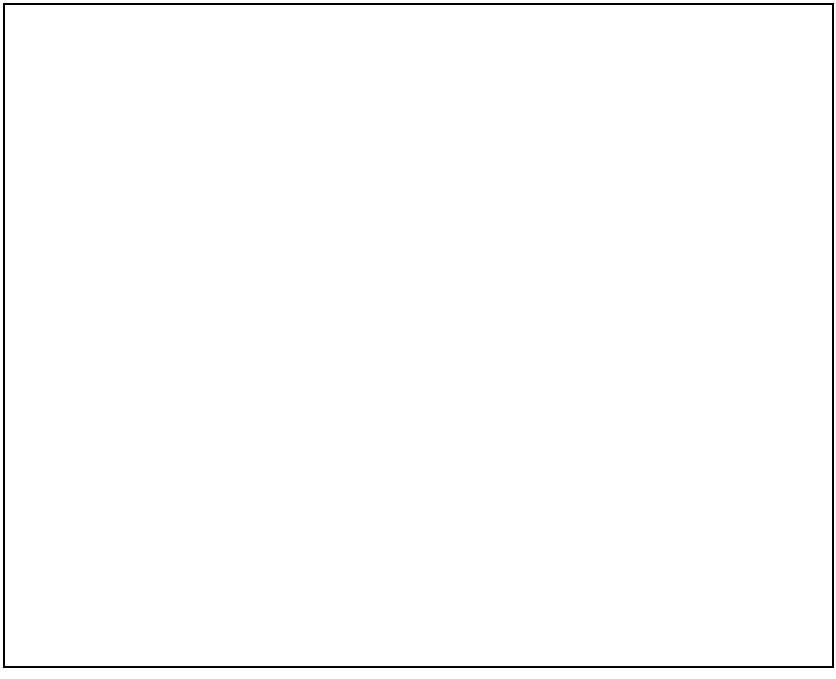

Рис. 1.16. PE Tools — один из лучших дамперов PE-файлов

После снятия дампа необходимо, как минимум, восстановить таблицу импорта, а иногда еще и таблицу перемещаемых элементов вместе с секцией ресурсов. Таблицу импорта лучше всего восстанавливать знаменитой утилитой Import REConstructor, которую вместе с утилитой ReloX (восстанавливающей таблицу перемещаемых элементов) и минимально работающим универсальным распаковщиком можно найти по адресу: **http://wave.prohosting.com/mackt/main.htm**. А вот здесь лежит коллекция программ для восстановления таблицы ресурсов: **http://www.wasm.ru/baixado.php?mode=tool&id=156**. Если же ни одна из этих утилит не справится со своей задачей, то, быть может, поможет бесплатная программа Resource Binder: **http://www.setisoft.com/ru/redirect.php?dlid=89**.

Редакторы ресурсов

Редактировать ресурсы приходится во многих случаях. Например, чтобы сменить текст диалогового окна, разблокировать элемент управления, заменить логотип и т. д. Формально, редактор ресурсов входит в каждый Windows-компилятор, в том числе и в Microsoft Visual Studio. Вот только

после редактирования ресурсов файл зачастую становится неработоспособным! Это происходит потому, что штатный редактор ресурсов к таким задачам неприспособлен!

Лучим хакерским редактором был и остается коммерческий Restorator Resource Editor (**http://www.bome.com/Restorator**), который может практически все, что нужно, и даже чуточку больше (рис. 1.17). Из бесплатных утилит в первую очередь хотелось бы отметить XN Resource Editor (**http://www.wilsonc.demon.co.uk/d10resourceeditor.htm**), написанный на Delphi и распространяющийся в исходных текстах, что позволяет наращивать функциональные возможности программы.

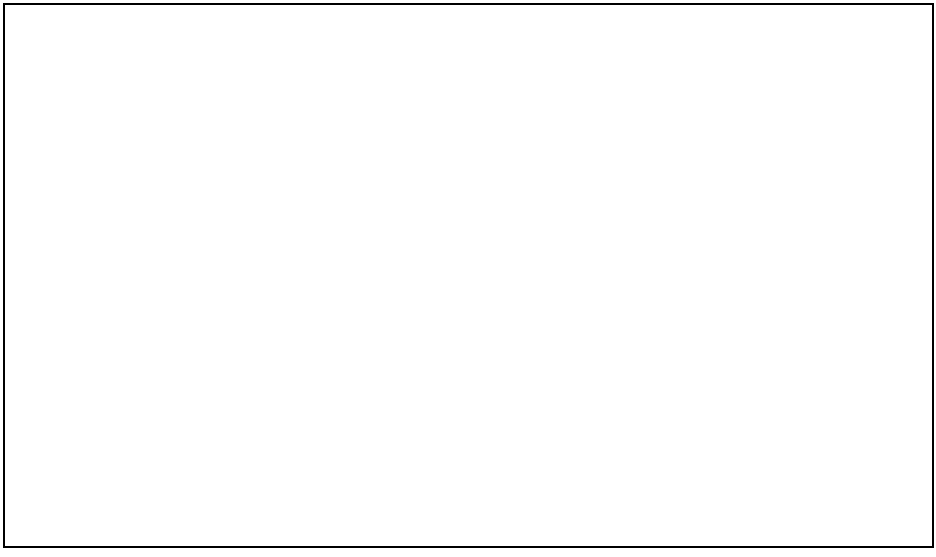

Рис. 1.17. Редактирование ресурсов в Restorator Resource Editor

Шпионы

В основном используется два типа шпионов — шпионы Windows-сообщений и API-шпионы. Первые следят за посылкой сообщений окнам и элементам управления, вторые — за вызовом функций API, включая функции, экспортируемые динамическими библиотеками, поставляемыми вместе с программой. Шпионаж — лучшее (и наиболее дешевое — в смысле усилий и времени) средство, позволяющее узнать, чем "дышит" защищенная программа.

Вполне достойный шпион сообщений, называющийся Spyxx.exe, входит в штатную поставку Microsoft Visual Studio. Аналогичный по возможностям шпион, но только с открытыми исходными текстами, лежит на **http://www.catch22.net/software/winspy.asp** и совершенно бесплатен (рис. 1.18).

Из API-шпионов лучшим является Kerberos (**http://www.wasm.ru/baixado.php?mode=tool&id=313**) от Рустема Фасихова, который взялся за клавиатуру тогда, когда остальные шпионы перестали его устаивать (рис. 1.19). Тем не менее, о вкусах не спорят, и многие программисты пользуются утилитой APISpy32 (такой же бесплатной, как и Kerberos), которую можно раздобыть на **http://www.internals.com**. Впрочем, и любой нормальный отладчик (например, SoftICE, OllyDbg) можно настроить так, чтобы он выполнял функции API-шпиона, причем действуя по весьма избирательному шаблону, избавляющему нас от просмотра многокилометровых листингов, генерируемых Kerberos и APISpy32. О том, как этого добиться, будет подробнее рассказано в *главе 10*.

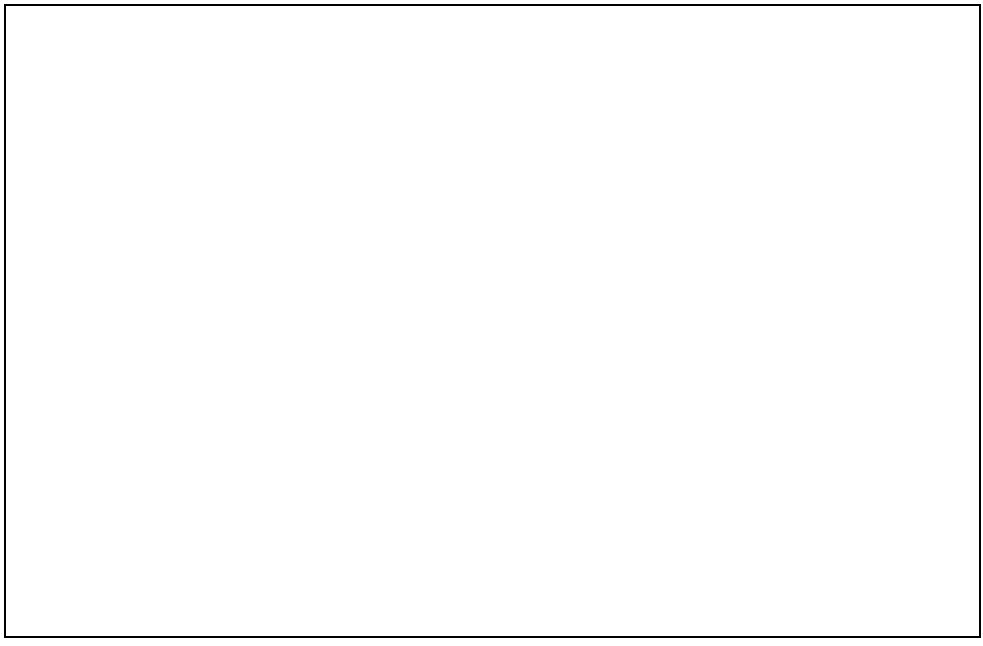

Рис. 1.18. Шпионаж за Windows-сообщениями при помощи бесплатной утилиты WinSpy

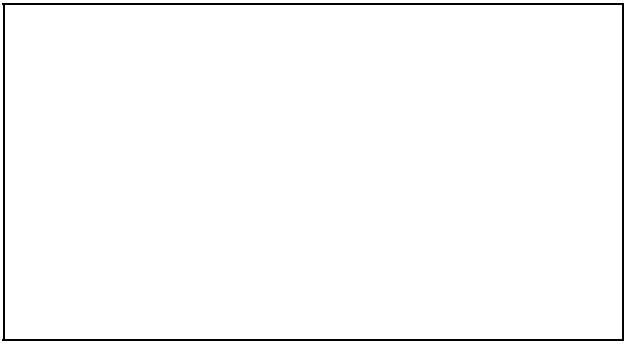

Рис. 1.19. API-шпион Kerberos от Рустема Фасихова

Мониторы

Чтобы узнать, к каким файлам или ветвям реестра обращается подопытная программа, достаточно воспользоваться файловым монитором Filemon.exe (рис. 1.20) и монитором реестра Regmon.exe (рис. 1.21), соответственно.

Обе утилиты были написаны легендарным исследователем недр Windows Марком Руссиновичем и долгое время распространялись вместе с исходными кодами совершенно бесплатно через некоммерческий сайт **http://www.sysinternals.com**. Однако в июне 2006 основатели этого сайта Марк Руссинович (Mark Russinovich) и Брюс Когсвелл (Bryce Cogswell) стали сотрудниками Microsoft. Хотя их утилиты и обещают остаться бесплатными и впредь, скорее всего, они будет бесплатными *только для легальных пользователей Windows*, так что спешите их скачать, пока это возможно (**http://www.microsoft.com/technet/sysinternals/default.mspx**).

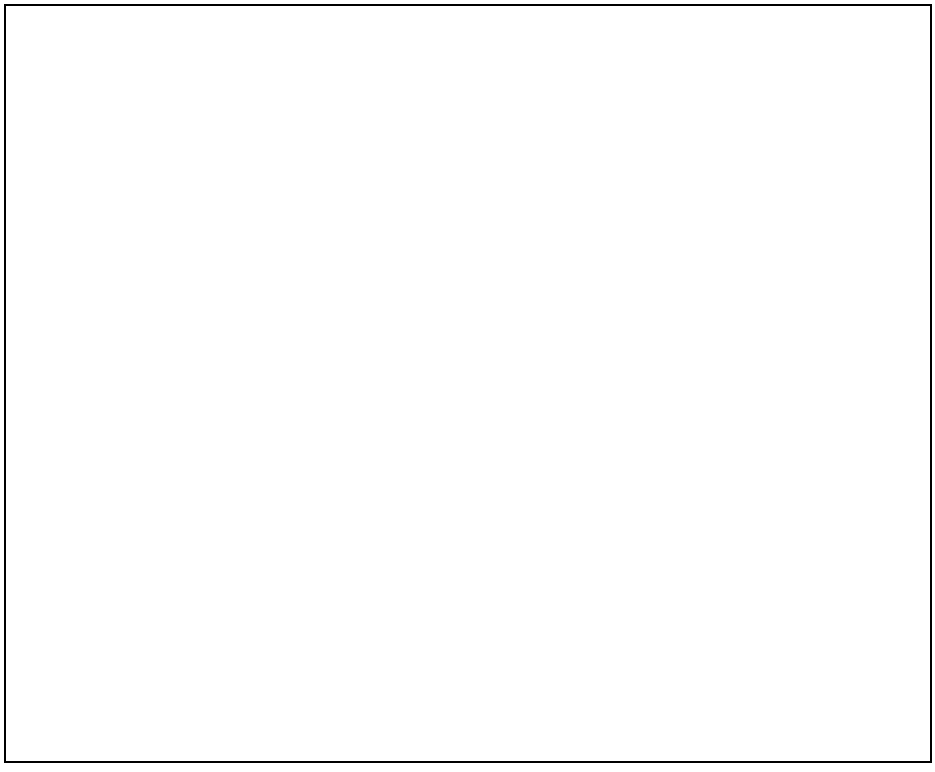

Рис. 1.20. Файловый монитор Марка Руссиновича за работой

Рис. 1.21. Монитор реестра Марка Руссиновича

Модификаторы

Существует два диаметрально противоположных подхода к взлому программ. Самое трудное (но при этом самое идеологически правильное и наименее всего наказуемое) — это создание своих собственных генераторов серийных номеров, ключевых файлов и т. д. Проанализировав, как работает оригинальный генератор, хакер пишет точно такой же. Однако это слишком утомительно, тем более что большинство защит нейтрализуются правкой нескольких байт. Вот только распространять взломанный файл нельзя. За это могут и по лапкам дать. К тому же, как правило, исполняемые файлы и DLL слишком громоздки, поэтому возникает естественная идея — распространять не сам взломанный файл, а список байтов с адресами, которые требуется исправить. Разумеется, рядовой пользователь не будет исправлять программу с помощью HIEW, поэтому на помощь приходит автоматизация.

Получить список различий между оригинальным и взломанным файлами поможет утилита fc.exe, входящая в штатный комплект поставки Windows. Для внесения исправлений в исполняемый файл или DLL потребуется утилита-модификатор, которую можно написать буквально за несколько минут. Коллекция уже готовых к использованию модификаторов находится здесь: **http://www.wasm.ru/baixado.php?mode=tool&id=35**.

Ситуация осложняется, если программа упакована или защищена протектором. В этом случае ее приходится править уже на лету, непосредственно в оперативной памяти. Для этой цели пригодятся утилиты Process Patcher (**http://www.wasm.ru/baixado.php?mode=tool&id=38**), R!SC's Process Patcher (**http://www.wasm.ru/baixado.php?mode=tool&id=39**) или *ABEL* Self Learning Loader Generator (**http://www.wasm.ru/baixado.php?mode=tool&id=144**). Последняя программа отличается тем, что ищет исправляемые байты не по фиксированным смещениям, а по регулярным шаблонам, что позволяет ей в большинстве случаев переживать выход новой, слегка измененной версии ломаемой программы (если, конечно, изменения не затронули сам защитный механизм).

Копировщики защищенных дисков

Копировать защищенные диски — это совсем не по-хакерски, а гораздо более по-пиратски. Тем не менее, заниматься этой деятельностью периодически приходится всем, так что лишними эти программы не будут.

Пара лучших коммерческих копировщиков это, бесспорно, Alcohol 120% (**http://www.alcohol-soft.com**) и CloneCD (**http://www.slysoft.com/en/clonecd.html**). Бесплатная утилита Daemon Tools (**http://www.daemon-tools.cc**) позволяет монтировать образы, снятые двумя этими копировщиками как виртуальные диски, образы которых лежат на HDD. Очень удобно!

Глава 2

Эмулирующие отладчики и эмуляторы

Еще несколько лет назад основными хакерскими средствами были дизассемблеры и отладчики. Теперь же к ним добавляются еще и *эмуляторы*, открывающие перед кодокопателями поистине безграничные возможности, ранее доступные только крупным компаниям, а теперь появившиеся в арсенале любого исследователя. Что же представляют собой эмуляторы и какие именно возможности они открывают?

Вводная информация об эмуляторах

В общем виде, *эмуляцию* можно определить как способность программы или устройства имитировать работу другой программы или другого устройства. Например, в эпоху перехода с 8-битных компьютеров наподобие ZX Spectrum на серьезный (по тем временам) IBM PC XT/AT, ностальгирующие разработчики, не желавшие расставаться со старой техникой, писали эмуляторы. Эмуляторы появлялись как грибы после дождя и позволяли всем желающим поиграть в любимые игры, прямых аналогов для которых на IBM PC не существовало.

Через десять лет история повторилась: операционные системы семейства Windows NT, ставшие стандартом де-факто, заблокировали прямой доступ к оборудованию, и 90% игр тут же отказались запускаться (или теряли звуковое сопровождение). Ответом на это стало появление DOSBox и других эмуляторов, позволявших пользователям обратить время вспять и вспомнить свою былую молодость, прошедшую в окровавленных коридорах DOOM II. Однако на P-III 733 МГц даже старый добрый Aladdin шел на пределе, с пропуском кадров, не говоря уже о более серьезных играх. Наращивать мощность было невыгодно — проще было купить старый компьютер, водрузить на него MS-DOS и наслаждаться играми без внезапных вылетов эмулятора, рывков и тормозов.

Идея эмулировать IBM PC на самом IBM PC родилась не вдруг и не сразу, но тут же завоевала расположение хакеров, системных администраторов и просто любопытствующих.

Исторический обзор

Вслед за возникновением концепции многозадачности, появилась идея виртуализации, выдвинутая в 1974 году Джеральдом Попеком (Gerald Popek) совместно с Робертом Голдбергом (Robert Goldberg) и описанная ими в статье *"Formal Requirements for Virtualizable Third Generation Architectures"* (**http://doi.acm.org/10.1145/361011.361073**).

Чтобы запускать несколько операционных систем одновременно, процессор должен поддерживать виртуальную память и обладать раздельными уровнями привилегий, согласованными с набором команд. Программа, называемая Монитором виртуальных машин (Virtual Machine Monitor, VMM), запускается на наивысшем уровне привилегий, перехватывает выполнение привилегированных инструкций (пытающихся обратиться, например, к физической памяти или портам ввода/вывода)

и эмулирует их выполнение. Непривилегированные инструкции в этой схеме выполняются на "живом" железе без потери скорости. При этом, поскольку процент привилегированных инструкций относительно невелик, накладными расходами на эмуляцию можно полностью пренебречь.

Из всех существовавших на тот момент компьютеров, данным критериям отвечали только IBM System/370 и Motorola MC68020. Именно на них и были реализованы эффективные эмуляторы, позволяющие, в частности, создавать виртуальные серверы, обслуживающие разных пользователей или просто дублирующие друг друга. В случае внезапного сбоя одного из серверов, инициативу тут же подхватывает другой, причем для этого совершенно не требуется держать несколько физических серверов! Отметим, что выход из строя самого сервера не рассматривается, поскольку сбои в работе операционных систем происходят значительно чаще, чем отказы оборудования.

А как же процессоры семейства Intel 80386? Они поддерживают виртуальную память и разделение привилегий по целым четырем кольцам защиты. Что мешает реализовать на них полноценный эмулятор, такой же как и на IBM System/370? Увы, компьютеры класса "Big Iron" потому и ценятся, что их проектировщики подошли к своей задаче тщательно и продуманно! А вот об IBM PC этого не скажешь. Начнем с того, что все операционные системы, какие только имеются на IBM PC, помещают свои ядра в нулевое кольцо (ring 0), на которое понятие привилегированных команд не распространяется. Следовательно, монитор виртуальных машин уже не может их перехватывать.

Правда, существует лазейка — устанавливаем некоторую операционную систему, объявляя ее базовой или основной (host), и запускаем все остальные ОС в третьем кольце (ring 3). Тогда привилегированные инструкции будут генерировать исключения, легко перехватываемые монитором виртуальных машин, работающим в нулевом кольце. Однако даже здесь не все так легко, как кажется!

Инструкции `LGDT`, `LLDT` и `LIDT`, загружающие во внутренние регистры процессора указатели на глобальные и локальные таблицы дескрипторов сегментов и прерываний, совершенно не приспособлены для одновременной работы с несколькими операционными системами, поскольку таблицы GDT (Global Descriptor Table), LDT (Local Descriptor Table) и IDT (Interrupt Descriptor Table) существуют в единственном экземпляре. Таблицы дескрипторов сегментов хранят линейные адреса и атрибуты каждого из сегментов, причем дескриптор представляет собой 16-разрядное число, загружаемое в сегментный регистр `CS`, `DS`, `SS` и т. д. С таблицей дескрипторов прерываний — та же самая картина. Гостевая ОС не может использовать "хозяйские" таблицы дескрипторов по той простой причине, что селекторы сегментных регистров жестко прописаны внутри самой ОС. Таким образом, если Windows загружает в регистр `DS` селектор `23h` (а она действительно это делает), то становится непонятно, что делать всем остальным операционным системам.

Существует единственный выход — для каждой из виртуальных машин создавать отдельную копию таблиц дескрипторов, переключая их при переходе с одной виртуальной машины на другую, что серьезно сказывается на производительности. Но это еще что! Инструкции `SGDT`/`SLDT` и `SIDT`, считывающие значения внутренних регистров процессора, не являются привилегированными, в результате чего гостевая ОС читает таблицу дескрипторов основной операционной системы вместо своей собственной! То же самое относится и к инструкции `SMSW`, считывающей значение командного слова процессора, в которое, в частности, попадают биты из регистра `CR0`, который невидим для гостевой ОС, работающей в третьем кольце.

Инструкции `POPF`/`POPFD` сохраняют содержимое регистра `EFLAGS` в памяти без генерации исключения, и хотя попытка модификации привилегированных полей `EFLAGS` приводит к исключению, это не спасает ситуацию. Допустим, гостевая ОС заносит в регистр `EFLAGS` некое значение `X`, затрагивающее одно или несколько привилегированных полей. Процессор генерирует исключение, эмулятор перехватывает его и имитирует запись, "подсовывая" гостевой системе "виртуальный" `EFLAGS`. Однако чтение `EFLAGS`, не являясь привилегированной инструкцией, возвращает его немодифицированное содержимое, поэтому вместо ожидаемого значения `X` гостевая ОС получит совершенно другое значение.

С инструкциями LAR, LSL, VERR и VERW дела обстоят еще хуже, поскольку они по-разному работают в привилегированном и непривилегированном режимах. В непривилегированном режиме исключение не генерируется, но инструкция возвращает совсем не тот результат, которого от нее ожидали.

Вот лишь неполный перечень причин, делающих платформу x86 непригодной для эффективной виртуализации (подробнее об этом можно прочитать в статье *"Proceedings of the 9th USENIX Security Symposium"*, доступной по адресу **http://www.usenix.org/events/sec2000/robin.html**).

Непригодность архитектуры x86 для эффективной виртуализации еще не запрещает эмулировать IBM PC программно с минимальной поддержкой со стороны оборудования. В грубом приближении это будет интерпретатор, "переваривающий" машинные команды с последующей имитацией их выполнения. С процессорами 8086 никаких проблем не возникает, но вот эмуляция страничной организации памяти и прочих функциональных возможностей, характерных для процессоров 386+, не только усложняет кодирование, но и снижает скорость выполнения программы в сотни или даже тысячи раз!

Области применения эмуляторов

Несмотря на недостатки, перечисленные в предыдущем разделе, интерес к программным эмуляторам постоянно растет. Это происходит по целому ряду причин. Например, специалисты в области сетевой безопасности должны иметь в своем распоряжении не менее трех различных операционных систем: по крайней мере одну систему из семейства Windows NT, один из вариантов Linux, а также одну из версий FreeBSD; причем наличие других популярных операционных систем также весьма желательно. Многие уязвимости (в частности, ошибки переполнения) проявляются только в конкретных версиях операционных систем, в то время как другие системы или даже другие версии одной и той же операционной системы могут быть свободны от этого недостатка. Но только подумайте, как же неудобно постоянно переустанавливать операционные системы, отказываясь от уже привычных и обжитых. Помимо колоссальных потерь времени (а время всегда играет не в вашу пользу), существует еще и риск потери накопленных данных, представляющих для вас большую ценность!

Эксперименты с вирусами и эксплоитами также должны выполняться на отдельно стоящем компьютере, изолированном от внешнего мира, так как система контроля доступа, встроенная в операционные системы семейства Windows NT и клоны UNIX, далека от совершенства. Следовательно, любая небрежность, допущенная исследователем, может привести к катастрофическим последствиям.

Традиционно эти проблемы решались за счет приобретения нескольких компьютеров или множества жестких дисков, которые подключались к тестовому компьютеру поочередно. Первое решение является слишком дорогим, а второе — крайне неудобным, особенно если учесть, что жесткие диски крайне чувствительны к такому обращению.

С этой точки зрения, трудно не согласиться с тем, что эмуляторы в такой ситуации — чрезвычайно полезные средства. При этом данная глава не "рекламирует" какой-либо отдельный эмулятор. Напротив, она описывает задачи, которые могут быть решены с помощью эмуляторов, и демонстрирует области их применения. Ее основная цель — помочь читателям выбрать эмуляторы, наиболее подходящие для практического решения стоящих перед ними задач.

Эмуляторы для пользователей

Представьте себе ситуацию: вы прочли в журнале статью о замечательной новой игре. Вы с энтузиазмом начали искать эту игру, раздобыли ее и внезапно обнаруживаете, что под вашей операционной системой она работать не будет. Какое разочарование! Пользователи FreeBSD в этом отношении находятся в наихудшей ситуации, поскольку игр для этой операционной системы

крайне мало. Как выйти из этой ситуации? На диске есть достаточно свободного пространства для установки Windows, но перезагружаться каждый раз, как только вам захочется поиграть — нет уж, увольте! А что, если эта игра предназначена для Mac или для Sony PlayStation? К счастью, современные компьютеры позволяют вам забыть о "родном железе", эмулируя компьютер целиком (рис. 2.1) и открывая пользователю безграничный мир программного обеспечения. Теперь вы не привязаны к конкретной аппаратной платформе и можете запускать любую программу, независимо от того, для какого компьютера она была написана — это может быть, например, ZX Spectrum или Xbox. Единственная проблема заключается в поиске качественного эмулятора.

При использовании эмуляции, основная операционная система превращается в "фундамент", на который можно надстраивать множество гостевых операционных систем. Одну из "комнат" в этом "отеле" рекомендуется отвести в качестве "карантинного помещения". Как известно, при установке любой новой программы существует риск краха операционной системы вследствие некорректно работающего инсталлятора, конфликта библиотек, вредоносного программного обеспечения или просто обычного невезения. Именно поэтому программы, полученные из ненадежных источников, и рекомендуется устанавливать и запускать в изолированной среде. Для этого можно выделить в эмуляторе отдельную виртуальную машину.

Примечание

Хотя данный совет хорош, следует сразу же отметить, что и он не обеспечивает полной безопасности. Способы вырваться за пределы виртуальной машины все же существуют. Некоторые из них будут подробно рассмотрены в *главе 41*.

Рис. 2.1. Хороший эмулятор позволит запустить любую операционную систему

Эмуляторы для администраторов

Для администраторов эмулятор — это в первую очередь полигон для всевозможных экспериментов. Поставьте себе десяток различных клонов UNIX и издевайтесь над ними по полной программе. Устанавливайте систему, сносите ее и снова устанавливайте, слегка подправив конфигурацию. На работу ведь принимают не по диплому, а по специальности, а специальность приобретается только в боях. То же самое относится и к восстановлению данных. Без специальной подготовки Disk Editor на рабочей машине лучше не запускать, а Disk Doctor — тем более[1]. Нет никакой гарантии того, что он действительно "вылечит" диск, а не превратит его в винегрет. Короче говоря, эмулятор — это великолепный тестовый стенд, о котором раньше не приходилось даже мечтать (рис. 2.2).

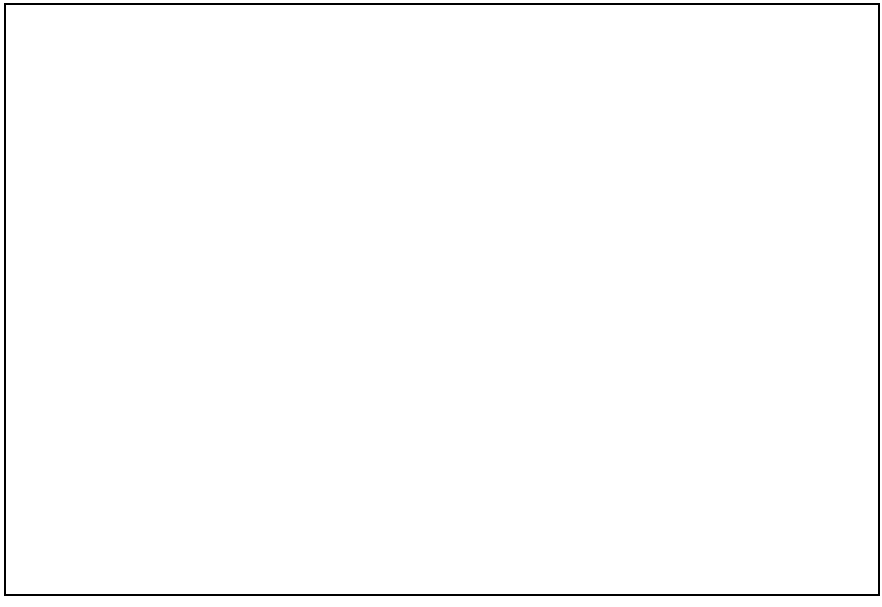

Рис. 2.2. Эмулятор — это своего рода тестовый стенд, позволяющий, например, приобрести навыки восстановления поврежденной файловой системы

В крупных организациях администратор всегда держит на резервном компьютере точную копию сервера и все заплатки сначала обкатывает на нем. В более мелких организациях добиться выделения отдельной машины специально для этой цели практически нереально, поэтому и приходится прибегать к эмулятору. На нем же тестируются различные эксплоиты, и если факт существования уязвимости подтверждается, принимаются оперативные меры по ее устранению.

Общение виртуальной машины с основной операционной системой и другими виртуальными машинами обычно осуществляется через виртуальную локальную сеть. При наличии 512—1024 Мбайт памяти можно создать настоящий корпоративный интранет — с SQL и WEB-серверами, демилитаризованной зоной, брандмауэром и несколькими рабочими станциями. Пример такой виртуальной сети показан на рис. 2.3. Лучшего полигона для обучения сетевым премудростям и не придумаешь. Хочешь — атакуй, хочешь — администрируй.

[1] Более подробно о причинах, по которым этого делать не следует, рассказано в следующей книге: К. Касперски, "Восстановление данных. Практическое руководство". — СПб.: БХВ-Петербург, 2007.

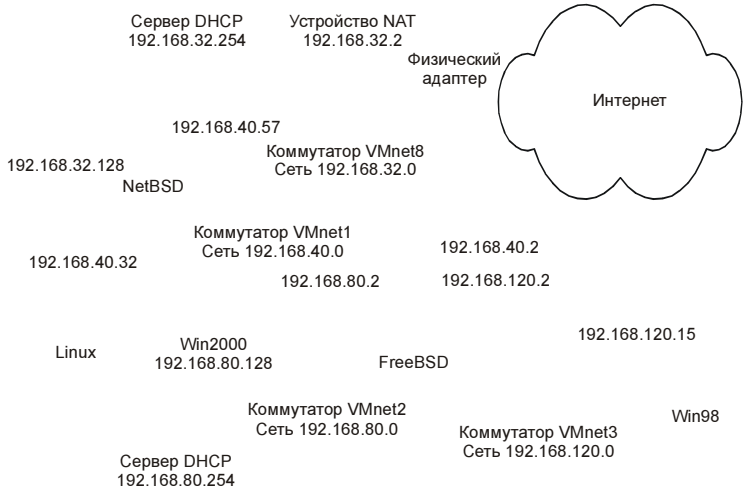

Рис. 2.3. Пример виртуальной сети

Эмуляторы для программистов

Больше всего эмуляторы любят разработчики драйверов. Ядро операционной системы не прощает ошибок и мстительно разрушает жесткий диск, уничтожая все данные накопленные за многие годы. Перезагрузки и зависания — вообще обычное дело, к которому привыкаешь как к стуку колес или шороху шин. К тому же, большинство отладчиков ядерного уровня требует наличия двух компьютеров, соединенных COM-кабелем или локальной сетью. Для профессионального разработчика это не роскошь, но… куда их ставить?

С эмулятором все намного проще. Ни потери данных, ни перезагрузок, а всю работу по отладке можно выполнять на одном компьютере. Естественно, что совсем уж без перезагрузок дело не обходится, но пока перезагружается виртуальная машина, можно делать что-то полезное на основной (например, править исходный код драйвера). К тому же мы можем заставить эмулятор писать команды в файл журнала, анализируя который можно определить причину сбоя драйвера (правда, не все эмуляторы предоставляют такую возможность).

В стандартном ядре FreeBSD отладчика нет, а отладочное ядро вносит в систему побочные эффекты. Поэтому в отладочном ядре драйвер может работать нормально, но вызывать крах системы в стандартном. Отладчики Windows ведут себя схожим образом, поэтому окончательное тестирование драйвера должно проходить в стандартной конфигурации, где разработчик лишен всех средств отладки и мониторинга.

Что касается прикладных программистов, то эмуляторы позволяют им держать под рукой всю линейку операционных систем, подстраивая разрабатываемые программы под особенности поведения каждой из них. В мире Windows существуют всего два семейства операционных систем — Windows 9*x* и линейка Windows NT, и даже в этих условиях у разработчиков часто голова кругом идет, а вот мир UNIX намного более разнообразен!

Коварство багов в том, что они склонны появляться только в строго определенных конфигурациях. Установка дополнительного программного обеспечения, а уж тем более перекомпиляция ядра может их спугнуть и тогда — ищи-свищи. А это значит, что до тех пор пока баг не будет найден, ничего нельзя менять в системе. На основной машине это требование выполнить затруднительно, но в эмуляторе это проблем не вызывает. Виртуальная машина, отключенная от сети (в том числе и виртуальной) в заплатах не нуждается. Но как же тогда обмениваться данными? К вашим услугам — дискета и CD-R.

Самое главное достоинство эмуляторов в том, что они позволяют создавать так называемые "слепки" — "моментальные снимки" состояния системы (system snapshots) и возвращаться к ним в любое время неограниченное количество раз. Это значительно упрощает задачу воспроизведения сбоя (т. е. определения обстоятельств его возникновения). Чем такой слепок отличается от дампа памяти, сбрасываемого системой при сбое? Как и следует из его названия, дамп содержит только информацию, которая на момент сбоя находилась в системной памяти, а "слепок" — все компоненты системы, в том числе содержимое диска, памяти, регистров контроллеров и т. д.

Разработчики сетевых приложений от эмуляторов вообще в полном восторге. Раньше ведь при отладке таких приложений был необходим второй компьютер, и довольно сложно было обойтись без помощника, которого предварительно требовалось еще и обучить. Теперь же отладка сетевых приложений упростилась до предела (рис. 2.4).

Рис. 2.4. Отладка прикладной программы под эмулятором

Эмуляторы для хакеров

Эмулирующие отладчики появились еще во времена MS-DOS и сразу же завоевали бешеную популярность у хакеров. Неудивительно! Ведь рядовые защитные механизмы применяют две основных методики для борьбы с отладчиками — пассивное обнаружение отладчика и активный захват отладочных ресурсов, делающий отладку невозможной. На эмулирующий отладчик эти

действия никак не распространяются — он находится ниже виртуального процессора, и потому для отлаживаемого приложения он совершенно невидим. Кроме того, эмулирующие отладчики не используют никаких ресурсов эмулируемого процессора.

Слепки системы очень помогают во взломе программ с ограниченным сроком использования. Ставим программу, создаем слепок, переводим дату, после чего делаем еще один слепок. Смотрим — что изменилось. Делаем выводы и "отламываем" от программы лишние запчасти. Простейший и общедоступный вариант этой методики выглядит так: устанавливаем защищенную программу на отдельную виртуальную машину. Делаем "слепок". Все! Защите пришел конец! Сколько бы мы ни запускали "слепок", защита будет наивно полагать, что запускается в первый раз. Не сможет она привязываться и к оборудованию, так как оборудование эмулятора не зависит от аппаратного окружения.

Попутно эмулятор освобождает от необходимости ставить ломаемую программу на основную машину. Во-первых, некоторые программы, обнаружив, что их ломают, пытаются этому противодействовать. Например, такая программа может затереть данные на жестком диске. В любом случае, даже если защита и не подстроит вам никакой пакости, программа наверняка будет работать нестабильно. Пусть уж она лучше глючит на эмуляторе!

Аппаратная виртуализация

К счастью, проблема низкой производительности эмуляторов, упомянутая в начале этой главы, уже отошла в прошлое. В середине 2006, пробиваясь на рынок мощных серверов, компании Intel и AMD разработали технологии аппаратной виртуализации. Фактически, они добавили дополнительное кольцо защиты, работая в котором *гипервизор*[2] может перехватывать все события, требующие внимания с его стороны. В практическом плане это существенно снизило накладные расходы на работу эмулятора, и теперь производительность виртуальных машин достигает порядка ~90% производительности основного процессора. Технология аппаратной виртуализации уже поддерживается новыми версиями эмуляторов. Например, такая поддержка обеспечивается эмуляторами VMware 5.5, Xen, а также целым рядом других продуктов, которые будут рассматриваться далее в этой главе.

Итак, что же вам понадобится для экспериментов с эмуляторами? Если проблема низкой производительности вас не смущает, то аппаратные требования будут достаточно скромными. Например, чтобы обеспечить комфортную работу с Windows 2000 и FreeBSD 4.5, вполне хватит процессора Pentium III 733-МГц. Это вполне позволит вам поиграть, например, в Quake I, хотя и на пределе возможностей.

Требования к оперативной памяти будут несколько более жесткими. Как правило, основной операционной системе необходимо оставить, как минимум, 128 Мбайт оперативной памяти, а каждой виртуальной машине (т. е. каждой гостевой операционной системе) выделить примерно 128—256 Мбайт. Естественно, объем требуемой памяти зависит от типа эмулируемой операционной системы. Например, для эмуляции MS-DOS будет достаточно и 4 Мбайт. Выделив виртуальной машине 256 Мбайт, вы сможете эмулировать Windows 2000/XP/2003.

Наличие свободного дискового пространства обычно не является вопросом первостепенной важности. Виртуальные машины создаются не для накопления и хранения данных. За редким исключением, они не содержат ничего, кроме типовой копии эмулируемой операционной системы и необходимого минимума приложений. Образ виртуального диска хранится в обычном файле, который находится под полным контролем основной операционной системы. Виртуальные диски

[2] Гипервизор (hypervisor) — программа (или аппаратная схема), позволяющая одновременно и параллельно выполнять на одном хост-компьютере несколько операционных систем. Термин "гипервизор" введен специально для того, чтобы подчеркнуть отличие от термина "супервизор" (supervisor), которым традиционно называли менеджер ресурсов ядра в эпоху мэйнфреймов.

бывают двух типов — фиксированные (fixed) и динамические, или так называемые "разреженные" (sparse). При создании *фиксированного виртуального диска*, эмулятор сразу же "распределяет" файл образа по всему выделенному дисковому пространству, даже если образ и не содержит никакой полезной информации. При создании *динамического виртуального диска*, напротив, в файле образа сохраняются только использованные виртуальные секторы, а размер образа увеличивается по мере того, как он заполняется данными.

Если же производительность эмулятора для вас важна, то ситуация обстоит иначе. Чтобы в полной мере использовать преимущества аппаратной виртуализации, вы должны иметь в своем распоряжении процессор, обеспечивающий поддержку этой технологии. Для этой цели подойдут следующие процессоры Intel: Pentium 4 6x2, Pentium D 9xx, Xeon 7xxx, Core Duo и Core 2 Duo (технология Vanderpool) и Itanium (технология Silvervale). Что касается процессоров AMD, то аппаратную виртуализацию поддерживают все процессоры, произведенные позднее мая 2006 (Socket AM2, Socket S1 и Socket F —Athlon 64, Turion 64), а также все процессоры AMD Opteron, выпущенные позднее августа 2006 (технология Pacifica). Официально технологии аппаратной виртуализации, реализованные Intel и AMD, называются VT-X и AMD-V, соответственно.

Естественно, кроме процессора вам потребуется современная материнская плата и, возможно, обновленная версия BIOS. Некоторые BIOS позволяют включать и выключать поддержку аппаратной виртуализации, причем в некоторых из них она по умолчанию выключена.

ОК, железо куплено, собрано, настроено и готово к работе. Теперь очередь за ПО.

Обзор популярных эмуляторов

Среди всех доступных эмуляторов наиболее популярны DOSBox, Bochs, Microsoft Virtual PC и VMware. Каждый из них обладает своими преимуществами, недостатками, и, естественно, у каждого из них есть свой круг поклонников.

DOSBox

Бесплатный эмулятор, распространяющийся в исходных текстах (**http://dosbox.sourceforge.net/download.php?main=1**). Эмулирует единственную операционную систему — MS-DOS 5.0. Применяется он, главным образом, для запуска старых игр. Жесткие диски не эмулируются (эмуляция дискового ввода-вывода заканчивается на прерывании `INT 21h`), и SoftICE на нем не работает. Зато cup386 (распаковщик исполняемых файлов плюс отладчик), скачать который можно по адресу **ftp://ftp.elf.stuba.sk/pub/pc/pack/ucfcup32.zip**, работает вполне исправно (рис. 2.5). Кроме того, имеется неплохой интегрированный отладчик (правда, для этого эмулятор должен быть перекомпилирован с отладочными ключами).

Возможность расширения конструктивно не предусмотрена. Тем не менее, доступность хорошо структурированных исходных текстов делает эту проблему неактуальной. При желании, вы в любой момент сможете добавить к эмулятору любую недостающую функциональную возможность (например, виртуальный жесткий диск).

Поддерживаются три режима эмуляции — полная, частичная и динамическая. Полнота "полной" эмуляции на самом деле довольно условна. Например, отладчик SoftICE в этом эмуляторе не работает. Однако для подавляющего большинства неизвращенных программ с лихвой хватает и частичной эмуляции. Оба этих режима достаточно надежны, и вырваться за пределы эмулятора нереально, хотя производительность виртуальной машины оставляет желать лучшего — Pentium III 733 МГц опускается до 13.17 МГц, замедляясь более чем в 50 раз. Модуль динамической эмуляции (выполняющий код на "живом" процессоре) все еще находится в стадии разработки, и текущая версия содержит много ошибок, причем некоторые из них фатальны. По этой причине, пользоваться данным модулем не рекомендуется, несмотря даже на то, что его производительность вчетверо выше.

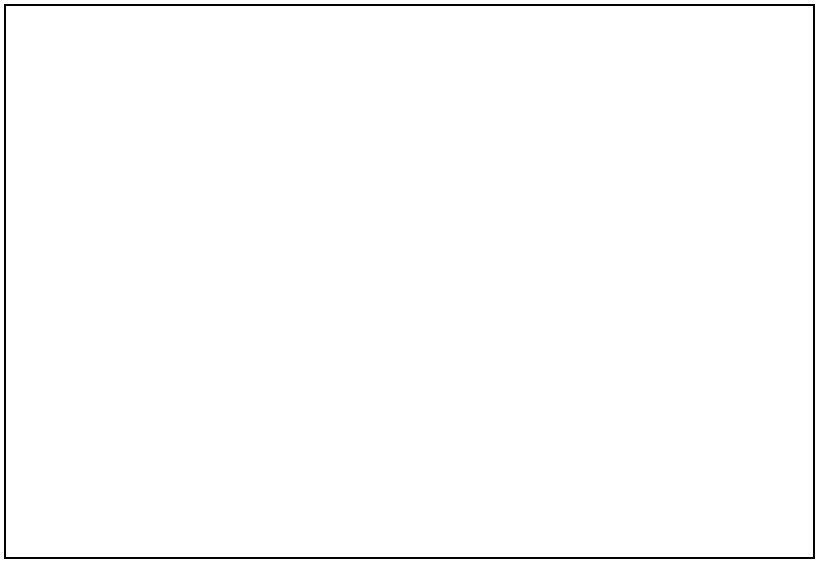

Рис. 2.5. Отладчик cup386, запущенный под управлением эмулятора DOSBox (непосредственно из-под Windows cup386 не запускается)

Обмен данными с внешним миром происходит либо через прямой доступ к CD-ROM, либо через монтирование каталогов физического диска на виртуальные логические диски, доступные из-под эмулятора через интерфейс `INT 21h`. Это обеспечивает достаточно надежную защиту от вредоносных программ. Уничтожить смонтированный каталог они смогут, но вот все остальные — нет!

DOSBox хорошо подходит для экспериментов с большинством MS-DOS-вирусов (исключая, пожалуй, лишь тех из них, что нуждаются в прерывании `INT 13` или портах ввода/вывода), а также для взлома программ, работающих как в реальном, так и защищенном режимах.

Bochs и QEMU

Bochs — это подлинно хакерский эмулятор, ориентированный на профессионалов (рис. 2.6). Простые смертные находят его слишком запутанным и непроходимо сложным. Здесь все настраивается через текстовые конфигурационные файлы — от количества процессоров до геометрии виртуального диска.

Это — некоммерческий продукт с открытыми исходными текстами и впечатляющим качеством эмуляции. Контроллеры гибких и жестких дисков IDE эмулируются на уровне портов ввода/вывода, обеспечивая совместимость практически со всеми низкоуровневыми программами. Полностью эмулируется защищенный режим процессора. Во всяком случае, SoftICE запускается вполне успешно (рис. 2.7), хотя и работает несколько нестабильно, периодически завешивая виртуальную клавиатуру. Имеется достаточно приличный интегрированный отладчик с неограниченным количеством виртуальных точек останова и функций обратной трассировки.

Эмулятор хорошо подходит для исследования вирусов и отладки извращенных программ, работающих в MS-DOS или терминальном режиме Linux/FreeBSD, а также для экспериментов с различными файловыми системами (см. рис. 2.2).

Полная версия исходного кода находится по адресу **http://bochs.sourceforge.net**. Здесь же можно найти и готовые к применению исполняемые файлы для Windows и Linux. К сожалению, чтобы запустить в этом эмуляторе Windows 2000, потребуется мощный современный процессор, и даже в этом случае производительность будет разочаровывающе низкой.

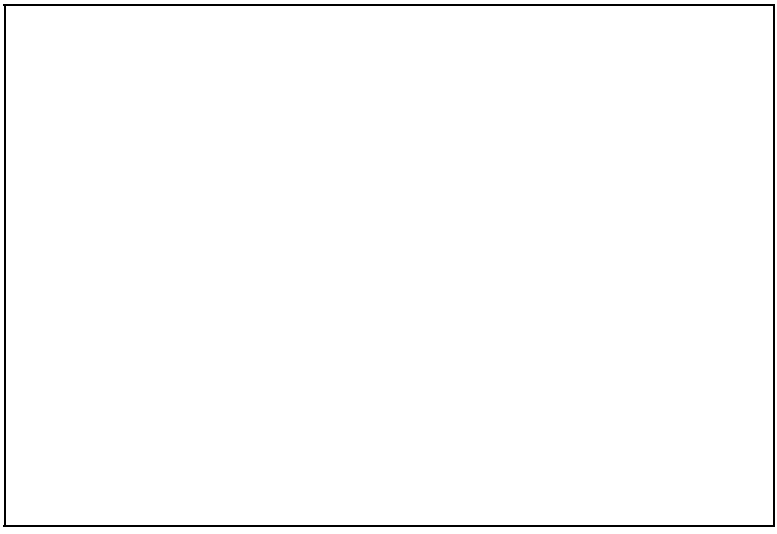

Рис. 2.6. Bochs позволяет запускать 64-разрядную версию Linux Debian на процессоре x86

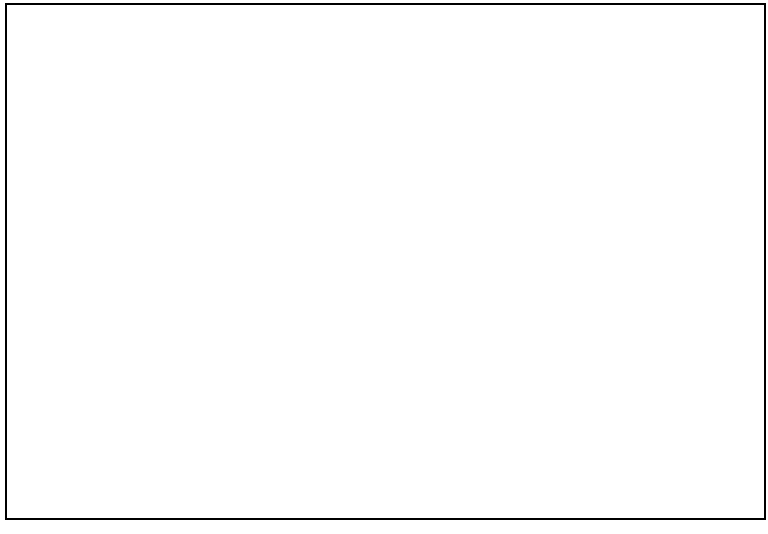

Рис. 2.7. Отладчик, запущенный в сеансе MS-DOS под управлением эмулятора Bochs, работающего под управлением Windows

На основе Bochs был создан еще один замечательный эмулятор — QEMU, использующий режим *динамической эмуляции*, который повышает производительность в десятки раз. Не вдаваясь в технические подробности, отметим, что QEMU как бы "компилирует" машинный код, который при повторном исполнении работает на "живом" железе на полной скорости. В циклах это дает колоссальный выигрыш!

Правда, общая производительность все равно оставляет желать лучшего, да и стабильность (в силу технических сложностей реализации динамической эмуляции) прихрамывает. Некоторые программы вообще не запускаются (особенно игры), некоторые периодически вылетают. Тем не менее, QEMU легко тянет Linux/BSD без графической оболочки, и ряды его поклонников неуклонно растут. Свежую версию всегда можно бесплатно скачать с **http://fabrice.bellard.free.fr/qemu**.

VMware

Осознав перспективность рынка высококачественных эмуляторов, в 1999 компания VMware выпустила революционный продукт под названием VMware Virtual Platform. Реализован он был на основе исследований, проведенных в Стэнфордском университете и запатентованных в мае 2002. Сам патент, а также вся необходимая техническая информация доступны для свободного скачивания по адресу **http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=6,397,242**.

Чтобы добиться высокой скорости эмуляции на процессорах архитектуры x86, разработчики VMware использовали ряд сложных технологий, требующих тесного взаимодействия с ядром основной операционной системы. К сожалению, все они налагают существенные ограничения на гостевые операционные системы. Полной виртуализации добиться не удалось. Эмулируются лишь некоторые функциональные возможности процессоров x86, в то время как попытки воспользоваться остальными приводят к непредсказуемому поведению гостевой операционной системы. Тем не менее, понижение производительности не слишком существенно, что позволяет успешно работать с системой Windows 2000 на процессоре Pentium III. Таким образом, VMware можно охарактеризовать как многоцелевой эмулятор, подходящий для разнообразных экспериментов, особенно если вы располагаете мощным и современным компьютером (рис. 2.8).

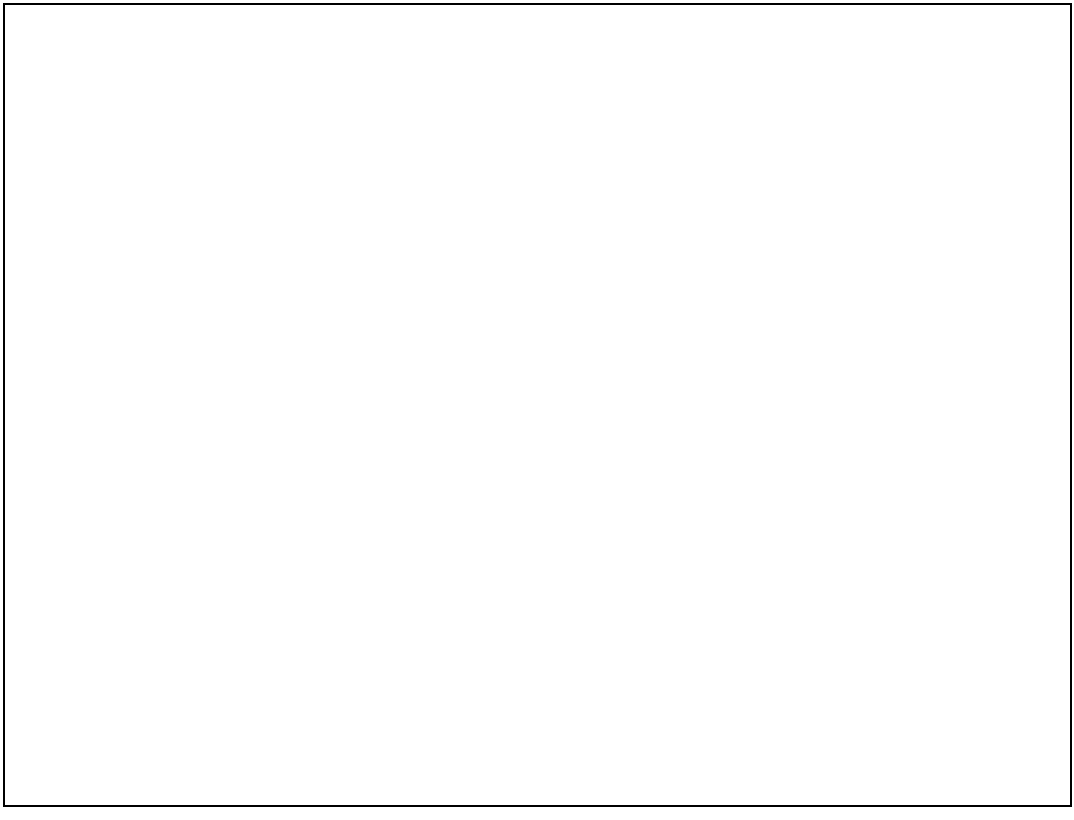

Рис. 2.8. Windows Vista, запущенная в эмуляторе VMware под Windows XP

Неоспоримым преимуществом VMware является устойчивая работа SoftICE (рис. 2.9), а также поддержка мгновенных снимков системы.

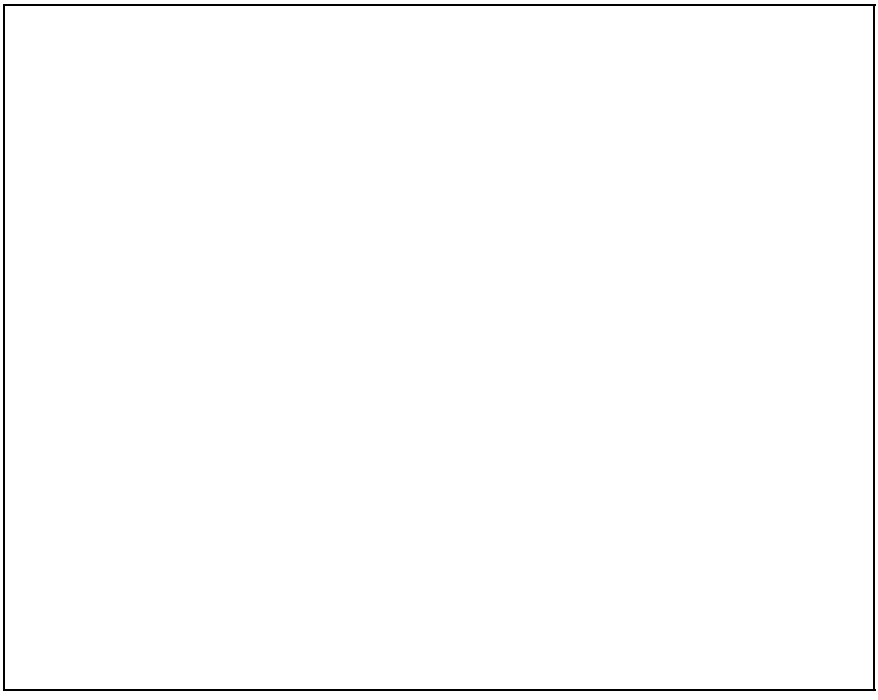

Рис. 2.9. SoftICE устойчиво работает с VMware

Как настроить SoftICE под VMware

При попытке использования SoftICE под Windows 2000, запущенной из-под VMware, вы можете столкнуться с проблемами. Суть проблемы в том, что SoftICE работает только из полноэкранного текстового режима (заходим в FAR Manager, нажимаем <ALT>+<ENTER>, затем <CTRL>+<D>), а во всех остальных режимах наглухо завешивает систему. Зато под Windows 98 он чувствует себя вполне нормально, но ведь переход на Windows 98 — не вариант.

Это — известная ошибка реализации, признанная NuMega и устраненная лишь в DriverStudio версии 3.1 (в официальной формулировке это именуется "поддержкой VMWARE"). Подробности можно найти в сопроводительной документации (см. \Compuware\DriverStudio\Books\Using SoftICE.pdf, приложение E — "SoftICE and VMware"). При этом в конфигурационный файл виртуальной машины (*имя_виртуальной_машины*.vmx) необходимо добавить строки `svga.maxFullscreenRefreshTick = "2"` и `vmmouse.present = "FALSE"`.

Тщательно спроектированная виртуальная сеть позволяет экспериментировать с сетевыми червями. Кроме того, имеется возможность прямого доступа к гибким/лазерным дискам. Реализованы разделяемые папки с достойной защитой.

Примечание

Заниматься разведением вирусов в недрах виртуальной машины следует с осторожностью, так как "скорлупа", отделяющая гостевую систему от реального мира, слишком тонка. Можно, конечно, запустить эмулятор в эмуляторе (например, Bochs внутри VMware), только это все равно не решит всех проблем, а вот производительность упадет колоссально!

Начиная с версии 5.5, VMware поддерживает технологию аппаратной виртуализации Vanderpool, позволяющую ей запускать 64-битные гостевые операционные системы на процессорах x86. Правда, для 32-битных гостевых операционных систем аппаратная виртуализация по умолчанию выключена, так как вследствие недостатков реализации вместо обещанного ускорения она дает

замедление. Подробное разъяснение причин такого поведения эмулятора можно найти в статье *"A Comparison of Software and Hardware Techniques for x86 Virtualization"*, написанной двумя сотрудниками VMware — Кейзом Адамсом (Keith Adams) и Олом Агесеном (Ole Agesen) и выложенной на сайте компании (**http://www.vmware.com/pdf/asplos235_adams.pdf**).

> ***Примечание***
>
> Заставить VMware использовать аппаратную виртуализацию в принудительном порядке поможет строка `monitor_control.vt32 = "TRUE"`, добавленная в *.vmx-файл соответствующей виртуальной машины. Правда, большой пользы это не принесет.

Последнюю версию VMware можно скачать с фирменного сайта **http://www.vmware.com**. Кроме коммерческих версий, VMware предоставляет и бесплатные эмуляторы: VMware Player и VMware Server, которые можно загрузить со страницы **http://www.vmware.com/products/free_virtualization.html**.

Несмотря на все очевидные преимущества VMware, этот эмулятор не является бесспорным лидером и не делает бесполезными все остальные виртуализационные продукты. Это особенно справедливо в отношении эмуляторов, имеющих встроенные отладчики и поставляемых с исходными кодами, что дает возможность неограниченно расширять их функциональные возможности.

Microsoft Virtual PC

Неплохой эмулятор, распространяющийся без исходных текстов, но зато обеспечивающий приличную скорость эмуляции, превращающую Pentium III 733 МГц в Pentium III 187 МГц (динамический режим эмуляции обеспечивает поддержку всех машинных команд физического процессора).

Полностью эмулируются AMI BIOS (с возможностью конфигурирования через Setup (рис. 2.10), чипсет Intel 440BX, звуковая карта типа Creative Labs Sound Blaster 16 ISA, сетевой адаптер DEC 21140A 10/100 и видеокарта S3 Trio 32/64 PCI с 8 Мбайт памяти на борту. В целом, эта конфигурация позволяет запускать современные операционные системы семейства Windows NT, а также FreeBSD с графическими оболочками.

Имеется возможность прямого доступа к гибким дискам и приводам CD-ROM. Жесткие диски эмулируются на уровне двухканального контроллера IDE (см. документацию на чипсет 440BX), размещаясь на винчестере в виде динамического или фиксированного файла-образа. При желании можно взаимодействовать с основной операционной системой и другими виртуальными машинами через разделяемые папки или виртуальную локальную сеть. Оба этих метода с хакерской точки зрения небезопасны, и потому при исследовании агрессивных программ к ним лучше не прибегать.

Virtual PC использует метод виртуализации, аналогичный использованному в VMware. Однако VMware обеспечивает лучшее качество эмуляции и более широкую поддержку аппаратных средств. В частности, отладчик SoftICE, отлично работающий под VMware, под Virtual PC работать отказывается (рис. 2.11). Встроенного отладчика и возможностей работать с моментальными снимками состояния виртуальной машины также нет. Все эти недостатки существенно ограничивают область применения данного эмулятора.

Однако игры, не требующие быстрого процессора и мощного видеоадаптера, под Virtual PC работают лучше, чем под VMware. Наконец, не может не радовать и то, что в июле 2006, корпорация Microsoft наконец-то пошла навстречу пожеланиям пользователей и выпустила бесплатную версию Virtual PC 2004, которую можно скачать по адресу **http://www.microsoft.com/windows/virtualpc/downloads/sp1.mspx**. Этот бесплатный продукт можно рекомендовать для экспериментов с файловыми системами, что поможет приобрести практические навыки восстановления утраченных данных.

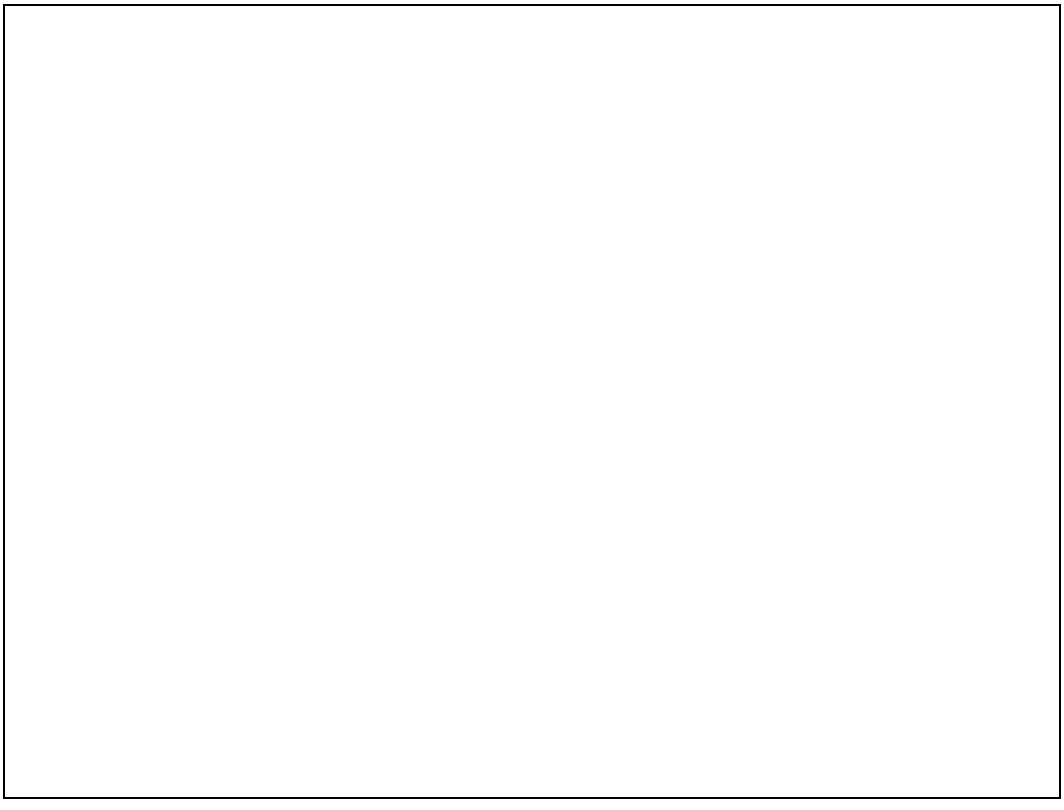

Рис. 2.10. Microsoft Virtual PC эмулирует весь компьютер целиком, включая BIOS Setup

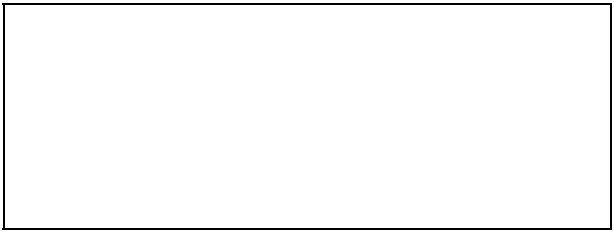

Рис. 2.11. Реакция Microsoft Virtual PC на попытку запуска SoftICE

Часто возникает довольно щекотливый вопрос о необходимости лицензирования Windows (и другого ПО) для каждой виртуальной машины, на которой она установлена. С юридической точки зрения все ОК, поскольку операционные системы до сих пор лицензируются под физические машины, что вполне логично, но вот мерзкая защита, встроенная в Windows, требует активации при смене всех трех ключевых компонентов — процессора, жесткого диска и видеокарты, а на виртуальной машине они… естественно виртуальные и совсем не совпадающие с реальными. Правда, VMware, выполняющая команду `CPUID` "вживую", показывает процессор таким, какой он есть, избавляя нас от необходимости платить за одну и ту же копию Windows много раз подряд.

Новые версии виртуализационного программного обеспечения Microsoft поддерживают технологии аппаратной виртуализации. Например, Microsoft Virtual Server 2007 поддерживает технологии Pacifica и Vanderpool.

Xen

Проект XEN — детище некоммерческой организации Xen Community, возглавляемой Яном Праттом (Ian Pratt) из XenSource, Inc (рис. 2.12) — появился задолго до "изобретения" аппаратной виртуализации. Он широко использовался такими компаниями, как IBM и Hewlett-Packard в мэйнфреймах для организации выделенных виртуальных серверов (virtual dedicated servers), подробнее о которых можно прочесть в статье **http://en.wikipedia.org/wiki/Virtual_dedicated_server**. Это, безусловно, говорит о высокой надежности и отличном качестве данного продукта. Он проверен временем и, что еще важнее, помимо платформы x86 поддерживает и такие платформы, как x86-64, IA64, PPC и SPARC.

Рис. 2.12. Гостевые операционные системы в эмуляторе Xen

Правда, на процессорах, не поддерживающих аппаратной виртуализации, требуется модификация ядра гостевой операционной системы, взаимодействующей с гипервизором посредством предоставляемого им набора функций API. С открытыми операционными системами (xBSD, Linux) в этом плане никаких проблем не возникает, а вот Windows XP удалось перенести на XEN исключительно в рамках проекта "Microsoft's Academic Licensing Program", позволяющего хачить ядро Windows в академических целях. Несмотря на то, что перенос осуществлялся при активном участии Microsoft Research в тесном сотрудничестве с группой University of Cambridge Operating Systems, условия лицензионного соглашения не позволяют распространять портированную версию Windows XP ни при каких условиях. Тем не менее, технические детали переноса подробно описаны в документации на XEN, так что при жгучем желании, помноженном на избыток свободного времени, этот фокус может повторить любая хакерская группа.

При поддержке аппаратной виртуализации со стороны процессора, XEN позволяет запускать гостевые системы без какой-либо их модификации (а XEN поддерживает все три технологии виртуализации: Pacifica, Vanderpool и Silvervale). Зачем ограничиваться только Windows XP, когда вокруг существуют Windows Vista, Server Longhorn и горячо любимая многими Windows 2000.

В роли базовой ОС могут выступать Linux, NetBSD или FreeBSD (последняя поддерживается в ограниченном режиме). XEN входит в состав множества дистрибутивов, в том числе и в Debian. Существуют и коммерческие версии XEN, например, Novell SLES10 или Red Hat RHEL5. Что же касается Windows, то она не входит в список базовых операционных систем, поддерживаемых XEN. Поэтому если вы выберете себе именно этот эмулятор, то устанавливать Linux/NetBSD вам все-таки придется. Собственно говоря, ничего страшного в этом нет, зато потом поверх него можно будет запустить множество гостевых Windows всех версий, какие только заблагорассудится.

Еще один вариант заключается в использовании образов LiveCD, которые доступны для свободного скачивания по адресу **http://www.xensource.com/download/dl_303cd.html**. Исходные коды Xen доступны здесь: **http://www.cl.cam.ac.uk/research/srg/netos/xen**.

Ближайшие конкуренты

Итак, вы убедились в том, что виртуализация — это передовая технология, позволяющая экспериментировать со всеми существующими операционными системами, организовывать виртуальные сети, а также моделировать атаки, одновременно наблюдая реакцию на них брандмауэров и систем обнаружения вторжений (intrusion detection systems, IDS). Неудивительно поэтому, что новые продукты этого класса появляются чуть ли не каждый день. Одним из таких продуктов является Parallels Workstation (рис. 2.13).

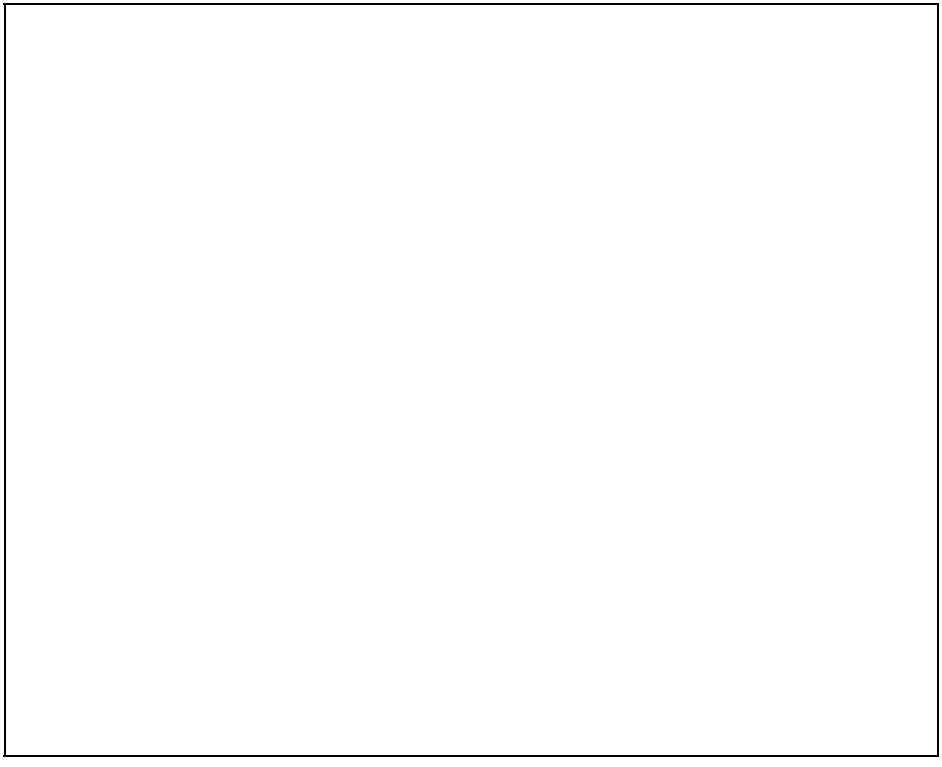

Рис. 2.13. Эмулятор Parallels Workstation

Наконец, нельзя не упомянуть и такие экзотические супервизоры, как Trango, ориентированные на решение задач реального времени, например, таких как обработка быстро меняющихся показаний датчиков.

Выбор подходящего эмулятора

При выборе подходящего эмулятора, хакеры обычно руководствуются следующими критериями: защищенностью, расширяемостью, открытостью исходных текстов, качеством и скоростью эмуляции, наличием встроенного отладчика и гибкостью механизмов работы с моментальными снимками. Рассмотрим все эти пункты более подробно.

Защищенность

Запуская агрессивную программу на эмуляторе, очень сложно отделаться от мысли, что в любой момент она может вырваться из-под его контроля, оставляя за собой длинный шлейф разрушений. Скажем прямо, эти опасения вполне обоснованны. Многие из эмуляторов (DOSBox, Virtual PC) содержат "дыры", позволяющие эмулируемому коду напрямую обращаться к памяти самого эмулятора (например, вызывать от его имени и с его привилегиями произвольные функции API основной операционной системы). Однако "пробить" эмулятор может только специальным образом спроектированная программа. Сетевое взаимодействие — другое дело. Эмуляция виртуальной локальной сети сохраняет все уязвимости базовой операционной системы, и сетевой червь может ее легко атаковать! Поэтому базовая операционная система должна быть в обязательном порядке исключена из виртуальной локальной сети. Естественно, такое решение существенно затрудняет общение виртуальных машин с внешним миром, и поэтому им часто пренебрегают. Кстати сказать, персональные брандмауэры в большинстве своем не контролируют виртуальные сети и не защищают от вторжения.

Некоторые эмуляторы позволяют взаимодействовать с виртуальными машинами через механизм общих папок, при этом папка хозяйской операционной системы видима как логический диск или сетевой ресурс. При всех преимуществах такого подхода, он интуитивно-небезопасен и в среде хакеров не сыскал особенной популярности.

> ***Примечание***
>
> В *главе 41* будут рассмотрены несколько вариантов теоретически обоснованных атак на эмулятор WMware и даны рекомендации по защите от них.

Расширяемость

Профессионально-ориентированный эмулятор должен поддерживать возможность подключения внешних модулей, имитирующих нестандартное оборудование (например, HASP). Особенно это актуально для исследования защит типа Star Force 3, напрямую взаимодействующих с аппаратурой и привязывающихся к тем особенностям ее поведения, о которых штатные эмуляторы порой даже и не подозревают.

Некоторые из эмуляторов расширяемы, некоторые — нет. Но даже у самых расширяемых из них степень маневренности и глубина конфигурируемости довольно невелики и поверхностно документированы (если документированы вообще). Наверное, это происходит от того, что фактор расширяемости реально требуется очень и очень немногим… Эмуляторы ведь пишут не для хакеров! А жаль!

Доступность исходных текстов

Наличие исходных текстов частично компенсирует низкое качество документации и недостаточную расширяемость эмулятора. Если подопытная программа отказывается выполняться под эмулятором, исходные тексты помогут разобраться в ситуации и устранить дефект. К тому же мы

можем оснастить эмулятор всем необходимым нам инструментарием. Например, это могут быть дампер памяти или средство обратной трассировки, позволяющее прокручивать выполнение программы в обратном порядке. Наконец, доступ к исходным текстам дает возможность оперативного добавления недокументированных машинных команд или наборов инструкций новых процессоров.

К сожалению, коммерческие эмуляторы распространяются без исходных текстов, а OpenSource-эмуляторы все еще не вышли из юношеского возраста и для решения серьезных задач непригодны. Увы! "Мир" — это синоним слова "несовершенство"!

Качество эмуляции

Какой прок от эмулятора, если на нем нельзя запускать SoftICE? Можно, конечно, использовать и другие отладчики (например, Olly Debugger), но их возможности намного более ограничены, к тому же на некачественных эмуляторах некоторые из защищенных программ просто не идут!

Для увеличения скорости эмуляции многие из разработчиков сознательно усекают набор эмулируемых команд, поддерживая только наиболее актуальные из них (в особенности это относится к привилегированным командам защищенного режима, командам математического сопроцессора, включая "мультимедийные", и некоторым "редкоземельным" командам реального режима). Служебные регистры, флаги трассировки и другие подобные им возможности чаще всего остаются незадействованными. Тем не менее, такие эмуляторы пригодны не только для запуска игр! Их можно использовать как "карантинную" зону для проверки свежераздобытых программ на вирусы или как подопытную мышь для экспериментов с тем же Disk Editor.

Коммерческие эмуляторы в большинстве своем используют механизмы динамической эмуляции, эмулируя только привилегированные команды, а все остальные выполняя на "живом" процессоре — в сумеречной зоне изолированного адресного пространства, окруженной частоколом виртуальных портов, что не только существенно увеличивает производительность, но и автоматически добавляет поддержку всех новомодных мультимедийных команд (разумеется, при условии, что их поддерживает ваш физический процессор).

Между тем, в обработке исключительных ситуаций, воздействиях команд на флаги, недопустимых способах адресации, эмуляторы (даже динамические!) зачастую ведут себя совсем не так, как настоящий процессор, и защитный код может выяснить это! Впрочем, если защищенная программа не будет работать под эмулятором, это сильно возмутит легальных пользователей.

Встроенный отладчик

Защищенные программы всячески противостоят отладчикам, дизассемблерам, дамперам и прочему хакерскому оружию. Как правило, до нулевого кольца дело не доходит, хотя некоторые защиты, например, Themida (бывший Extereme Protector) работают и там. Существуют десятки, если не сотни, способов "ослепить" отладчик, и противостоять им достаточно трудно, особенно если вы только начали хакерствовать.

Могущество эмулятора как раз и заключается в том, что он полностью контролирует выполняемый код, и обычные антиотладочные приемы на нем не срабатывают. К тому же, аппаратные ограничения эмулируемого процессора на сам эмулятор не распространяются. В частности, количество "аппаратных" точек останова не обязано равняться четырем, как на платформе x86. При необходимости, эмулятор может поддерживать тысячу или даже миллион точек останова, причем условия их срабатывания могут быть сколь угодно извращенными (например, можно всплывать на каждой команде `Jx`, следующей за командой `TEST EAX, EAX`, соответствующей конструкции `if (my_func())`…).

Естественно, для этого эмулятор должен быть оснащен интегрированным отладчиком. Любой другой отладчик, запущенный под эмулятором, например SoftICE, никаких дополнительных преимуществ не получает. Возможности имеющихся интегрированных отладчиков достаточно невелики. Как правило, они обеспечивают ничуть не лучшую функциональность, чем debug.com, а нередко

существенно уступают ему, поэтому к ним стоит прибегать лишь в крайних случаях, когда обыкновенные отладчики с защитой уже не справляются.

Сводная таблица характеристик эмуляторов

Итак, какой же эмулятор все-таки выбрать? Правильное решение поможет вам принять сводная таблица характеристик эмуляторов (табл. 2.1).

Таблица 2.1. *Сравнительная оценка характеристик наиболее популярных эмуляторов (неблагоприятные характеристики выделены серым цветом)*

	DOSBox	**Bochs**	**Microsoft Virtual PC**	**VMware Server**	**Xen**	**Parallels Workstation**
Разработчик	Питер Веенстра (Peter Veenstra) и Sjoerd при содействии сообщества пользователей	Кевин Лоутон (Kevin Lawton)	Microsoft	VMware	Intel и AMD при участии исследовательской группы Кэмбриджского Университета (University of Cambridge)	Parallels
Лицензия	GPL	LGPL	Закрытый код (бесплатное ПО с июля 2006)	Закрытый код (доступны бесплатные версии)	GPL	Закрытый код (коммерческий продукт)
Круг пользователей	Пользователи приложений DOS (в особенности игр)	Разработчики ПО	Любители игр, разработчики ПО, системные администраторы, офисные работники	Любители игр, разработчики ПО, системные администраторы, офисные работники, тестировщики	Любители игр, разработчики ПО, системные администраторы, офисные работники, тестировщики	Любители игр, разработчики ПО, системные администраторы, офисные работники, тестировщики
Эмулируемые процессоры	Intel x86	Intel x86, AMD64	Intel x86	Intel x86, AMD64	Соответствует физически установленному процессору (поддерживаются Intel x86, AMD64, IA-64)	Intel x86
Эмуляция SMP	Отсутствует	Реализована	Отсутствует	Реализована	Реализована	Отсутствует, но планируется в последующих версиях
Основные операционные системы	GNU/Linux, Windows, Mac OS X, Mac OS Classic, BeOS, FreeBSD, OpenBSD, Solaris, QNX, IRIX	Windows, Linux, BeOS, IRIX, AIX	Windows	Windows, Linux	Linux, NetBSD	Windows, Linux, Mac OS X (Intel version)
Официально поддерживаемые гостевые ОС	Внутренняя эмуляция DOS	DOS, Windows, Linux, xBSD	DOS, Windows, OS/2	DOS, Windows, Linux, FreeBSD, Netware, Solaris	Windows XP and 2003 Server, Linux, FreeBSD, NetBSD, OpenBSD	DOS, Windows, Linux, FreeBSD, OS/2, Solaris
Скорость эмуляции по сравнению с основной системой	Крайне низкая	Низкая	Высокая, приближается к скорости основной системы	Высокая, приближается к скорости основной системы	Соответствует скорости основной системы	Приближается к скорости основной системы

Глава 3

Хакерский инструментарий для UNIX и Linux

С точки зрения хакера, выбор специфических кодокопательских утилит для UNIX и Linux невелик. Во всяком случае, он существенно меньше, чем выбор аналогичных утилит в мире Windows. Хакерам приходится делать свою работу практически "голыми руками", причем для этого требуются изрядные усидчивость и трудолюбие. Больше всего удручает отсутствие отладчика, равного по своим возможностям если не SoftICE, то хотя бы OllyDbg. Готовых к употреблению и достойных утилит наподобие дамперов памяти, модификаторов (патчеров), автоматических распаковщиков упакованных файлов в сети практически нет, лишь бесконечные кладбища заброшенных проектов… Поэтому такие утилиты также приходится писать самостоятельно.

Будем надеяться, что через несколько лет ситуация изменится, поскольку, как известно, спрос рождает предложение. Пока же ограничимся кратким обзором доступных инструментов, а также рассмотрим некоторые методы, позволяющие вручную скомпилировать заброшенные и более не поддерживаемые проекты.

> ***Примечание***
>
> UNIX-программистам, как правило, не свойственно "зажимать" исходники, и подавляющее большинство программ (в том числе и рассматриваемые в данной главе) распространяются именно так. Тем не менее, многие пользователи стремятся скачивать готовые к употребления бинарные сборки, зачастую даже не догадываясь, каких возможностей они оказываются лишены! Во второй части данной главы, *"Скрытый потенциал ручных сборок"*, будут даны рекомендации и показаны приемы, используя которые вы сможете наращивать функциональные возможности хакерских утилит UNIX, если вас не устраивает набор функций, предоставляемых готовыми к употреблению исполняемыми файлами.

Отладчики

В первую очередь упоминания заслуживает GDB — кросс-платформенный отладчик исходного уровня (source-level debugger), основанный на библиотеке `Ptrace`[1] и ориентированный преимущественно на отладку приложений, поставляемых с исходными текстами. Для взлома защищенных приложений он не слишком пригоден. Отладчик поддерживает аппаратные точки останова на исполнение. Точки останова на чтение/запись памяти при запуске из-под VMware не срабатывают. Кроме того, GDB не дает возможности устанавливать точки останова на совместно используемую память и модифицировать ее (это означает, что отладить с его помощью утилиту наподобие `ls` вряд ли возможно). Поиск в памяти отсутствует как таковой. Отладчик отказывается загружать файл с искаженной структурой или с отсутствующей таблицей секций (sections table). Внешне он представляет собой консольное приложение (рис. 3.1) со сложной системой

[1] Если вы не знакомы с этой библиотекой, то в первую очередь прочтите минимально необходимую документацию о ней, дав команду `man ptrace`.

команд, полное описание которых занимает порядка трехсот страниц убористого текста. При желании, к отладчику можно добавить графическую оболочку (недостатка в которых испытывать не приходится), однако красивым интерфейсом перечисленные недостатки не исправишь. За время своего существования GDB успел обрасти огромным количеством антиотладочных приемов, многие из которых остаются актуальными и до сих пор. К достоинствам GDB можно отнести его бесплатность. Данный отладчик распространяется по лицензии GNU (отсюда и его название — GNU DeBugger) и входит в комплект поставки большинства дистрибутивов UNIX. Наконец, с его помощью можно накладывать заплатки на исполняемый файл, не выходя из отладчика.

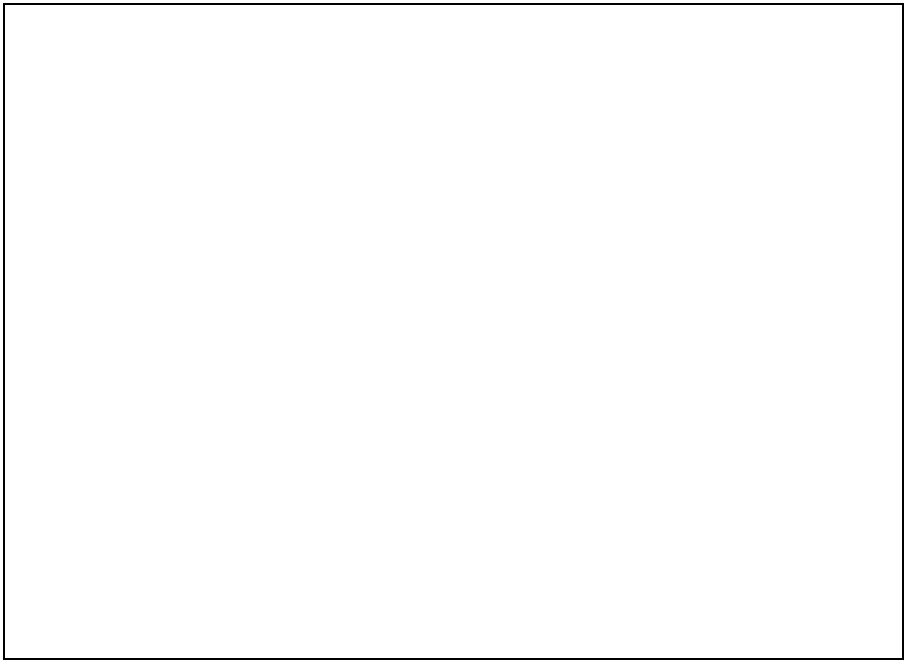

Рис. 3.1. Отладчик GDB за работой

Примечание

Вот небольшой совет для тех, кто только приступает к работе с GDB: чтобы установить точку останова на точку входа (entry point) отлаживаемой программы, необходимо предварительно определить ее адрес. Для этой цели вам пригодится штатная утилита `objdump`[2] (только для незащищенных файлов!) или комбинация шестнадцатеричного редактора BIEW и дизассемблера IDA Pro[3]. Итак, чтобы добиться поставленной цели, дайте команду `objdump` *`file_name`* `– f`, затем загрузите отлаживаемую программу в GDB (`gdb -q` *`file_name`*) и дайте команду `break *0xXXXXXXXX`, где `0xX` — стартовый адрес, а затем запустите отлаживаемую программу на выполнение командой `run`. Если все прошло успешно, GDB тут же остановится, передавая вам бразды правления. Если же этого не произойдет, откройте файл с помощью шестнадцате-

[2] Утилита `objdump` — это аналог утилиты `dumpbin` от Microsoft, но предназначенный для файлов формата ELF (о структуре файлов формата ELF более подробно рассказывается в *главе 31, "Дизассемблирование ELF-файлов под Linux и BSD"*). `Objdump` содержит простенький дизассемблер. Утилита требует обязательного наличия таблицы секций, не может интерпретировать искаженные поля и не справляется с упакованными файлами. Тем не менее, при отсутствии IDA Pro сгодится и она.

[3] Более подробная информация об использовании этой комбинации будет приведена далее в этой главе.

ричного редактора BIEW[4] и внедрите в точку входа код `CCh` (машинная команда `INT 03`), предварительно запомнив ее оригинальное содержимое. Затем перезапустите отладчик, а после достижения точки останова восстановите ее содержимое (`set {char} *0xXXXXXXXX = YY`).

Отладчик ALD (Assembly Language Debugger) — это быстрый отладчик прикладного уровня, ориентированный на отладку ассемблерных текстов и двоичных файлов (рис. 3.2). Скачать его можно по адресу **http://ald.sourceforge.net/**. ALD основан на библиотеке `Ptrace` со всеми вытекающими отсюда последствиями. В настоящее время он работает только на платформе x86. Отладчик успешно компилируется для работы под управлением следующих операционных систем: Linux, FreeBSD, NetBSD и OpenBSD. Он поддерживает точки останова на выполнение, пошаговую/покомандную трассировку, просмотр/редактирование дампа, простор/изменение регистров и содержит простенький дизассемблер. Довольно аскетичный набор функций! Для сравнения, даже достопочтенный debug.com для MS-DOS — и тот предоставлял более широкий набор возможностей. Зато ALD бесплатен, распространяется в исходных текстах и, что немаловажно, не отказывается загружать файлы, в которых отсутствует таблица секций. Для обучения взлому он вполне подойдет, но на основной хакерский инструмент, увы, не тянет.

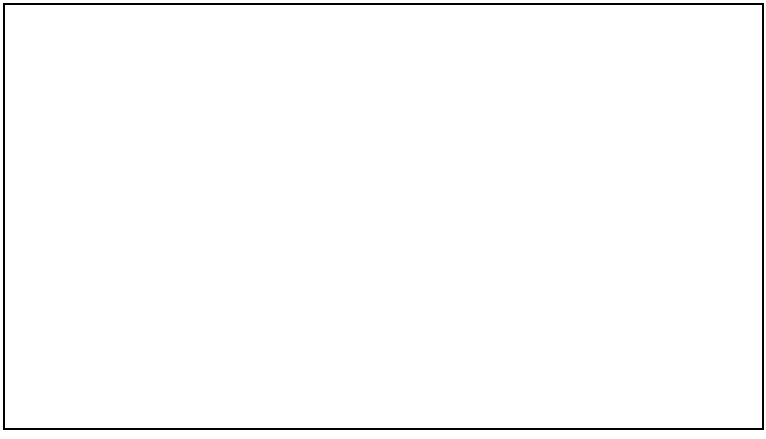

Рис. 3.2. Отладчик ALD за работой

Еще один интересный отладчик исходного уровня — The Dude (**http://the-dude.sourceforge.net/**). Он работает в обход `Ptrace` и успешно выполняет свои задачи там, где GDB и ALD уже не справляются. К сожалению, он работает только под Linux, а поклонникам остальных клонов UNIX остается лишь завидовать. Архитектурно The Dude стоит из трех основных частей: модуля ядра the_dude.o, реализующего низкоуровневые отладочные функции, сопрягающей библиотечной "обертки" вокруг него — libduderino.so, а также внешнего пользовательского интерфейса — ddbg. Собственно говоря, пользовательский интерфейс лучше сразу же переписать. Отладчик бесплатен, но для его скачивания требуется предварительная регистрация на сайте **http://www.sourceforge.net**.

Linice (**http://www.linice.com/**) — это чрезвычайно мощный отладчик ядерного уровня, ориентированный на работу с двоичными файлами без исходных кодов. Фактически он представляет собой аналог SoftICE для Linux (рис. 3.3). Этот отладчик — основной инструмент любого хакера, работающего под Linux. В настоящее время он работает только на ядре версии 2.4 (и, предположительно — 2.2), но вот компиляция для всех остальных ядер завершается неудачей по причине ошибки в файле Iceface.c. Отладчик добавляет в систему устройство `/dev/ice`, чем легко

[4] О шестнадцатеричном редакторе BIEW чуть подробнее будет рассказано далее в этой главе.

выдает свое присутствие. Впрочем, благодаря наличию исходных текстов, это не представляет серьезной проблемы. Всплывает по нажатию клавиатурной комбинации <CTRL>+<Q>, причем USB-клавиатура пока не поддерживается. Загрузчика нет и пока не предвидится, поэтому единственным способом отладки остается внедрение машинной команды `INT 03` (опкод `CCh`) в точку входа с последующим ручным восстановлением оригинального содержимого.

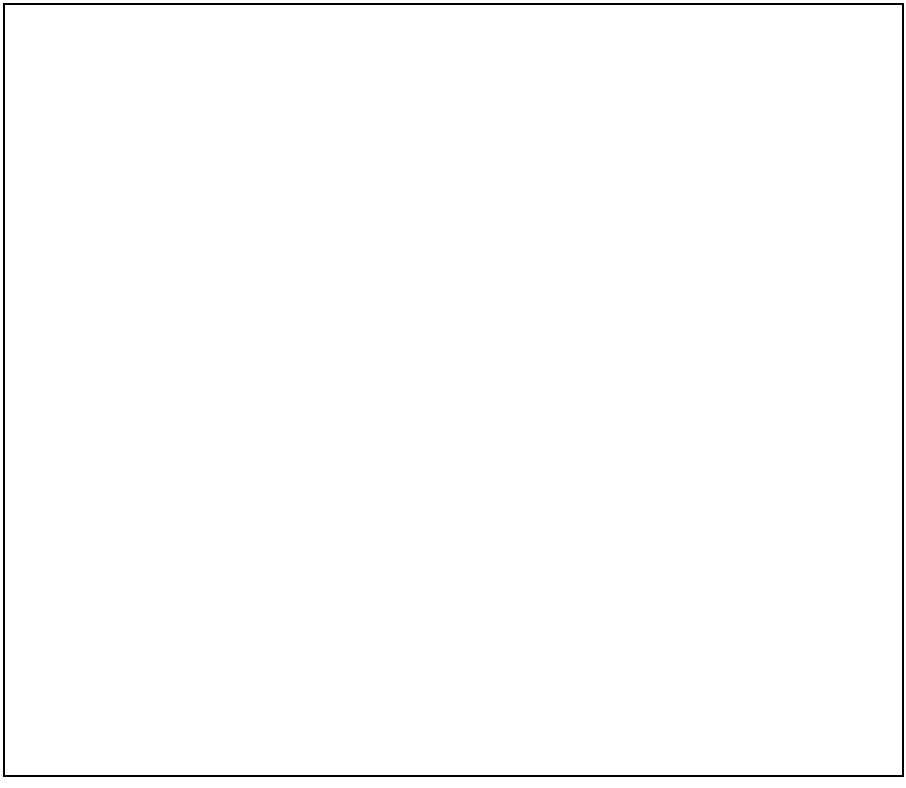

Рис. 3.3. Нет, это не сон, это аналог SoftICE под Linux!

Рис. 3.4. Основная панель эмулятора

PIce (**http://pice.sourceforge.net/**) — это экспериментальный ядерный отладчик для Linux, работающий только в консольном режиме и реализующий минимум функций. Тем не менее, и он на что-то может сгодиться.

Наконец, x86 Emulator (рис. 3.4), который можно скачать по адресу **http://ida-x86emu.sourceforge.net/** — это эмулирующий отладчик, конструктивно выполненный в виде плагина для IDA Pro и распространяющийся в исходных текстах без прекомпиляции[5]. Основное достоинство эмулятора состоит в том, что он позволяет выполнять произвольные фрагменты кода на виртуальном процессоре. Например, с его помощью можно передавать управление процедуре проверки серийного номера/пароля, минуя остальной код. Такая техника совмещает лучшие черты статического и динамического анализа, значительно упрощая взлом заковыристых защит.

Дизассемблеры

IDA Pro (**http://www.datarescue.com/idabase**) — лучший дизассемблер всех времен и народов, теперь доступный и под Linux! Поклонники же FreeBSD и остальных операционных систем могут довольствоваться консольной Windows-версией (рис. 3.5), запущенной под эмулятором, или работать с ней непосредственно из-под MS-DOS, OS/2 или Windows. До недавнего времени IDA Pro отказывалась дизассемблировать файлы без таблицы секций, однако в последних версиях этот недостаток был устранен. Отсутствие достойных отладчиков под UNIX превращает IDA Pro в основной инструмент взломщика.

Рис. 3.5. Консольная версия IDA Pro

[5] Это означает, что кроме IDA Pro вам дополнительно потребуется и IDA Pro SDK, поставляющийся только для коммерческих версий IDA Pro.

Если же у вас нет возможности раздобыть IDA Pro, то, возможно, ваше внимание привлечет один из следующих дизассемблеров:

- Ранее упомянутая утилита `objdump` — штатная утилита, которая может применяться для общего анализа двоичных файлов.
- Bastard Disassembler (**http://bastard.sourceforge.net/**) — дизассемблер (или, точнее говоря, среда дизассемблирования) для Linux и FreeBSD. Эта среда дизассемблирования предоставляет интерпретатор (по аналогии с Perl или Python), с помощью которого можно загружать и дизассемблировать файлы, сохранять дамп дизассемблированного кода. Предоставляет возможности написания макросов.
- Lida (Linux Interactive DisAssembler) — интерактивный дизассемблер, предоставляющий ряд специализированных функций (например, криптоанализатор). Домашняя страница проекта — **http://lida.sourceforge.net**.
- LDasm (Linux Disassembler) — графическая оболочка для `objdump`/`binutils`, имитирующая поведение упоминавшегося в *главе 1* дизассемблера W32Dasm. Скачать LDasm можно по адресу **http://www.feedface.com/projects/ldasm.html**.

Шпионы

`Truss` — полезная утилита, штатным образом входящая в комплект поставки большинства дистрибутивов UNIX. Отслеживает системные вызовы (syscalls) и сигналы (signals), совершаемые подопытной программой с прикладного уровня (рис. 3.6), что позволяет многое сказать о внутреннем мире защитного механизма.

Рис. 3.6. Отслеживание системных вызовов с помощью truss

Пример отчета, создаваемого утилитой `truss`, приведен в листинге 3.1.

Листинг 3.1. Образец отчета, созданного утилитой truss

```
mmap(0x0,4096,0x3,0x1002,-1,0x0)                = 671657984 (0x2808b000)
break(0x809b000)                                = 0 (0x0)
break(0x809c000)                                = 0 (0x0)
break(0x809d000)                                = 0 (0x0)
break(0x809e000)                                = 0 (0x0)
stat(".",0xbfbff514)                            = 0 (0x0)
```

```
open(".",0,00)                                          = 3 (0x3)
fchdir(0x3)                                             = 0 (0x0)
open(".",0,00)                                          = 4 (0x4)
stat(".",0xbfbff4d4)                                    = 0 (0x0)
open(".",4,00)                                          = 5 (0x5)
fstat(5,0xbfbff4d4)                                     = 0 (0x0)
fcntl(0x5,0x2,0x1)                                      = 0 (0x0)
__sysctl(0xbfbff38c,0x2,0x8096ab0,0xbfbff388,0x0,0x0)   = 0 (0x0)
fstatfs(0x5,0xbfbff3d4)                                 = 0 (0x0)
break(0x809f000)                                        = 0 (0x0)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)           = 512 (0x200)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)           = 0 (0x0)
lseek(5,0x0,0)                                          = 0 (0x0)
close(5)                                                = 0 (0x0)
fchdir(0x4)                                             = 0 (0x0)
close(4)                                                = 0 (0x0)
fstat(1,0xbfbff104)                                     = 0 (0x0)
break(0x80a3000)                                        = 0 (0x0)
write(1,0x809f000,158)                                  = 158 (0x9e)
exit(0x0)                                              process exit, rval = 0
```

`Ktrace` — еще одна утилита из штатного комплекта поставки. Она отслеживает системные вызовы, синтаксический разбор имен (names translation), операции ввода-вывода, сигналы, трассировку режима пользователя и переключение контекстов, совершаемое исследуемой программой с ядерного уровня. Иными словами, `ktrace` представляет собой улучшенный вариант `truss`, однако в отличие от последней, `ktrace` выдает отчет не в текстовой, а двоичной форме (листинг 3.2). Поэтому для генерации отчетов необходимо воспользоваться утилитой `kdump`.

Листинг 3.2. Образец отчета ktrace

```
 8259      ktrace          CALL    write(0x2,0xbfbff3fc,0x8)
 8259      ktrace          GIO     fd 2 wrote 8 bytes
           "ktrace: "
 8259      ktrace          RET     write 8
 8259      ktrace          CALL    write(0x2,0xbfbff42c,0x13)
 8259      ktrace          GIO     fd 2 wrote 19 bytes
           "exec of 'aC' failed"
 8259      ktrace          RET     write 19/0x13
 8259      ktrace          CALL    write(0x2,0xbfbff3ec,0x2)
 8259      ktrace          GIO     fd 2 wrote 2 bytes
           ": "
 8259      ktrace          RET     write 2
 8259      ktrace          CALL    write(0x2,0xbfbff3ec,0x1a)
 8259      ktrace          GIO     fd 2 wrote 26 bytes
           "No such file or directory
           "
 8259      ktrace          RET     write 26/0x1a
 8259      ktrace          CALL    sigprocmask(0x1,0x2805cbe0,0xbfbffa94)
 8259      ktrace          RET     sigprocmask 0
 8259      ktrace          CALL    sigprocmask(0x3,0x2805cbf0,0)
 8259      ktrace          RET     sigprocmask 0
 8259      ktrace          CALL    exit(0x1)
 8265      ktrace          RET     ktrace 0
```

Шестнадцатеричные редакторы

Наиболее мощными из шестнадцатеричных редакторов являются упомянутый в *главе 1* универсальный редактор HT Editor, поддерживающий как Windows, так и Linux, и BIEW (**http://belnet.dl.sourceforge.net/sourceforge/biew/biew562.tar.bz2**) — это шестнадцатеричный

редактор, комбинирующий функции дизассемблера, шифратора и инспектора файлов формата ELF (рис. 3.7). Встроенный ассемблер отсутствует, поэтому работать приходится непосредственно в машинном коде, что не слишком удобно. Однако другого выбора у нас нет (разве что дописать ассемблер самостоятельно).

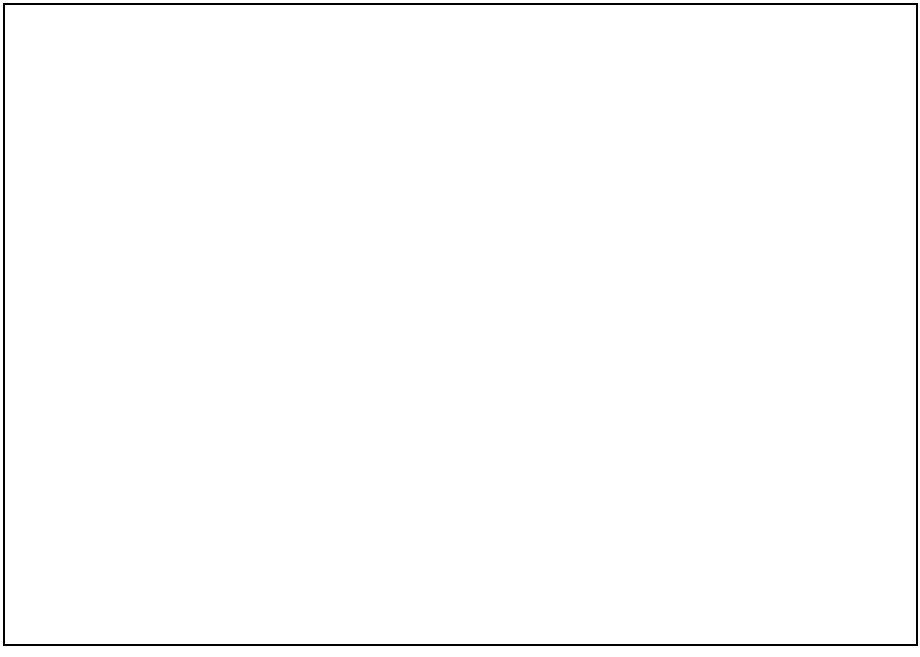

Рис. 3.7. Шестнадцатеричный редактор BIEW

Из других шестнадцатеричных редакторов стоит упомянуть GHex (**http://directory.fsf.org/All_Packages_in_Directory/ghex.html**) — шестнадцатеричный редактор для GNOME и KHexEdit (**http://home.online.no/~espensa/khexedit/index.html**) — достаточно гибкий шестнадцатеричный редактор для KDE, отображающий данные в шестнадцатеричном, восьмеричном, двоичном и текстовом форматах. Для работы с крупными файлами (600 Мбайт и более) подойдет разработанный специально для этой цели экспериментальный редактор hview (пока доступна только бета-версия), который можно скачать по адресу **http://tdistortion.esmartdesign.com/Zips/hview.tgz**. Кроме того, говоря о шестнадцатеричных редакторах для UNIX и Linux, не стоит забывать и о `emacs`, который, наряду с прочими функциями, включает и шестнадцатеричный редактор.

Дамперы

В UNIX содержимое памяти каждого из процессоров представлено в виде набора файлов, расположенных в каталоге /proc. Здесь же хранится контекст регистров и прочая информация. Однако дамп памяти — это еще не готовый ELF-файл, и к непосредственному употреблению он не пригоден. Тем не менее, дизассемблировать его "сырой" образ вполне возможно.

Скрытый потенциал ручных сборок

Как уже говорилось в начале данной главы, подавляющее большинство программного обеспечения для UNIX и Linux распространяется не только в виде готовых сборок, но и в исходных кодах. Тем не менее, большинство обычных пользователей предпочитают скачивать готовые к употреб-

лению дистрибутивы. Почему они это делают? Да просто не хотят мучиться, компонуя программу вручную. Однако не стоит торопиться сразу же упрекать их в лени, говоря, что пользоваться готовыми сборками — это не "unix-way" и совсем не по-хакерски. Вот целый ряд причин, по которым готовые сборки иногда оказываются лучше исходных текстов:

- Готовая сборка имеет намного меньший объем, чем исходные тексты, даже сжатые самым лучшим архиватором. Таким образом, скачивать их выгодно, если вы экономите на трафике или владеете медленным интернет-соединением (особенно dial-up). Кроме того, не следует забывать и о том, что далеко не каждый сервер поддерживает докачку.
- В разархивированном виде исходные тексты занимают существенный объем дискового пространства, а сама компиляция требует значительного времени, которое, как известно, всегда работает против нас.
- "Ручная" настройка программы требует внимательного чтения руководств и изучения конфигурационных скриптов. Дело в том, что сборка с опциями по умолчанию в лучшем случае ничем не отличается от официальной сборки.
- Довольно часто требуется скачивать дополнительные заголовочные файлы и библиотеки, обновлять компилятор, и так далее. Все это тоже требует времени, а также потребляет и трафик, и дисковое пространство.
- Качество автоматических инсталляторов обычно оставляет желать лучшего, и скомпилированную программу еще долго приходится дорабатывать.
- Готовые сборки обычно включают в себя "бонусы", например, нестандартные цветовые схемы или дополнительные компоненты, созданные сторонними разработчиками (рис. 3.8). В официальных исходных текстах таких "бонусов" может и не быть.

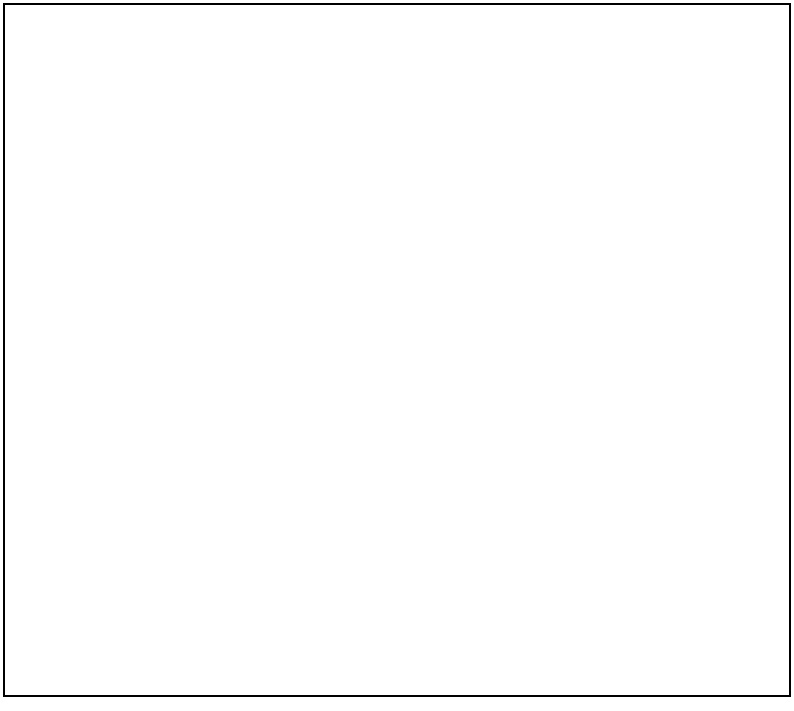

Рис. 3.8. Графический интерфейс к интегрированному отладчику эмулятора Bochs, входящий в одну из неофициальных сборок

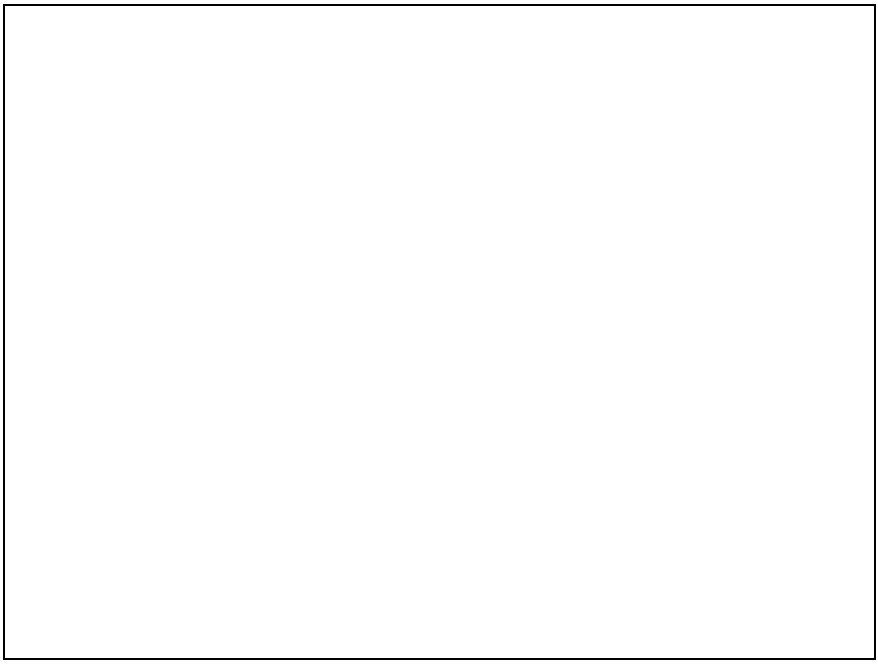

Рис. 3.9. Официальная сборка эмулятора Bochs не поддерживает архитектуры x86-64, поэтому 64-битная версия Linux грустно говорит `Sorry, your CPU is not capable of running 64-bit kernel`, и загрузка прекращается (`System halted`)

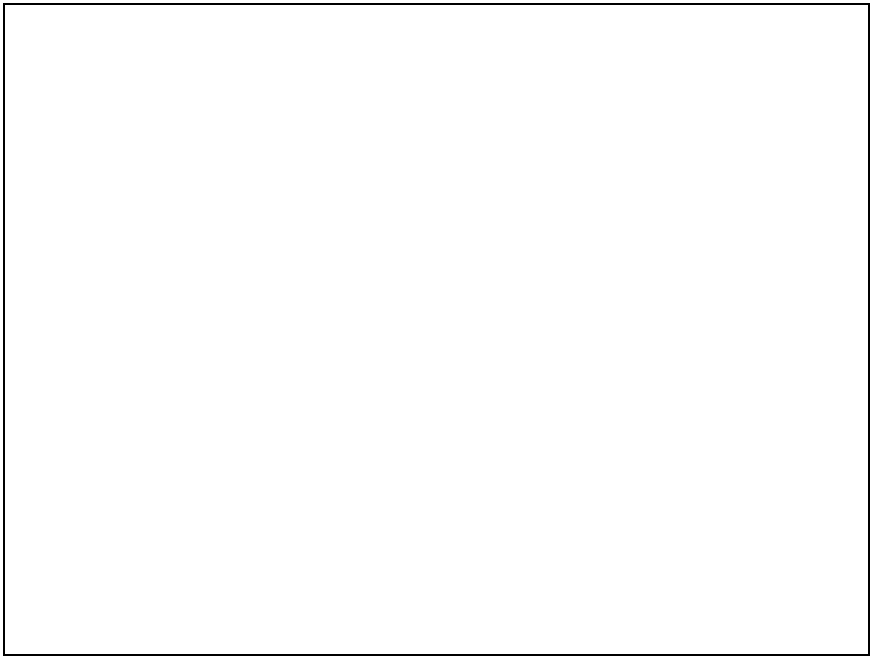

Рис. 3.10. После перекомпиляции с ключом `--enable-x86-64` 64-битная версия Linux запускается без проблем и работает с приличной скоростью, даже если сам Bochs запущен из-под VMware на Pentium III 733

- Всегда найдется тысяча причин, по которым собранная вручную программа будет работать неправильно или нестабильно. Представьте, например, такую ситуацию, когда пользователь активирует соблазнительную опцию, которая еще не доработана и находится в стадии "under construction". Это запросто может привести к появлению глюков в самых неожиданных местах.
- Программы, собранные вручную, значительно труднее удалить из системы, чем пакеты rpm (впрочем, существуют утилиты, автоматизирующие этот процесс).
- Если необходимые вам опции отсутствуют в официальной сборке, как, например, поддержка архитектуры x86-64 в эмуляторе Bochs (рис. 3.9), то практически всегда можно найти неофициальную сборку, в которой этот недостаток устранен (рис. 3.10). Стоит, правда, отметить, что далеко не все неофициальные сборки собраны правильно.
- Пословица "лучше за день долететь, чем за час добежать" в условиях сурового корпоративного мира неприменима, и если готовая сборка гарантированно хоть как-то работает, то эксперименты с ручной компиляцией "за просто так" нам никто не оплатит.

А вот теперь рассмотрим основные причины, по которым исходные тексты лучше готовых дистрибутивов.

- Сборки существуют не для всех платформ. Это особенно справедливо для платформ x86-64 и операционных систем наподобие QNX или BeOS.
- Для экспериментальных альфа- и бета-версий готовые сборки практически всегда отсутствуют, а для текущих стабильных версий они выходят нечасто. Поэтому, если вы не будете заниматься ручными сборками, вам зачастую придется работать с версиями одно- или двухгодичной давности (и это далеко не преувеличение!), с завистью поглядывая на коллег, собравших последнюю альфу с огромным количеством всяких "вкусностей" и нововведений.
- Готовые сборки включают не все функциональные возможности, реализованные в исходных текстах (в частности, в состав популярного эмулятора Bochs входит внешний интерактивный отладчик в стиле Turbo Debugger, тогда как официальные сборки содержат простейший интегрированный отладчик в стиле debug.com).
- Для многих программ существуют расширения, созданные сторонними разработчиками и устанавливаемые только посредством перекомпиляции.
- Готовая сборка зачастую включает много лишних компонентов, которые только потребляют процессорные ресурсы и память, но лично вам попросту не нужны. В частности, программа может поддерживать консольный и графический интерфейсы. Если вы хотите работать только с командной строкой и не намерены пользоваться графическими оболочками, то прямой резон перекомпилировать программу без поддержки GUI.
- Официальные сборки компилируются с типовыми опциями оптимизации, общими для всех процессоров. В результате этого код получается неэффективным и неоптимальным. В ряде случаев он может оказаться вообще неработоспособным (на старых процессорах 80386 или 80486).
- При выходе новой версии всю сборку приходится заново скачивать в полном объеме, вместо того, чтобы забрать только измененные файлы (имейте в виду, что для часто обновляемых программ это актуально).
- Если программа содержит уязвимость, то атаковать готовую сборку проще, поскольку атакующий знает точное расположение всех машинных команд и раскладку памяти.
- Заплатки к исходным текстам выходят намного быстрее и чаще, чем к готовым сборкам (зачастую на готовые сборки вообще невозможно наложить заплатку, и приходится скачивать весь "залатанный" дистрибутив целиком).

Именно поэтому многие пользователи и берутся за перекомпиляцию программ, но лишь немногие делают это *правильно*! На первый взгляд может показаться, что ручная сборка доступна любому

продвинутому пользователю, имеющему минимальный опыт общения с компиляторами, а также обладающему достаточным количеством свободного времени и умением читать по-английски без помощи переводчиков. Это действительно так, но лишь отчасти. На самом деле ручная сборка — это достаточно сложный, взаимно противоречивый и неочевидный процесс, который мы сейчас и попробуем рассмотреть. Имейте в виду, что универсальных решений здесь нет! Каждый путь содержит свои минусы и плюсы.

> ***Примечание***
>
> Если вы еще не имеете достаточного опыта в области ручной сборки, то наилучшим подходом будет соблюдение следующих простых рекомендаций. Сначала скачайте готовую сборку, немного поработайте с программой, разберитесь в структуре каталогов и освойтесь с основными возможностями. Затем можно приступать к экспериментам. По крайней мере, правильно собранный эталон всегда будет перед глазами. Если компиляция пойдет не так, как ожидалось, или если собранная программа откажет в работе, эталонная сборка поможет установить, что же было сделано не так.

Философская подготовка

Компиляция программы всегда начинается с чтения инструкции. Зайдите на сайт проекта, перейдите в раздел **Downloads**, скачайте протокол изменений (файлы changes, what's new, readme) и вдумчиво прочитайте их. Это необходимо для того, чтобы выяснить, чем отличается имеющаяся у вас версия от скачиваемой, и нужны ли вам все эти нововведения. Практика показывает, что многие программы останавливаются в своем развитии еще в зачатии, а затем "жиреют", наращивая избыточную функциональность. Нужно ли гнаться за модой и прогрессом, стремясь использовать последние версии программ, только потому, что они "последние"? Машинный код, в отличие от молока, со временем не портится и не скисает, а хакеры, в отличие от простых пользователей, намного более консервативны. Обычно они с большим недоверием относятся ко всему новому. Как сказал кто-то из них, *"я не могу работать инструментом, который совершенствуется у меня в руке"*.

Пользователи в этом отношении намного более "прогрессивны" и качают все, что только попадает в поле их зрения. При этом среди них существует устойчивое предубеждение, гласящее, что лучше всего скачивать стабильные ветви (stable), также называемые релизами (release), дескать, они работают намного надежнее экспериментальных альфа/бета-версий. Какая-то доля правды в этом есть. Тем не менее, в общем случае дела обстоят совсем не так. Стабильные версии выходят редко. За это время в них находятся баги, планомерно устраняемые в промежуточных версиях, носящих статус "нестабильных". В перерывах между схватками с багами, разработчики добавляют новые функциональные возможности (или расширяют уже существующие). Например, может быть реализована поддержка передовых протоколов передачи данных или новых форматов файлов. Какой смысл ждать релиза, когда текущую версию можно скачать прямо сейчас? К тому же, чем больше людей "подсядут" на альфу, тем больше ошибок будет в ней обнаружено! Не стоит надеяться, что другие сделают эту работу за вас! В отличие от Microsoft, разработчики бесплатных программ не могут содержать бригаду бета-тестеров, поэтому с ошибками вынуждены сражаться сами пользователи. Впрочем, никакого произвола здесь нет. Не хотите — не сражайтесь.

Пошаговая инструкция

Исходные тексты обычно распространяются в архивах, упакованных популярными архиваторами, такими как pkzip, gzip, bzip, реже — в виде дерева CVS. "Что такое CVS?" — спросите вы. Это — одна из самых популярных систем управления версиями (Concurrent Version System), позволяющая нескольким программистам работать над одним проектом. Система не только отслеживает изменения, синхронизуя файлы всех участников, но еще и разграничивает привилегии — кто и куда может писать. Посторонние лица (анонимные пользователи, не участвующие в проекте)

могут только читать. Более подробную информацию об этой системе можно найти по адресам: **http://ru.wikipedia.org/wiki/CVS**, **http://www.nongnu.org/cvs/** (на английском языке), и **http://alexm.here.ru/cvs-ru/** (на русском языке).

Чтобы работать с деревом CVS, необходимо установить клиента CVS (в большинстве дистрибутивов UNIX это программное обеспечение уже установлено), зарегистрировавшись в системе как `anonymous`. В общем случае это делается так: `$cvs -d:pserver:anonymous@server:patch login`. В листинге 3.3 приведен конкретный пример, иллюстрирующий процесс скачивания дерева CVS для эмулятора Bochs.

Листинг 3.3. Сеанс работы с сервером CVS на примере эмулятора Bochs

```
$ cvs -d:pserver:anonymous@cvs.bochs.sourceforge.net:/cvsroot/bochs login
  (Logging in to anonymous@cvs.bochs.sourceforge.net)
CVS password:    (there is no password, just press Enter)
user$ cvs -z3 -d:pserver:anonymous@cvs.bochs.sf.net:/cvsroot/bochs checkout bochs
  cvs server: Updating bochs
  U bochs/.bochsrc
  U bochs/.conf.AIX.4.3.1
  U bochs/.conf.beos-x86-R4
  U bochs/.conf.macos
     .
     .   (This might take a few minutes, depending on your network connection.)
     .
  U bochs/patches/patch.seg-limit-real
```

Если подключение к серверу завершилось успехом, то сразу же после авторизации начинается процедура синхронизации файлов (в данном случае — `checkout`, то есть извлечение целого модуля из CVS и создание рабочей копии). Скачивание всех файлов проекта происходит потому, что синхронизовать пока еще нечего.

Скачивание всех файлов проекта даже на выделенных линиях занимает довольно длительный промежуток времени. Файлы передаются в несжатом (точнее, очень слабо сжатом) виде, и докачка поддерживается лишь частично. Частичная поддержка докачки означает, что при неожиданном разрыве связи файлы, скачанные целиком, повторно не передаются. Однако загрузка файлов, переданных не полностью, начинается сначала. При частых разрывах связи это создает серьезные неудобства, усугубляющиеся тем, что дерево CVS содержит много лишних файлов, которые не попадают в "упакованный дистрибутив". Тем не менее, вы будете вынуждены их качать...

Так какой же тогда смысл возиться с CVS? Не проще ли (быстрее, дешевле) воспользоваться готовым архивом? Однозначного ответа на вопрос нет и не будет. Начнем с того, что некоторые программы распространяются ***только*** через CVS. Архив, если и выкладывается, зачастую содержит не все файлы или не обновляется месяцами. С другой стороны, при выходе новой версии, весь архив приходится перекачивать от начала и до конца, в то время как клиент CVS забирает только реально измененные файлы, что существенно экономит трафик.

Короче говоря, при частых обновлениях использовать CVS выгодно, в противном случае, лучше скачать готовый архив, выбрав из предложенных архиваторов свой любимый. Между прочим, даже при отсутствии прав на запись, CVS-клиент все равно отслеживает изменение локальных файлов, и если мы что-то подправили в программе, измененные файлы останутся необновленными.

При желании скачать стабильную (а не текущую!) версию, следует использовать ключ `-r`, и тогда командная строка будет выглядеть так: `$cvs update -d -r tagname`, где `tagname` — кодовое имя проекта (например, `REL_2_0_2_FINAL`), название которого можно найти на сайте разработчиков или почерпнуть из документации.

Примечание

Многие серверы (в том числе и **http://www.sourceforge.net**) позволяют просматривать дерево CVS через WEB-интерфейс, однако особого смысла в этом нет. Для того чтобы скачать исходные файлы, вам потребуется оффлайн-браузер наподобие Teleport Pro (**http://www.listsoft.ru/programs/172/**), так что гораздо лучший подход состоит все же в использовании клиента CVS.

Последнюю версию самого клиента CVS можно скачать с ftp-сервера **http://ftp.gnu.org/non-gnu/cvs/** (**http://www.tortoisecvs.org**/, **http://www.wincvs.org/** — версии под Windows), а если возникнут вопросы — к вашим услугам огромный список часто задаваемых вопросов: **http://www.cs.utah.edu/dept/old/texinfo/cvs/FAQ.txt**. На этом обсуждение CVS можно считать законченным.

Займемся теперь готовыми архивами. Как правило, их бывает не просто много, а очень много! Например, страница дистрибутивов, с которой можно скачать архивы исходных текстов браузера Lynx (**http://lynx.isc.org/current/index.html**), в различных версиях насчитывает с полсотни файлов в трех форматах: pkzip, gzip и bzip, имеющих самые различные размеры (листинг 3.4). Какой из них брать? Самый расточительный — pkzip, за ним с небольшим отрывом идет gzip (по сути дела, представляющий той же самый архиватор, но в другом "воплощении"), а bzip лидирует с 1,5—2-кратным разрывом. Правда, не во всех дистрибутивах UNIX он установлен по умолчанию, и тогда его приходится качать самостоятельно: **http://www.bzip.org/**, благо он бесплатен.

Листинг 3.4. Исходные тексты, упакованные различными архиваторами, имеют неодинаковые размеры

```
4550647 Oct 30 12:54 [26]lynx2.8.6dev.15.tar.Z
2259601 Oct 30 12:54 [27]lynx2.8.6dev.15.tar.bz2
3141568 Oct 30 12:54 [28]lynx2.8.6dev.15.tar.gz
3344962 Oct 30 12:54 [29]lynx2.8.6dev.15.zip
```

Скачав архив, поинтересуйтесь, а не прилагается ли к нему заплатка с последними исправлениями? Версия заплатки в обязательном порядке должна совпадать с версией архива, иначе программа рухнет еще на стадии компиляции. Впрочем, это не правило, а скорее рекомендация. Все зависит от того, что это за заплатка и что именно она исправляет. Тем не менее, без необходимости лучше не рисковать.

Заплатки бывают различных видов. Например, в случае с Lynx — это обыкновенный архив измененных файлов, который нужно просто распаковать (с перезаписью) в основной каталог программы. По своему размеру этот архив вплотную приближается к дистрибутиву.

Большинство программистов создают патчи с помощью утилиты `diff` (чтобы получить о ней более подробную информацию, дайте команду `man diff`), получившей свое название в результате сокращения английского difference — разница. Эта утилита построчно сравнивает файлы, отображая только реальные изменения. Знак минуса (-), стоящий впереди, означает, что данная строка была удалена, а знак плюса (+) говорит о том, что данная строка была добавлена. Имя файла предваряется тройным знаком минуса (---) или плюса (+++). Файлы изменений обычно имеют расширение .diff или .patch, но даже без расширения их легко отождествить визуально (листинг 3.5). Как правило, все изменения, внесенные в дистрибутив, собраны в одном diff-файле.

Листинг 3.5. Так выглядит заплатка, созданная утилитой diff

```
diff -pruN BIEW-561/BIEWlib/sysdep/ia32/os2/timer.c BIEW-562/BIEWlib/sysdep/ia32/os2/timer.c
--- BIEW-561/BIEWlib/sysdep/ia32/os2/timer.c    2001-11-18 17:05:48.000000000 +0000
+++ BIEW-562/BIEWlib/sysdep/ia32/os2/timer.c    2004-09-20 19:34:11.000000000 +0000
@@ -29,7 +29,7 @@ static HTIMER   timerID = 0;
 static TID      timerThread = 0;
```

```
 static          timer_callback *user_callback = NULL;

-static VOID __NORETURN__ thread_callback( ULONG threadMsg )
+static VOID __NORETURN__ _Syscall thread_callback( ULONG threadMsg )
 {
   ULONG recv;
   UNUSED(threadMsg);
```

Наложить diff-заплатку можно, в принципе, и вручную. Некоторые так и делают. Другие используют утилиту `patch` (чтобы получить более подробную информацию, дайте команду `man patch`), полностью автоматизирующую этот процесс. В общем случае, ее вызов выглядит, как показано в листинге 3.6.

Листинг 3.6. Наложение заплатки утилитой patch

```
$patch -p1 < my_patch.patch
```

Здесь `my_patch.patch` — имя diff-файла, а `p1` — уровень вложенности. Номер <1> означает, что мы вызываем `patch` из основного каталога программы. Зачем это нужно? Откроем diff-файл в любом редакторе и посмотрим, каким образом в нем задаются пути к файлам, например, для редактора BIEW это сделано так: `BIEW-561/BIEWlib/sysdep/ia32/os2/timer.c`. Ага, путь начинается с имени каталога, в который должна быть распакована программа (в данном случае это имя — `BIEW-561`). Но можем ли мы переименовать его? Ведь многие хакеры предпочитают короткие имена файлов в стиле "`bw`". Ключ `-p1` заставляет утилиту `patch` игнорировать первое слева имя в цепочке, и тогда путь начинается с `/BIEWlib`, при этом естественно, каталог `BIEW-561` (как уже говорилось, его можно и переименовать) должен быть текущим. Если же мы накладываем заплатку извне каталога `BIEW-561`, необходимо указать ключ `-p0`. Отсутствие ключа `-p` приводит к полному игнорированию путей, и все файлы ищутся в текущем каталоге, где они, естественно, найдены не будут!

> ***Примечание***
>
> Установка заплатки — обратимая операция, и при желании заплатку можно удалить, воспользовавшись ключом `-R`, возвращающим все измененные строки на место. Кроме того, обратите внимание на ключ `-b`, создающий резервные копии измененяемых файлов.

Иногда к одной версии прилагается сразу несколько заплаток, что может серьезно озадачить даже бывалых пользователей. Внимательно прочитайте описание: в каком порядке их следует устанавливать! Если же описание отсутствует — смотрите на изменения и разбирайтесь с порядком наложения самостоятельно, или же вовсе откажитесь от установки. В экзотических случаях заплатка представляет собой скрипт, выполняющий все изменения самостоятельно.

Многие разработчики прилагают к архивам цифровые подписи типа PGP или эталонные контрольные суммы. Теоретически это позволяет предотвратить возможное искажение информации или ее подделку. Современные архиваторы контролируют целостность данных самостоятельно, а от преднамеренного взлома никакая цифровая подпись не спасет! Так что решайте сами — использовать их или нет, а мы приступаем к главному — к компиляции.

Приступаем к сборке

В среде пользователей Linux бытует мнение, что если программа не собирается "стандартным" путем (например, как показано в листинге 3.7), то это неправильная программа, и работать она будет неправильно. На самом деле, неправильных пользователей на свете гораздо больше, чем неправильных программ!

Листинг 3.7. Типовой порядок сборки большинства программ

```
$./configure
$make
$make install
```

Начнем с того, что в отличие от мира Windows, где программа устанавливается/собирается путем запуска программ setup.exe и nmake.exe, соответственно, в UNIX процесс сборки начинается... с чтения документации! Читать документацию обязательно! Даже если сборка с настройками по умолчанию пройдет без сучка и задоринки, полученная конфигурация вряд ли будет оптимальной.

Обычно к исходным текстам прилагается файл, называющийся install, readme и т. п. Если же архив не содержит ничего подобного (как, например, в случае с Bochs), то ищите инструкцию по сборке на сайте проекта. В особо тяжелых случаях инструкция может находиться внутри файлов configure и makefile.

Файл configure представляет собой достаточно сложный скрипт (рис. 3.11), анализирующий текущую конфигурацию, распознающий платформу, определяющий наличие всех необходимых библиотек и управляющий опциями сборки (в частности, в нем указывается, какие функции должны быть включены, а какие — блокированы). В результате его работы генерируется файл с именем makefile, который и собирает (компилирует, затем компонует) программу.

Некоторые конфигураторы имеют продвинутый интерфейс и работают в интерактивном режиме, но это не правило, а скорее приятное исключение. Гораздо чаще опции сборки задаются через ключи командной строки или даже путем правки самого конфигурационного файла.

Рис. 3.11. Конфигуратор за работой

Самая важная опция компиляции — это платформа. Под UNIX-системами она в большинстве случаев распознается автоматически, и никаких проблем при этом не возникает, но вот при сборке под MacOS, BeOS, QNX, Win32 вы можете встретиться с серьезными сложностями. Несмотря на то, что разработчики стремятся обеспечить максимальную переносимость, на практике все происходит совсем не так. Больше всего страдают пользователи Windows, так как эта платформа не поддерживает shell-скриптов, и конфигуратор там не работает. Даже если разработчик предусмотрел возможность компиляции под Win32, управлять опциями проекта приходится вручную, путем правки make-файла, а для этого еще необходимо разобраться, какая строка за что отвечает. Положение частично спасает пакет cygwin[6] (если, конечно, он установлен), но проблемы все равно остаются.

Остальные опции имеют уже не столь решающее значение, но и к второстепенным их не отнесешь. Сборка с настойками по умолчанию гарантирует, что программа соберется правильно и, может быть, даже заработает. Однако поддержки нужных вам режимов в этой сборке может и не оказаться. В частности, уже не раз упомянутый Bochs по умолчанию собирается без эмуляции SoundBlaster, без поддержки сетевой карты, наборов инструкций SSE/MMX, архитектуры x86-64, без интегрированного отладчика и без оптимизации скорости выполнения виртуального кода. Можно, конечно, бездумно активировать все опции, но это — далеко не лучшая идея. Во-первых, многие опции конфликтуют друг с другом, а во-вторых, дополнительные компоненты не только увеличивают размер откомпилированного файла, но и зачастую замедляют скорость работы программы. Поэтому, составляя "меню", необходимо быть очень внимательным и предусмотрительным. В частности, заставить Bochs поддерживать архитектуру x86-64 вместе с интегрированным отладчиком можно так, как показано в листинге 3.8.

Листинг 3.8. Задание опций сборки Bochs посредством командной строки

```
$./configure --enable-x86-64 --enable-debugger
```

А что делать, если в документации нет никакого упоминания о сборочных опциях (или есть, но неполное)? Тогда следует открыть файл configure в любом текстовом редакторе и просмотреть доступные опции. Если вам повезет, то они будут снабжены комментариями (листинг 3.9). Как вариант, можно попробовать дать команду `$./configure --help` — если вам повезет, она выведет хоть какую-нибудь справочную информацию.

Листинг 3.9. Фрагмент файла configure с опциями сборки

```
--enable-processors      select number of processors (1,2,4,8)
--enable-x86-64          compile in support for x86-64 instructions
--enable-cpu-level       select cpu level (3,4,5,6)
--enable-apic            enable APIC support
--enable-compressed-hd   allows compressed zlib hard disk image (not implemented yet)
--enable-ne2000          enable limited ne2000 support
--enable-pci             enable limited i440FX PCI support
--enable-pcidev          enable PCI host device mapping support (linux host only)
--enable-usb             enable limited USB support
--enable-pnic            enable PCI pseudo NIC support
```

[6] Cygwin — набор свободно распространяемых программных средств, разработанных фирмой Cygnus Solutions (в ноябре 1999 компания Cygnus Solutions объявила о своем слиянии с Red Hat и прекратила свое существование в начале 2000 года). Пакет Cygwin позволяет превратить Windows различных версий в некое подобие UNIX-системы. Его можно свободно скачать из Интернета (**http://www.cygwin.com/**).

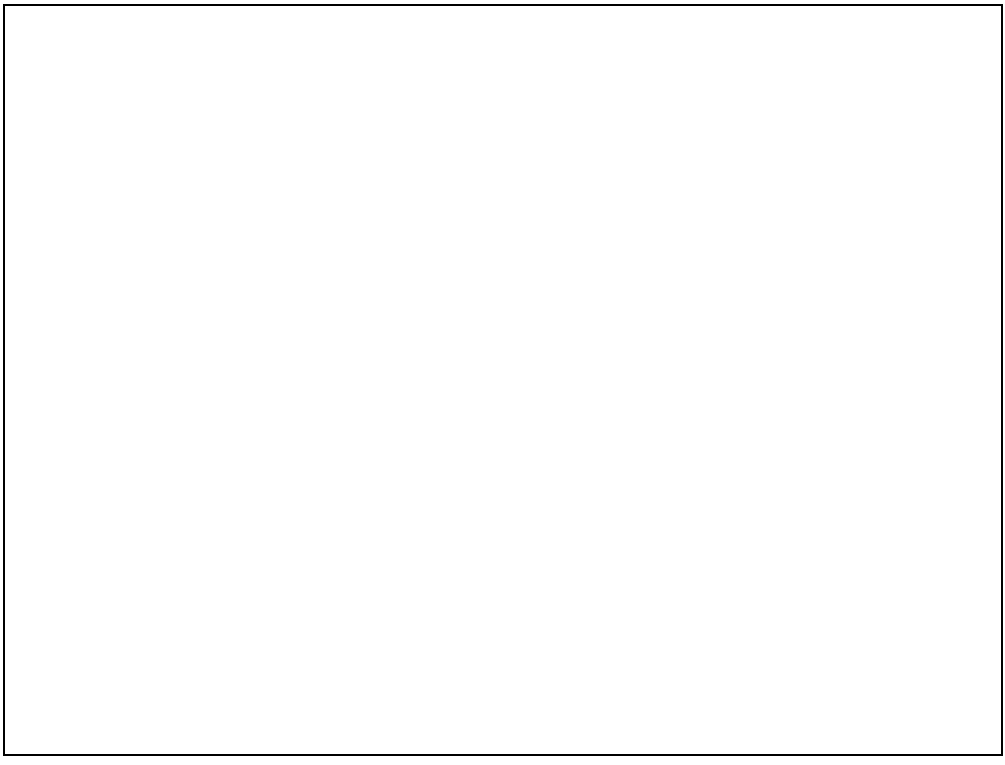

Рис. 3.12. Конфигурирование программы посредством правки make-файла

Некоторые программы (например, hex-редактор BIEW) вообще не имеют configure-файла. Это значит, что настраивать программу приходится вручную, путем редактирования файла makefile (рис. 3.12). Но не пугайтесь, на практике эта задача окажется намного проще, чем может показаться на первый взгляд. Структура файла makefile довольно проста. Фактически этот файл представляет собой последовательность команд и переменных (листинг 3.10). Вот переменными-то мы и будем управлять! Перечень возможных значений обычно содержится тут же, в комментариях.

Листинг 3.10. Фрагмент make-файла с опциями и комментариями

```
# Please select target platform. Valid values are:
# For 16-bit Intel: i86 i286 ( still is not supported by gcc )
# For 32-bit Intel
#        basic    : i386 i486
#        gcc-2.9x : i586 i686 p3 p4 k6 k6_2 athlon
#        pgcc     : i586mmx i686mmx p3mmx p4mmx k5 k6mmx k6_2mmx 6x86 6x86mmx
#                   athlon_mmx
# Other platform  : generic
#-----------------------------------------------------------
TARGET_PLATFORM=i386

# Please select target operating system. Valid values are:
# dos, os2, win32, linux, unix, beos, qnx4, qnx6
#-----------------------------------------------------------
TARGET_OS=unix
```

```
# Please add any host specific flags here
# (like -fcall-used-R -fcall-saved-R -mrtd -mregparm=3 -mreg-alloc=  e.t.c ;-):
#-------------------------------------------------------------------------
# Notes: You can also define -D__EXPERIMENTAL_VERSION flag, if you want to
# build experimental version with fastcall technology.
# *************************************************************************
# You can also define:
# -DHAVE_MMX     mostly common for all cpu since Pentium-MMX and compatible
# -DHAVE_MMX2    exists on K7+ and P3+
# -DHAVE_SSE     exists only on P3+
# -DHAVE_SSE2    exists only on P4+
# -DHAVE_3DNOW   exists only on AMD's K6-2+
# -DHAVE_3DNOWEX exists only on AMD's K7+
# *************************************************************************
# -D__DISABLE_ASM disables all inline assembly code.
# Try it if you have problems with compilation due to assembler errors.
# Note that it is not the same as specifying TARGET_PLATFORM=generic.
#-------------------------------------------------------------------------
HOST_CFLAGS=
```

Нужно заранее приготовиться к тому, что часть переменных окажется привязанной к рабочей среде (окружению) автора. Эта привязка выражается в том, что файл будет содержать абсолютные пути к целевым каталогам, включаемым файлам, библиотекам и т. д., или же эти переменные окажутся неинициализированными. В этом случае вам придется задать их самостоятельно. Некоторые make-файлы управляются через переменные окружения, которые опять-таки необходимо задать перед компиляцией (листинг 3.11).

Листинг 3.11. Фрагмент make-файла, управляемого через переменные окружения

```
PDCURSES_HOME                =$(PDCURSES_SRCDIR)
```

И вот, наконец, наступает торжественный момент, когда все опции настроены и можно приступать к сборке. С замиранием сердца пишем `$make`, и процесс сборки начнется. Компиляция длится долго, и достаточно часто она прерывается сообщением об ошибке. Что делать? Главное — не паниковать, а внимательно прочесть и проанализировать выведенное сообщение. Чаще всего причина ошибки заключается в том, что программе не хватает какой-нибудь библиотеки или заголовочного файла. Вообще-то, это должен быть выявить конфигуратор (если только он есть), но скачать недостающие компоненты при наличии Интернета — не проблема. Знать бы только, что именно надо скачать! К сожалению, далеко не всегда makefile сообщает "официальное" название библиотеки. Но и это еще не беда! Поищите в Интернете по имени файла и выясните, какому пакету он принадлежит и откуда его можно скачать. Кроме того, неплохо заглянуть на форум техподдержки (наверняка не вы одни столкнулись с этой ошибкой, а раз так, то этот вопрос должен обсуждаться на различных форумах). Если и это не поможет — прочтите документацию еще раз, обращая внимание на то, какие библиотеки и системные компоненты должны быть установлены. Наконец, попробуйте различные комбинации опций сборки (как вариант — отключите все, что только можно отключить).

Ситуация становится хуже, если вы столкнетесь с ошибками самих разработчиков. Ведь make-файлы тоже люди пишут, и далеко не на всех платформах их тестируют. Если так — попробуйте связаться с разработчиками или соберите программу на другой платформе (другим компилятором).

По умолчанию, программы, как правило, собираются с отладочной информацией, что существенно упрощает их отладку, но вместе с тем и увеличивает размер. Если отладка чужих программ

не входит в ваши планы, то отладочную информацию лучше убрать. Это можно сделать либо на стадии конфигурации, дав команду `$./configure --disable-debug`, либо вручную вырезать отладочную информацию из ELF-файла, пропустив его через утилиту `strip`, которая входит в большинства дистрибутивов UNIX. Но перед этим запустите программу `file` и посмотрите, как обстоят дела. Пример, иллюстрирующий ситуацию с Bochs, приведен в листинге 3.12.

Листинг 3.12. Эмулятор Bochs, собранный с настройками по умолчанию, содержит отладочную информацию

```
$file bochs
bochs: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.0,
dynamically linked (uses shared libs), not stripped
```

Поскольку отладочная информация присутствует (not stripped), Bochs занимает целых 9 Мбайт (листинг 3.13).

Листинг 3.13. С отладочной информацией bochs занимает целых 9 Мбайт

```
$ls -l bochs
итого 15052
-rw-r--r--  1 root staff     138 2006-03-21 05:24 1
-rw-r--r--  1 root staff       0 2006-03-21 05:25 2
-rwxr-xr-x  1 root staff 9407192 2006-03-20 20:44 bochs
-rwxr-xr-x  1 root staff   36966 2006-03-20 20:44 bxcommit
-rwxr-xr-x  1 root staff   37697 2006-03-20 20:44 bximage
-rwxr-xr-x  1 root staff 5390592 2006-02-19 22:12 elinks
-rwxr-xr-x  1 root staff  488395 2006-02-19 21:58 js
```

Итак, запустим утилиту strip и удалим отладочную информацию (листинг 3.14).

Листинг 3.14. Утилита strip удаляет отладочную информацию из файла

```
$strip bochs
```

После удаления отладочной информации размер файла сокращается в девять (!) раз, причем без какой бы то ни было потери функциональности (листинг 3.15).

Листинг 3.15. После удаления отладочной информации размер файла Bochs уменьшается до 1 Мбайта

```
$file bochs
bochs: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.0,
dynamically linked (uses shared libs), stripped

$ls -l bochs
итого 6844
-rw-r--r--  1 root staff     138 2006-03-21 05:24 1
-rw-r--r--  1 root staff     399 2006-03-21 05:25 2
-rw-r--r--  1 root staff       0 2006-03-21 05:25 3
-rwxr-xr-x  1 root staff 1009368 2006-03-21 05:25 bochs
-rwxr-xr-x  1 root staff   36966 2006-03-20 20:44 bxcommit
-rwxr-xr-x  1 root staff   37697 2006-03-20 20:44 bximage
-rwxr-xr-x  1 root staff 5390592 2006-02-19 22:12 elinks
-rwxr-xr-x  1 root staff  488395 2006-02-19 21:58 js
```

Инсталляция

Откомпилированная программа, как правило, еще не готова к работе. Нам предстоит еще много работы: удалить промежуточные файлы, созданные в процессе компиляции (библиотеки, объектные файлы), настроить конфигурационные файлы, разместить файлы данных по своим каталогам, а при необходимости — изменить системные настройки.

За это отвечает команда `$make install`, однако далеко не во всех программах она реализована, взять, например, хотя бы тот же BIEW (листинг 3.16).

Листинг 3.16. Фрагмент make-файла из BIEW показывает, что автоматическая инсталляция не реализована

```
install:
        @echo Sorry! This operation should be performed manually for now!
        @exit
```

С другой стороны, автоматическая инсталляция — это рулетка. Все мы знаем, во что способен превратить систему кривой setup.exe. Поэтому, прежде чем вводить команду `$make install`, неплохо бы заглянуть в файл makefile (раздел `install:`) и посмотреть, что он собирается делать. Вдруг это вас не устраивает?

Если вам не повезло с автоматической инсталляцией, попробуйте дать команду `$make uninstall`, удаляющую программу из системы — вдруг повезет? Однако в подавляющем большинстве случаев она не реализована.

Кроме того, существует полезная утилита CheckInstall (**http://checkinstall.izto.org/**). Это — бесплатно распространяемая утилита, трассирующая `$make install` на виртуальной машине и автоматически генерирующая полноценный "дистрибутив" любого типа: Slackware, RPM или Debian, устанавливаемый в систему соответствующим менеджером инсталляций, который всегда может сделать корректный uninstall, даже если он не был предусмотрен автором программы. Просто вместо `$make install` следует дать команду `$sudo checkinstall` и немного подождать...

Настоящие хакеры предпочитают устанавливать программу руками, используя готовую бинарную сборку как образец. Это самый надежный способ, однако требующий времени и квалификации.

Кстати говоря, большинство инсталляторов помещают программы в каталог /usr/local/bin/, что не всем нравится. Правильные конфигураторы поддерживают ключ `--prefix`, позволяющий устанавливать программы куда угодно (например, `$./configure --prefix=/usr`), неправильные заставляют нас это делать вручную.

Заключение

Вот, оказывается, какой нетривиальный процесс представляет собой ручная сборка! Ручная компиляция — это дверь в мир практически неограниченных возможностей, однако попасть в него может только тот, кто не боится сложностей, готов совершать ошибки, умеет работать с документацией и движется вперед даже под проливным дождем.

Глава 4

Ассемблеры

Низкоуровневое программирование — это разговор с компьютером на естественном для него языке, радость общения с "голым" железом, высший пилотаж полета свободной мысли и безграничное пространство для самовыражения. Вопреки распространенному мнению, ассемблер намного проще большинства языков высокого уровня. Он значительно проще, чем C++, и овладеть им можно буквально в течение нескольких месяцев. Все, что для этого требуется — это взять правильный старт и уверенно продвигаться в нужном направлении, а не петлять во тьме наугад.

Хакер, не знающий ассемблера, — это все равно, что гребец без весла. На языках высокого уровня далеко не уедешь. Чтобы взломать приложение, исходные тексты которого недоступны (а в подавляющем большинстве случаев дело обстоит именно так), необходимо проанализировать его алгоритм, растворенный в дебрях машинного кода. Существует множество переводчиков машинного кода на язык ассемблера (они называются дизассемблерами), но автоматически восстановить исходный текст по машинному коду невозможно!

Поиск недокументированных возможностей в недрах операционной системы также ведется на ассемблере. Поиск закладок, обезвреживание вирусов, адаптация приложений под собственные нужды, обратная разработка приложений с целью заимствования реализованных в них идей, рассекречивание засекреченных алгоритмов. Перечисление можно продолжать бесконечно. Сфера применения ассемблера настолько широка, что проще перечислить области, к которым он не имеет никакого отношения.

Ассемблер — это мощное оружие, дающее безграничную власть над системой. Это отнюдь не заумная теория, а самый настоящий хардкор. Самомодифицирующийся код, технологии полиморфизма, противодействие отладчикам и дизассемблерам, эксплоиты, генетически модифицированные черви, шпионаж за системными событиями, перехват паролей…

Ассемблер — это седьмое чувство и второе зрение. Когда на экране всплывает хорошо известное окно с воплем о критической ошибке, прикладные программисты лишь ругаются и разводят руками (это карма у программы такая). Все эти сообщения и дампы для них — китайская грамота. Но не для ассемблерщиков! Эти ребята спокойно идут по указанному адресу и правят баг, зачастую даже без потери несохраненных данных!

Философия ассемблера

Ассемблер — это низкоуровневый язык, оперирующий машинными понятиями и концепциями. Не ищите в нем команду для вывода строки "Hello, world!". Здесь ее нет. Вот краткий перечень действий, которые может выполнить процессор: сложить/вычесть/разделить/умножить/сравнить два числа и, в зависимости от полученного результата, передать управление на ту или иную ветку, переслать число с одного адреса по другому, записать число в порт или прочитать его оттуда. Управление периферией осуществляется именно через порты или через специальную область

памяти (например, видеопамять). Чтобы вывести символ на терминал, необходимо обратиться к технической документации на видеокарту, а чтобы прочитать сектор с диска — к документации по накопителю. К счастью, эту часть работы берут на себя драйверы, и выполнять ее вручную обычно не требуется (к тому же, в нормальных операционных системах, например, таких как семейство Windows NT, с прикладного уровня порты недоступны).

Другой машинной концепцией является *регистр*. Объяснить, что это такое, не погрешив против истины, невозможно. Регистр — это нечто такое, что выглядит как регистр, но таковым в действительности не является. В древних машинах регистр был частью устройства обработки данных. Процессор не может сложить два числа, находящихся в оперативной памяти. Сначала он должен взять их в руки (регистры). Это — на микроуровне. Над микроуровнем расположен интерпретатор машинных кодов, без которого не обходится ни один из современных процессоров (да-да, машинные коды интерпретируются!). Мини-ЭВМ DEC PDP-11 уже не требовала от программиста предварительной загрузки данных в регистры, делая вид, что берет их прямо из памяти. На самом же деле, данные скрытно загружались во внутренние регистры, а после выполнения арифметических операций результат записывался в память или в… "логический" регистр, представляющий собой ячейку сверхбыстрой памяти.

В компьютерах x86 регистры также виртуальны, но в отличие от PDP, частично сохранили свою специализацию. Некоторые команды (например, `mul`) работают со строго определенным набором регистров, который не может быть изменен. Это — плата за совместимость со старыми версиями. Другое неприятное ограничение состоит в том, что x86 не поддерживает адресации типа "память — память", и одно из обрабатываемых чисел обязательно должно находиться в регистре или представлять собой непосредственное значение. Фактически, ассемблерная программа наполовину состоит из команд пересылки данных.

Все эти действия происходят на арене, называемой *адресным пространством*. Адресное пространство — это просто совокупность ячеек виртуальной памяти, доступной процессору. Операционные системы типа Windows 9*x* и большинство клонов UNIX создают для каждого приложения свой независимый 4-Гбайтный регион, в котором можно выделить, по меньшей мере, три области: область *кода*, область *данных* и *стек*.

Стек — это способ хранения данных, представляющий собой нечто среднее между списком и массивом (читайте "Искусство программирования" Д. Кнута[1]). Команда `push` помещает новую порцию данных на вершину стека, а команда `pop` — снимает данные с вершины стека. Это позволяет сохранять данные в памяти, не заботясь об их абсолютных адресах. Очень удобно! Вызов функций происходит именно так. Команда `call func` забрасывает в стек адрес следующей за ней команды, а команда `ret` снимает его со стека. Указатель на текущую вершину хранится в регистре `ESP`, а дно… формально стек ограничен лишь протяженностью адресного пространства, а так — количеством выделенной ему памяти. Направление роста стека в архитектуре x86: от старших адресов — к младшим. Еще говорят, что стек растет "сверху вниз".

Регистр `EIP` содержит указатель на следующую выполняемую команду и недоступен для непосредственной модификации. Регистры `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` называются регистрами общего назначения и могут свободно участвовать в любых математических операциях или операциях обращения к памяти. Их всего семь. Семь 32-разрядных регистров. Первые четыре из них (`EAX`, `EBX`, `ECX` и `EDX`) допускают обращения к своим 16-разрядным половинкам, хранящим младшее слово — `AX`, `BX`, `CX` и `DX`. Каждое из этих слов, в свою очередь, делится на старший и младший байты — `AH`/`AL`, `BH`/`BL`, `CH`/`CL` и `DH`/`DL`. Важно понять, что `AL`, `AX` и `EAX` — это не три разных регистра, а разные части одного и того же регистра!

Кроме того, существуют и другие регистры — сегментные, мультимедийные, регистры математического сопроцессора, отладочные регистры. Без хорошего справочника в них легко запутаться и утонуть, однако на первых порах мы их касаться не будем.

[1] Дональд Э. Кнут. "Искусство программирования", Т. 1—3. — "Вильямс", 2007.

Объяснение ассемблера на примерах C

Основной ассемблерной командой является команда пересылки данных `mov`, которую можно уподобить оператору присваивания. `c = 0x333` на языке ассемблера записывается так: `mov eax, 333h` (обратите внимание на различие в записи шестнадцатеричных чисел!). Можно также записать `mov eax, ebx` (записать в регистр `EAX` значение регистра `EBX`).

Указатели заключаются в квадратные скобки. Таким образом, конструкция языка C `a = *b` на ассемблере записывается так: `mov eax, [ebx]`. При желании, к указателю можно добавить смещение: `a = b[0x66]` эквивалентно: `mov eax, [ebx + 66h]`.

Переменные объявляются директивами `db` (однобайтная переменная), `dw` (переменная длиной в одно слово), `dd` (переменная длиной в двойное слово) и т. д. Знаковость переменных при их объявлении не указывается. Одна и та же переменная в различных участках программы может интерпретироваться и как число со знаком, и как беззнаковое число. Для загрузки переменной в указатель применяется либо команда `lea`, либо `mov` с директивой `offset`. Покажем это на следующем примере (листинг 4.1).

Листинг 4.1. Основные методы пересылок данных

```
LEA EDX,b             ; Регистр EDX содержит указатель на переменную b
MOV EBX,a             ; Регистр EBX содержит значение переменной a
MOV ECX, offset a     ; Регистр ECX содержит указатель на переменную a

MOV [EDX],EBX         ; Скопировать переменную a в переменную b

MOV b, EBX            ; Скопировать переменную a в переменную b

MOV b, a              ; !!!Ошибка!!! так делать нельзя!!!
                      ; Оба аргумента команды MOV не могут быть в памяти!

a DD 66h              ; Объявляем переменную a типа двойного слова и
                      ; Инициализируем ее числом 66h
b DD ?                ; Объявляем неинициализированную переменную b типа
                      ; двойного слова
```

Теперь перейдем к условным переходам. Никакого оператора `if` на ассемблере нет, и эту операцию приходится осуществлять в два этапа. Команда `cmp` позволяет сравнить два числа, сохраняя результат своей работы во флагах. *Флаги* — это биты специального регистра, описание которого заняло бы слишком много места и поэтому здесь не рассматривается. Достаточно запомнить три основных состояния: меньше (below или less), больше (above или great) и равно (equal). Семейство команд условного перехода `jx` проверяют условие `x` и, если оно истинно, совершают переход (jump) по указанному адресу. Например, `je` выполняет переход, если числа равны (Jump if Equal), а `jne` — если эти числа не равны (Jump if Not Equal). Команды `jb/ja` работают с беззнаковыми числами, а с `jl/jg` — со знаковыми. Любые два не противоречащих друг другу условия могут быть скомбинированы друг с другом. Например, команда `jbe` осуществляет переход, если одно беззнаковое число меньше другого или равно ему. Безусловный переход осуществляется командой `jmp`.

Конструкция `cmp/jx` больше похожа на конструкцию `IF` *xxx* `GOTO` языка Basic, чем на C. Несколько примеров ее использования продемонстрированы в листинге 4.2.

Листинг 4.2. Основные типы условных переходов

```
CMP EAX, EBX      ; Сравнить EAX и EBX
JZ xxx            ; Если они равны, сделать переход на xxx

CMP [ECX], EDX    ; Сравнить *ECX и EDX
JAE yyy           ; Если беззнаковый *ECX >= EDX, то перейти на yyy
```

Вызов функций на ассемблере реализуется намного сложнее, чем на C. Во-первых, существует по меньшей мере два типа соглашений о вызовах (calling conventions) — C и Pascal. В соглашении C аргументы передаются функции справа налево, а из стека их вычищает вызывающий функцию код. В соглашении Pascal все происходит наоборот! Аргументы передаются слева направо, а из стека их вычищает сама функция. Большинство функций API операционный системы Windows придерживаются комбинированного соглашения `stdcall`, при котором аргументы заносятся в соответствии с соглашением C, а из стека вычищаются по соглашению Pascal. Возвращаемое функцией значение помещается в регистр `EAX`, а для передачи 64-разрядных значений используется регистровая пара `EDX:EAX`. Разумеется, этих соглашений необходимо придерживаться только при вызове внешних функций (API, библиотек и т. д.). "Внутренние" функции им следовать не обязаны и могут передавать аргументы любым мыслимым способом, например, через регистры. В листинге 4.3 приведен простейший пример вызова функции.

Листинг 4.3. Вызов функций API операционной системы

```
PUSH offset LibName     ;// Засылаем в стек смещение строки
CALL LoadLibrary        ;// Вызов функции
MOV h, EAX              ;// EAX содержит возращенное значение
```

Ассемблерные вставки как тестовый стенд

Как же сложно программировать на чистом ассемблере! Минимально работающая программа содержит множество разнообразных конструкций, сложным образом взаимодействующих друг с другом и открывающих огонь без предупреждения. Одним махом мы отрезаем себя от привычного окружения. Сложить два числа на ассемблере не проблема, но вот вывести результат этой операции на экран…

Ассемблерные вставки — другое дело. В отличие от классических руководств по ассемблеру (Зубков, Юров), которые буквально с первых же строк бросают читателя в пучину системного программирования, устрашая его ужасающей сложностью архитектуры процессора и операционной системы, ассемблерные вставки оставляют читателя в привычном для него окружении языков высокого уровня (C и/или Pascal). При их использовании знакомство с внутренним миром процессора происходит постепенно, безо всяких резких скачков. Кроме того, ассемблерные вставки позволяют начать изучение непосредственно ассемблера с 32-разрядного защищенного режима процессора. Дело в том, что в чистом виде защищенный режим чрезвычайно сложен для освоения, и потому практически все руководства начинают изложение с описания морально устаревшего 16-разрядного реального режима, что не только оказывается бесполезным балластом, но и замечательным средством запутывания ученика (помните, "забудьте все, чему вас учили раньше…"). Методика обучения на основе ассемблерных вставок превосходит все остальные как минимум по двум категориям:

- Скорость — буквально через три-четыре дня интенсивных занятий человек, ранее никогда не знавший ассемблера, начинает вполне сносно на нем программировать.
- Легкость освоения — изучение ассемблера происходит практически безо всякого напряжения и усилий. Ни на каком из этапов обучения ученика не заваливают ворохом неподъемной и непроходимой информации, а каждый последующий шаг интуитивно понятен.

Ну так чего же мы ждем? Для объявления ассемблерных вставок в Microsoft Visual C++ служит ключевое слово `__asm`. Пример простейшей программы, использующей ассемблерную вставку, приведен в листинге 4.4.

Листинг 4.4. Ассемблерная вставка для сложения двух чисел

```
main()
{
    int a = 1;          // Объявляем переменную a и кладем туда значение 1
    int b = 2;          // Объявляем переменную a и кладем туда значение 1
    int c;              // Объявляем переменную c, но не инициализируем ее

    // Начало ассемблерной вставки
        __asm{
        mov eax, a      ; Загружаем значение переменной a в регистр EAX
        mov ebx, b      ; Загружаем значение переменной b в регистр EBX
        add eax, ebx    ; Складываем EAX с EBX, записывая результат в EAX
        mov c, eax      ; Загружаем значение EAX в переменную c
        }
    // Конец ассемблерной вставки

    // Выводим содержимое c на экран
    // с помощью привычной для нас функции printf
    printf("a + b = %x + %x = %x\n",a,b,c);
}
```

Необходимый инструментарий

Чтобы программировать с помощью ассемблерных вставок, вам потребуется компилятор с интегрированной средой разработчика (например, Microsoft Visual Studio). Ассемблерные вставки отлаживаются точно так же, как и код, написанный на языке высокого уровня, что очень удобно.

Программы, написанные непосредственно на языке ассемблера, транслируются в машинный код с помощью транслятора ассемблера. Более подробная информация о доступных ассемблерных трансляторах будет приведена в следующем разделе этой главы, *"Сравнение ассемблерных трансляторов"*. Оттранслированную ассемблерную программу необходимо обработать компоновщиком. Для компоновки ассемблированных программ можно использовать стандартный компоновщик, входящий в состав Microsoft Visual Studio или продукта Microsoft Platform SDK. Среди нестандартных компоновщиков лучшим является Ulink, разработанный Юрием Хароном. Он поддерживает большинство форматов файлов и предоставляет множество опций, недоступных в других компоновщиках. Скачать его можно по адресу **ftp://ftp.styx.cabel.net/pub/UniLink/**. Для некоммерческих целей Ulink бесплатен.

Наконец, вам потребуются отладчик и дизассемблер — для поиска ошибок в ваших собственных программах и взлома чужих приложений. В отношении отладчиков у вас имеется широкий выбор: Microsoft Visual Debugger, интегрированный в состав Microsoft Visual Studio, Microsoft Windows Debugger (WDB); Kernel Debugger, поставляемый в составе SDK и DDK; SoftICE; OllyDbg и т. д. Из дизассемблеров следует отдать предпочтение IDA Pro, как бесспорному лидеру.

Кроме того, рекомендуется добавить в свой арсенал и другие хакерские средства, краткий обзор которых был приведен в *главе 1*, *"Инструментарий хакера"*.

Сравнение ассемблерных трансляторов

Проблема выбора "единственно правильного" ассемблерного транслятора мучает не только начинающих, но и профессиональных программистов. У каждого продукта есть своя когорта поклонников, и спор о преимуществах и недостатках рискует превратиться в "священные войны". На форумах такие дискуссии лучше не разводить. В этом разделе будут кратко рассмотрены наиболее популярные ассемблеры (MASM, TASM, FASM, NASM, YASM) по широкому спектру критериев, значимость которых каждый должен оценивать сам.

Компиляторы языков высокого уровня (C, Pascal) в определенной степени совместимы между собой. Хотя исходный текст, предназначенный для одного компилятора, не всегда без переделок транслируется на другом, синтаксис и прочие языковые концепции остаются неизменными. За счет этого программисты и могут "летать" между Microsoft Visual C++, Intel C++, GCC, Open Watcom[2], сравнивая полноту поддержки Стандарта, скорость трансляции, качество кодогенерации, популярность компилятора, количество библиотек и дополнительных компонентов к нему.

С трансляторами ассемблера все обстоит иначе. Казалось бы, ассемблер стандартной архитектуры x86 и трансляторы должен поддерживать стандартные. Однако, в действительности, это не так. Помимо поддержки мнемоник машинных команд, каждый транслятор обладает собственным набором директив и макросредств, зачастую ни с чем не совместимых. Ассемблерный листинг, написанный для трансляции с помощью MASM, бесполезно переносить на FASM, поскольку возможности, предоставляемые макросредствами, у них сильно различаются.

Перефразируя известную поговорку, можно сказать, что, выбирая ассемблерный транслятор, вы выбираете судьбу, изменить которую впоследствии за здорово живешь не удастся! Придется переучиваться, фактически осваивая новый язык. Но это еще полбеды! Всякий уважающий себя программист со временем обрастает ворохом разнокалиберных библиотек, выполняющих всю грязную работу. Их-то куда при переходе на новый ассемблер девать?! Перенести ничуть не легче, чем переписать с нуля!

К счастью, ассемблер — это только инструмент, а не религия, и совместное использование нескольких трансляторов еще никто не запрещал. На практике, обычно, так и поступают. Ставится конкретная задача, и для ее решения выбирается наиболее адекватный инструмент. Естественно, чтобы сделать правильный выбор, необходимо знать, какие ассемблеры вообще существуют и чем они отличаются друг от друга.

Основополагающие критерии

Богатый ассортимент — верный признак отсутствия продукта, который устраивает если не всех, то хотя бы большинство. Поэтому если некий продукт существует и не просто существует, но еще и собирает под свое крыло большое количество пользователей, значит, кому-то этот продукт действительно необходим.

Сравнивая ассемблеры друг с другом, невозможно найти "самый лучший" транслятор (таких просто не существует). В этом разделе просто собрана воедино вся информация о предоставляемых ими возможностях, ведь значимость любых критериев всегда субъективна по определению. Человеку, упорно игнорирующему существование Linux/BSD, абсолютно безразлично количество платформ, на которые был перенесен тот или иной транслятор. А для кого-то это — вопрос первостепенной важности!

Тем не менее, существует ряд основополагающих критериев, существенных для всех категорий программистов. Начнем с *генерации отладочной информации*, без которой отладка программы сложнее, чем "Hello, world", превращается в настоящую пытку. Вот тут некоторые уже пытаются возразить, что ассемблерам, в отличие от языков высокого уровня, отладочная информация не нужна, так как мнемоники машинных команд, что в листинге, что в отладчике — одни и те же. А метки?! А структуры?! А имена функций?! Уберите их — и код станет совершенно не читаемым! Можно, конечно, воспользоваться отладочной печатью: просто вставлять в интересующие вас точки программы макросы, выводящие на экран или в файл значения регистров/переменных. Давным-давно, когда интерактивных отладчиков еще не существовало, именно отладочная печать и была основным средством борьбы с багами. Еще можно отлаживать программу, держа перед собой распечатку исходных текстов, но это уже извращение.

[2] Open Watcom — проект по созданию компиляторов с открытым исходным кодом на базе коммерческого компилятора фирмы Watcom. Главная страница проекта — **http://www.openwatcom.org/index.php/Main_Page**.

Проблема заключается в том, что формат отладочной информации не стандартизован, и различные трансляторы используют различные форматы. Это ограничивает нас в выборе отладчиков или вынуждает использовать конверторы сторонних производителей. Причем стоит упомянуть, что некоторые ассемблеры (например, FASM) не генерируют отладочной информации вообще. Ну хоть бы простейший map-файл, эх…

Но если формат отладочной информации — это "задворки" транслятора, то *формат выходных файлов* — это его "лицо". Непосвященные только пожмут плечами. Какой там формат? Обыкновенный obj-файл, из которого с помощью компоновщика можно изготовить все, что угодно — от exe до dll. На самом деле, "обыкновенных" объектных файлов в природе не бывает. Существуют файлы формата omf[3] (в редакциях от Microsoft и IBM), coff[4], elf[5], a.out[6] и множества других экзотических форматов в стиле as86[7], rdf[8], ieee[9] и т. д. Также заслуживает внимания возможность "сквозной" генерации двоичных файлов, не требующая помощи со стороны компоновщика. А некоторые ассемблеры (например, FASM) даже позволяют "вручную" генерировать исполняемые файлы и динамические библиотеки различных форматов, полностью контролируя процесс их создания и заполняя ключевые поля по своему усмотрению. Впрочем, программы, целиком написанные на ассемблере, — это либо вирусы, либо демонстрационные программки, либо учебные программы. Как правило, на ассемблере пишутся лишь системно-зависимые компоненты или модули, критичные к быстродействию, которые затем линкуются к основному проекту. Поэтому если ассемблер генерирует только omf, а компилятор — coff, то возникает проблема сборки "разнокалиберных" форматов воедино. Лишь один компоновщик способен делать это — Ulink от Юрия Харона, обеспечивающий к тому же богатые возможности по сборке файлов "вручную". Поэтому выбор конкретного ассемблерного транслятора целиком лежит на совести (и компетенции) программиста, но все-таки лучше, чтобы и ассемблер, и компилятор генерировали объектные файлы одинаковых форматов.

Другой немаловажный критерий — *количество поддерживаемых процессорных архитектур*, которых в линейке x86 набралось уже больше десятка. Конечно, недостающие команды можно реализовать с помощью макросов или запрограммировать непосредственно в машинном коде через директиву `db`. Однако если так рассуждать, то зачем вообще нужны ассемблеры, когда есть hex-редакторы?! Особое внимание следует обратить на платформы AMD x86-64 и Intel IA64. Хотим мы этого или нет, но 64-разрядные архитектуры имеют хорошие шансы потеснить x86, поэтому учиться программировать под них обязательно, так что их поддержка со стороны транслятора должна быть обеспечена уже сейчас!

Кстати, ни один из трансляторов не поддерживает набор команд процессоров x86 в полном объеме. Например, на MASM невозможно написать `jmp 0007h:00000000h`, и приходится прибегать

---

[3] OMF — формат объектного модуля (Object Module Format).

[4] COFF — общий формат объектного модуля (Common Object File Format).

[5] ELF — формат исполнения и компоновки (Executable and Link Format).

[6] A.OUT — классический формат объектного кода UNIX (само название означает "Assembler Output"), в настоящее время вытесняется ELF (исполняемые модули расширений не имеют, объектные файлы имеют расширение .o).

[7] 16-битный ассемблер для Linux as86 имеет собственный нестандартный формат объектного файла. Хотя компоновщик для этого ассемблера ld86 генерирует файлы, похожие на стандартный формат a.out, формат объектного файла, используемого для взаимодействия между as86 и ld86, имеет формат, отличный от a.out.

[8] Файлы с расширением rdf представляют собой объектные файлы формата RDOFF (Relocatable Dynamic Object File Format). RDOFF — это формат объектного файла, разработанный для транслятора NASM.

[9] Формат IEEE (IEEE-695) используется на множестве платформ (в том числе Motorola 68000, Motorola 68HC08, Hitachi, Zilog) и для кросс-платформенного программирования.

к различным ухищрениям. Например, можно реализовать команду через `db`, но это неэлегантно и неудобно. Альтернативный вариант — занести в стек сегмент/смещение, а потом выполнить `retf`, но это — слишком громоздко и, к тому же, при этом подходе задействуется стек, которого у нас может и не быть.

Программирование на смеси 16- и 32-разрядного кода с кратковременным переходом в защищенный режим и возвращением обратно в реальный — это вообще песня. На MASM скорее умрешь, чем такое запрограммируешь, однако большинству программистов подобного трюкачества просто не нужно!

А вот что реально нужно большинству — так это *интеграция в мировое сообщество*. Разработкой собственных операционных систем обычно занимаются независимые исследователи-энтузиасты. В учебном плане это, бесспорно, очень даже хорошо, но коммерческим программистам обычно приходится программировать под уже существующие системы, например, ту же Windows. И если в состав DDK входит MASM и множество исходных текстов драйверов, то пытаться собрать их под другим транслятором — пустая трата времени. Опять-таки, если компилятору Microsoft Visual C++ задать ключ `/FA`, то он выдаст ассемблерный листинг в стиле MASM. Точно так же поступит и IDA Pro, Borland C++ выберет TASM (ну еще бы!), а GCC — GNU Assembler (он же GAS). Это и есть интеграция в среду. Чистый ассемблер сам по себе используется редко, и практически всегда он становится приложением к чему-то еще. То есть если вы пишете драйверы под Windows на Microsoft Visual C++, то разумнее всего остановить свой выбор на MASM, поклонникам же Borland C++ лучше TASM ничего не найти. Под Linux/BSD лидирует GAS (GNU Assembler), уже хотя бы за счет того, что ассемблерные программы можно транслировать с помощью компилятора GCC, используя препроцессор C и освобождаясь от головной боли с поиском стартового кода и библиотек. Однако, GAS использует синтаксис AT&T[10], являющийся полной противоположностью синтаксису Intel, которого придерживаются MASM, TASM, FASM, NASM/YASM. Разработчики вирусов и просто маленьких ассемблерных программ, написанных из любви к искусству, намного меньше ограничены в своем выборе и могут использовать все, что им по душе, вне зависимости от степени "поддержки".

Качество документации играет весьма важную роль. Не менее важно и то, кому принадлежит проект. Трансляторы, созданные энтузиастами-одиночками, могут умереть в любой момент, поэтому полагаться на них в долговременной перспективе не рекомендуется. Коммерческие ассемблеры крупных компаний выглядят намного более стабильными и непоколебимыми, однако нет никаких гарантий того, что в один "прекрасный" момент компания не прекратит поддержку своего продукта (достаточно вспомнить историю с TASM). Открытые трансляторы, поддерживаемые независимой группой лиц, наиболее живучи. Стоило коллективу NASM чуть-чуть приостановить его развитие, как тут же появился YASM — "позаимствовавший" исходные тексты и добавивший все необходимое (поддержку x86-64, формат отладочной информации CodeView и т. д.).

Последнее, на чем хотелось бы заострить внимание — это *макросредства*. Отношение к ним у программистов двоякое. Одни вовсю пользуются плодами прогресса, программируя на смеси ассемблера, бейсика и препроцессора C (существует даже проект HLA: High Level Assembler — Ассемблер высокого уровня), другие — презирают их, ратуя за чистоту ассемблерного кода. Вот и разберись тут, кто прав, а кто виноват! Макросы упрощают программирование, зачастую позволяя невозможное (например, шифровать код программы еще на стадии ассемблирования!), но переносимость программы от этого резко ухудшается, и переход на другой транслятор становится труднореализуемым. Но, как бы там ни было, поддержка макросов совсем не обязывает этими макросами пользоваться!

---

[10] В последнюю версию GAS, входящую в состав GNU binutils 2.17, была добавлена и поддержка синтаксиса языка ассемблера Intel (см. **http://sources.redhat.com/binutils/**).

MASM

Продукт жизнедеятельности ранней компании Microsoft, который был нужен ей для создания MS-DOS, а позднее — и для Windows 9*x*/NT. После выхода MASM версии 6.13 развитие продукта на некоторое время тормозилось, но затем здравый смысл взял вверх, и последняя версия (на момент написания этих строк — 6.13.8204) поддерживает Unicode, все расширения SSE/SSEII/SSEIII (объявляемые двумя директивами `.686`/`.XMM`), а также архитектуру AMD x86-64. Платформа Intel IA64 не поддерживается, но Microsoft поставляет Intel-ассемблер IAS.EXE.

Аббревиатура MASM расшифровывается отнюдь не как Microsoft Assembler, а как Macro Assembler, то есть "Ассемблер с поддержкой Макросов". Макросы покрывают своими возможностями широкий круг задач: повторения однотипных операций с параметризацией (шаблоны), циклические макросы, условное ассемблирование и т. д. Имеется даже зачаточная поддержка основных парадигм ООП, впрочем, так и не получившая большого распространения, поскольку ассемблер и ООП концептуально несовместимы. Многие программисты пишут вообще без макросов на чистом ассемблере, считая свой путь идеологически наиболее правильным. Но о вкусах не спорят.

Сначала MASM распространялся в виде самостоятельного (и притом весьма дорогостоящего) пакета, но впоследствии он был включен в состав продукта DDK, который вплоть до Windows 2000 DDK раздавался бесплатно, а сейчас доступен только подписчикам MSDN. Впрочем, вполне полноценный DDK (с ассемблером) для Windows Server 2003 входит в Kernel-Mode Driver Framework, а сам транслятор MASM — еще и в состав продукта Visual Studio Express, который также распространяется бесплатно.

Стив Хатчессон (Steve Hutchessen) собрал последние версии транслятора MASM, компоновщика от Microsoft, включаемые файлы, библиотеки, обширную документацию, статьи различных авторов, посвященные ассемблеру, и даже простенькую интегрированную среду разработки (IDE) в один дистрибутив, известный как "Пакет Хатча" (Hutch's package), бесплатно раздаваемый всем желающим на вполне лицензионной основе. Так что это не хак, а вполне удобный комплект инструментов для программирования под Windows на ассемблере (рис. 4.1).

Транслятору MASM посвящено множество книг, что упрощает процесс обучения, а в сети можно найти множество исходных текстов ассемблерных программ и библиотек, освобождающих программиста от необходимости изобретать велосипед. Кроме того, MASM является выходным языком для многих дизассемблеров (Sourcer, IDA Pro). Все это делает MASM транслятором номером один в программировании для платформы Wintel[11].

Поддерживаются два выходных формата: 16/32-битный Microsoft OMF и 32/64-битный COFF. Это позволяет транслировать следующие виды программ:

- 16/32-разрядные программы под MS-DOS, работающие в реальном и защищенном режиме,
- 16-разрядные приложения и драйверы для Windows 3.*x*,
- 32-разрядные приложения и драйверы для Windows 9*x*/NT,
- 64-разрядные приложения и драйверы для Windows NT 64-bit Edition.

Для создания бинарных файлов потребуется компоновщик, который способен это делать (например, Ulink от Юрия Харона). Кстати говоря, последние версии штатного компоновщика Microsoft, входящего в SDK и DDK, утратили способность собирать 16-разрядные файлы под MS-DOS/Windows 3.*x*. Поэтому, если у вас возникнет такая потребность, придется вернуться к старой версии, которая лежит в папке NTDDK\win_me\bin16.

MASM генерирует отладочную информацию в формате CodeView, которую Microsoft Linker может преобразовывать в формат Program Database (PDB). Хотя этот формат и не документирован, но он поддерживается библиотекой dbghelp.dll. Это позволяет сторонним разработчикам интерпретировать отладочную информацию, поэтому файлы, оттранслированные с помощью MASM, можно отлаживать в SoftICE, дизассемблировать в IDA Pro и прочих продуктах подобного типа.

[11] Wintel = Windows + Intel.

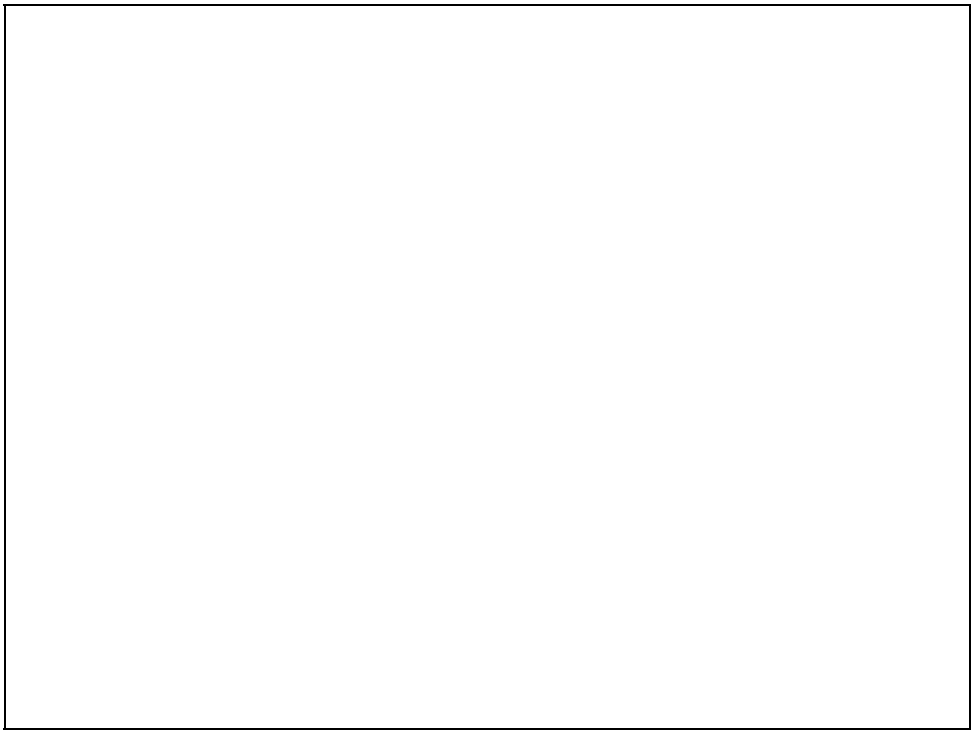

Рис. 4.1. Установка пакета Хатча

Главный недостаток MASM — это большое количество ошибок. Стоит только открыть Knowledge Base, посмотреть на список официально подтвержденных багов и… ужаснуться! Как только после этого на MASM вообще можно программировать?! Особенно много ошибок в штатной библиотеке. Вот лишь несколько примеров:

- Функции `dwtoa` и `atodw_ex` не распознают знака и по скорости очень тормозят, хотя в документации написано: "A high speed ascii decimal string to DWORD conversion for applications that require high speed streaming of conversion data"[12].
- Функция `ucFind` не находит в строке подстроку, если длина подстроки равна 1 символу.
- Функции `BMHBinsearch` и `SBMBinSearch`, осуществляющие поиск алгоритмом Бойера—Мура (Boyer—Moore search algorithm) реализованы с ошибками.
- Некоторые функции обрушивают программу (например, если передать функции `ustr2dw` строку длиннее пяти байт — программа падает).

Другой минус — отсутствие поддержки некоторых инструкций и режимов адресации процессора. Так, например, невозможно осуществить `jmp far seg:offset`, а попытка создания смешанного 16/32-разрядного кода — это настоящий кошмар, который приходится разгребать руками, преодолевая сопротивление "менталитета" транслятора.

Наконец, MASM — типичный продукт с закрытыми исходными текстами, судьба которых покрыта мраком. Microsoft интенсивно продвигает высокоуровневое программирование, отказываясь от ассемблера везде, где это только возможно, поэтому не исключено, что через несколько лет MASM прекратит свое существование...

[12] Высокая скорость преобразования десятичной строки ASCII в `DWORD` для приложений, нуждающихся в высокоскоростном потоке преобразуемых данных.

Тем не менее, несмотря на все эти недостатки, MASM остается самым популярным профессиональным транслятором ассемблера при программировании под линейку Windows NT. Хотя разработчикам и приходится жутко ругаться, но реальных альтернатив ему нет.

TASM

Самый популярный транслятор ассемблера времен MS-DOS, созданный фирмой Borland, полностью совместимый с MASM вплоть до версий 6.*x* и поддерживающий свой собственный режим IDEAL с большим количеством улучшений и расширений.

Удобство программирования, скромные системные требования и высокая скорость трансляции обеспечивали TASM[13] лидерство на протяжении всего существования MS-DOS. Но с появлением Windows популярность TASM стала таять буквально на глазах. Не сумев (или не захотев) добиться совместимости с заголовочными файлами и библиотеками, входящими в комплект SDK/DDK, фирма Borland решила поставлять свою собственную портированную версию, причем далекую от идеала. К тому же штатный компоновщик tlink/tlink32 не поддерживает возможности создания драйверов, а формат выходных файлов (Microsoft OMF, IBM OMF, PharLap), не поддерживается текущими версиями компоновщика Microsoft (впрочем, 16-битные версии на это способны). В довершение всего, формат отладочной информации не совместим с CodeView и реально поддерживается только Turbo Debugger и SoftICE.

Эти проблемы принципиально разрешимы, так как возможность низкоуровневого ассемблерного программирования (без включаемых файлов и макросов) сохранилась, а несовместимость форматов компенсируется наличием конверторов. Тем не менее, преимущества режима IDEAL над стандартным синтаксисам MASM день ото дня казались все менее и менее значительными, и ряды поклонников проекта редели, и в конце концов он был закрыт окончательно. Последней версией транслятора TASM стала версия 5.0, поддерживающая наборы команд процессоров Intel вплоть до 80486. Отдельно был выпушен патч, обновляющий TASM до версии 5.3 и поднимающий его вплоть до Pentium MMX, однако команды Pentium II (например, `SYSENTER`) как не работали, так и не работают. Поддержка Unicode тоже отсутствует.

В настоящее время Borland прекратила распространение своего ассемблера, и достать его можно только в магазинах, торгующих старыми CD-ROM, или у какого-нибудь коллекционера. Человек, известный под ником !tE, выпустил пакет TASM 5+, включающий транслятор, компоновщик, библиотеки, немного документации, несколько заголовочных файлов под Windows, а также пару демонстрационных примеров. Когда будете искать этот пакет, не перепутайте его с TASM32 фирмы Squak Valley Software — это совершенно независимый кросс-ассемблер, ориентированный на процессоры 6502,6800/6801/68HC11, 6805, TMS32010, TMS320C25, TMS7000, 8048, 8051, 8080/8085, Z80, 8096/80C196KC, о существовании которых большинство из нас в лучшем случае просто осведомлено.

В целом можно сказать, что TASM — это мертвый ассемблер. Однако для разработки приложений под Windows 16/32 и MS-DOS он все-таки подходит. Это особенно справедливо, если вы уже имеете опыт работы с ним и некоторые собственные наработки (библиотеки, макросы), с которыми жалко расставаться, а конвертировать под MASM — весьма проблематично. Возможно, вам понравится бесплатный Lazy Assembler (автор — Половников Степан), совместимый с режимом IDEAL TASM и поддерживающий команды из наборов MMX, SSE, SSEII, SSEIII, 3DNow!Pro.

FASM

Писать о культовых проектах, не затронув чувства верующих и сохранив при этом здоровую долю скептицизма и объективизма не так-то просто, особенно если и сам являешься его поклонником. FASM[14] — это крайне необычный транслятор с экзотичными возможностями, которых все мы давно (и безуспешно!) ждали от крупных производителей, которые были слишком далеки

[13] TASM = Turbo Assembler.

[14] FASM — Ассемблер плоского режима (Flat Assembler).

от практического программирования и пытались формировать новые потребности (например, путем введения поддержки ООП), вместо того, чтобы удовлетворять те, что есть.

Так продолжалось до тех пор, пока Томаш Гриштар (Tomasz Grysztar) — аспирант Ягеллонского университета в Кракове — не задумал написать свою собственную ОС, названную Титаном и представляющую некоторое подобие DOS-системы для защищенного режима. Перебрав несколько ассемблерных трансляторов, но так и не обнаружив среди них подходящего, Томаш пошел на довольно амбициозный шаг, решив разработать необходимый инструментарий самостоятельно. Это произошло в 1999-03-23, 14:24:33 (дата создания первого файла), и уже к началу мая 1999 года появилась версия, способная транслировать сама себя (FASM написан на FASM). Операционная система в результате одной случайной катастрофы пала смертью храбрых, а вот исходные тексты FASM — остались, и с тех пор он продолжает активно развиваться.

Что же такое FASM? Это ассемблер с предельно упрощенным синтаксисом (никаких захламляющих листинг директив типа `offset`), полной поддержкой всех процессорных команд (в том числе и `jmp 0007:00000000`), качественным кодогенератором, мощным макропроцессором и гибкой системой управления форматом выходных файлов.

FASM распространяется в исходных текстах на бесплатной основе. К настоящему моменту он перенесен на MS-DOS, Windows 9x/NT, Linux, BSD. FASM поддерживает Unicode и все процессоры линейки x86 вплоть до Pentium 4 с наборами мультимедийных инструкций MMX, SSE, SSEII, SSEIII, AMD 3DNow!, а также платформу AMD x86-64. Это позволяет генерировать не только Microsoft COFF, но и готовые файлы форматов bin, mz[15], pe[16] и elf. То есть, фактически, FASM позволяет обходиться без компоновщика, однако при этом раскладку секций в PE-файле и таблицу импорта приходится создавать "вручную" с помощью специальных директив ассемблера. Выглядит это очень заманчиво, но на практике все же намного удобнее сгенерировать файл формата coff и скомпоновать его с модулями, написанными на языках высокого уровня.

Макроязык FASM настолько мощен, что позволяет писать программы на себе самом без единой ассемблерной строки. Пример такой программы приведен в листинге 4.5. И пускай кто-то ворчит, ну вот, мол, еще одна попытка опустить ассемблер до уровня Basic. Ничего подобного! Макросы — вещь добровольная. Хочешь — пользуйся, не хочешь — не надо.

Листинг 4.5. Программа, целиком написанная на интерпретируемом языке FASM

```
file 'interp.asm'
repeat $
  load A byte from %-1
  if A>='a' & A<='z'
    A = A-'a'+'A'
  end if
  store byte A at %-1
end repeat
```

Все это были достоинства. Теперь поговорим о недостатках. Ни на что не похожий синтаксис FASM напрягает даже матерых программистов, заставляя их вгрызаться в плохо структурированную документацию и небольшое количество демонстрационных примеров, поставляемых вместе с транслятором. На это требуется время, которое в конечном счете ничем не компенсируется, так как круг задач, на которых FASM реально превосходит MASM крайне узок. Категорическая несовместимость с MASM чрезвычайно затрудняет разработку драйверов Windows (в большинстве своем создаваемых на основе примеров из DDK). Прикладным задачам, в свою

[15] Исполняемые файлы MS-DOS, распознаваемые по присутствию ASCII-строки `MZ` (в шестнадцатеричном формате `4D 5A`) в начале файла.

[16] PE = Portable Executable. Более подробно о файлах PE будет рассказано в *главе 30, "Дизассемблирование 32-разрядных PE-файлов"* и *главе 32, "Архитектура x86-64 под скальпелем ассемблерщика"*.

очередь, требуется SDK (желательно новейшей версии), да и программы, целиком написанные на ассемблере, — это совсем не то, чего требует бизнес-машина. "Математические" задачи, перемножающие матрицы, вычисляющие координаты пересечения кривых в N-мерном пространстве или трансформирующие графику — легко пишутся на FASM, поскольку они не привязаны к конкретной операционной системе и никаких API-функций не вызывают.

Если бы FASM поддерживал генерацию отладочной информации, его (с некоторой натяжкой) еще было бы можно рассматривать как серьезный инструмент, а так… он остается игрушкой, пригодной для мелких задач типа "Hello, world", вирусов, демонстрационных программок и прочих произведений хакерского творчества.

Наконец, ни у кого нет гарантий, что создатель FASM не утратит к нему интереса, а ведь без поддержки новых процессорных инструкций всякий транслятор обречен на медленное, но неизбежное умирание. Открытость исходных текстов тут не поможет, помимо них нужна еще и *команда*. Нужны "носители знания", способные удержать детали проекта у себя в голове, а тот факт, что FASM написан на себе самом, увы, читаемости листингам отнюдь не добавляет.

NASM

Транслятор NASM[17] (рис. 4.2) вырос из идеи, поданной на comp.lang.asm.x86 (или возможно на alt.lang.asm — сейчас точно никто и не помнит), когда не было ни одного хорошего свободного ассемблера под x86. FASM тогда еще не существовал. MASM/TASM стоили денег и работали только под MS-DOS/Windows. Единственный более-менее работающий транслятор под UNIX — GAS (GNU Assembler) завязан на компилятор GCC и имеет синтаксис AT&T, сильно отличающийся от синтаксиса Intel, используемого остальными популярными трансляторами. При этом примеров программ, запрограммированных на GAS, практически не было[18]. Остальные ассемблеры (типа A86, AS86) не позволяют писать 16/32-разрядный код или раздаются практически без документации.

Рис. 4.2. Официальный логотип NASM

В итоге группа программистов во главе с Петером Анвином (Peter Anvin) решила разработать собственный ассемблер, и это у нее получилось! NASM обладает следующими отличительными чертами:

- MASM-подобный синтаксис;
- Достаточно мощная макросистема (впрочем, несовместимая с MASM и ничего не знающая о множестве полезных возможностей наподобие `union`);
- Поддержка всей линейки процессоров x86 вплоть до IA64 в x86-режиме;
- Богатство форматов выходных файлов (bin, aout, aoutb[19], coff, elf, as86, obj, win32, rdf, ieee);

[17] NASM — Ассемблер шириной во всю сеть, или просто расширенный ассемблер (Netwide Assembler).

[18] В последнее время ситуация начала меняться к лучшему. См., например, **http://asm.sourceforge.net/resources.html**.

[19] Файлы aoutb — это объектные файлы a.out в форме, предназначенной для различных клонов BSD Unix — NetBSD, FreeBSD и OpenBSD.

- Генерация отладочной информации в форматах Borland, STABS[20] и DWARF2[21];
- Наличие версий, портированных на MS-DOS, Windows, Linux и BSD.

Все это обеспечило NASM широкую популярность, однако без ярко выраженного фанатизма, характерного для поклонников FASM. Количество ошибок в трансляторе довольно велико, причем в отличие от MASM/TASM при "хитрых ошибках" NASM не падает, а генерирует ошибочный (по структуре) объектный файл. Выяснение того, как он его сгенерировал, сложно даже матерым хакерам. И, как это принято в сообществе Open Source — полное игнорирование багрепортов, "неудобных" для авторов (разработчики даже утверждают, что ошибок в их трансляторе вообще нет). Тем не менее, в последней версии NASM, в зависимости от значения ключа `-On`, код может сгенерироваться в двух или большем количестве экземпляров, или может пропасть весь экспорт (pubdef).

К минусам NASM можно отнести отсутствие поддержки Unicode, платформы AMD x86-64, формата отладочной информации CodeView, а также некоторые странности синтаксиса. В частности, команда `mov eax, 1` не оптимизируется, и транслятор умышленно оставляет место для 32-разрядного операнда. Если же мы хотим получить "короткий" вариант, размер операнда необходимо указывать явно: `mov eax, byte 1`, что очень сильно напрягает, или… использовать опцию `-On` для автоматической оптимизации.

Также необходимо принудительно указывать длину переходов `short` или `near`, иначе очень легко нарваться на ругательство `short jump out of range`. Впрочем, опять-таки, существует возможность настроить транслятор на генерацию `near`-переходов по умолчанию.

Гораздо хуже то, что NASM не запоминает типы объявляемых переменных и не имеет нормальной поддержки структур (впрочем, само понятие "нормальности" структур в ассемблере весьма растяжимо, и каждый волен трактовать его по-своему).

Из мелких недочетов можно назвать невозможность автоматической генерации короткого варианта инструкции `push imm8` и отсутствие контроля соответствия транслируемых инструкций типу указанного процессора (команда `cpuid` под `.486` ассемблируется вполне нормально, а ведь не должна).

Непосредственная трансляция примеров из SDK/DDK под NASM невозможна, так что разрабатывать на нем драйверы Windows может только очень крутой его поклонник. NASM — один из лучших ассемблеров под Linux/BSD, а вот под Windows его позиции уже не так сильны (в основном из-за неполной совместимости с MASM).

YASM

Когда развитие NASM затормозилось, его исходные тексты легли в основу нового транслятора — YASM[22] (рис. 4.3).

---

[20] STABS — формат отладочной информации, изначально разработанный Петером Кесслером (Peter Kessler) в Университете Беркли для отладчика pdx, предназначенного для языка Pascal.

[21] DWARF — это формат отладочной информации, используемый множеством компиляторов и отладчиков для поддержки отладки приложений с исходными текстами. Само имя DWARF может быть расшифровано как "Debugging With Attributed Record Formats" (отладка с приписываемыми форматами записей), хотя нигде в официальной документации это прямо не упоминается. Можно сделать и другое предположение — что разработчики были большими любителями Толкиена (особенно если учесть, что параллельно велась работа и над форматом ELF). Подробную информацию о формате DWARF и версиях стандарта можно найти здесь: **http://dwarfstd.org/**.

[22] В зависимости от настроения аббревиатура YASM может расшифровываться и как "Yes, it's an assembler", и как "Your favorite assembler", и как "Yet another assembler", и даже как "Why an assembler" (последнее — шутка).

Рис. 4.3. Официальный логотип YASM

Вот основные черты, отличающие YASM от его предшественника:

- Поддержка платформы AMD x86-64;
- Большое количество исправленных ошибок (которых в NASM якобы "нет");
- Оптимизированный синтаксический анализатор (parser), интерпретирующий синтаксис как NASM, так и GAS;
- Более полная поддержка выходных файлов COFF (DJGPP[23]) и Win32 obj;
- Генерация отладочной информации в формате CodeView;
- Интернационализация (выполненная через GNU-библиотеку `gettext`).

Есть и другие мелкие улучшения, которых вполне достаточно, чтобы потеснить NASM, особенно в мире UNIX-подобных систем, где синтаксис GAS по-прежнему играет ведущую роль.

Под Windows же YASM не имеет никаких ощутимых преимуществ перед MASM, за исключением того, что поддерживает возможность генерации двоичных файлов, особенно удобных для создания shell-кода, но бесполезных для разработчика драйверов.

Программирование на ассемблере для UNIX и Linux

Все предыдущее обсуждение ассемблерных трансляторов, в основном, было ориентировано на Windows-программистов. UNIX-подобные системы упоминались лишь вкратце. Настало время восполнить этот пробел. Надо отметить, что мнение о том, что "никто не пишет на ассемблере для UNIX/Linux", несмотря на широкую распространенность, все же не совсем соответствует действительности, или, что точнее, оно уже устарело.

У многих читателей может возникнуть вопрос — а зачем вообще программировать на ассемблере под UNIX? Что полезного можно создать таким образом, кроме "безделушек" и игрушек для сумасшедшего хакера? Что же, на этот вопрос есть весьма и весьма аргументированный ответ. Вот скажите: а вас интересует разработка "заплаток" для BIOS или даже разработка собственных модулей BIOS специального назначения? Настоящего хакера эта тема просто не может не заинтересовать. А знаете ли вы, что как раз именно набор компиляторов проекта GNU (GNU Compiler Collection, GCC) и предоставляет множество интересных возможностей для разработки BIOS и связанного с ней программного обеспечения[24]? Так что, если вы заинтересованы в этой теме, продолжайте читать! В данном разделе будет приведен необходимый минимум информации, необходимый для того, чтобы начать программировать на ассемблере под UNIX.

Приведенный здесь материал призван развеять миф о том, что программирование на ассемблере под UNIX — это кошмар, что оно устарело, что все, кто этим занимается, лишь впустую тратят свое время. Вы увидите, что писать программы на ассемблере под UNIX ничуть не сложнее, чем делать то же самое под Windows, и что руганый-переруганый GAS даже имеет некоторые преимущества.

---

[23] DJGPP — это акроним от DJ's GNU Programming Platform (платформа программирования GNU от DJ). Это проект по переносу большинства современных утилит разработки GNU на платформы DOS и Windows. Название происходит от имени автора *DJ Delorie*. Более подробную информацию можно найти по адресам: **http://www.delorie.com/djgpp/** и **http://my.execpc.com/~geezer/osd/exec/**.

[24] Более подробно об этом рассказано в замечательной книге известного хакера Pinckzakko (Д. Салихан. "BIOS: дизассемблирование, модификация, программирование". — СПб.: БХВ-Петербург, 2007).

Разумеется, у каждого, кто приступает к освоению новой для себя темы (это относится не только к программированию на ассемблере), сразу же возникает множество вопросов. Постараемся ответить на наиболее распространенные из них по пунктам.

Выбор транслятора

Как и в мире Windows, первый вопрос, который встает перед UNIX-программистом, решившим заняться написанием программ на ассемблере, — это выбор транслятора.

Стандартный ассемблер для UNIX — это `as` (GAS), входящий в состав бесплатно распространяемого комплекта Binutils (**ftp://ftp.gnu.org/gnu/binutils**). GAS имеется в составе практически любого дистрибутива UNIX/Linux. Он поддерживает огромное количество процессоров (включая Intel Pentium 4 SSE3 и AMD x86-64). В качестве макропроцессора использует штатный препроцессор C. Выходной формат: a.out. GAS поддерживает синтаксис языка ассемблера AT&T.

В последнюю версию GAS, входящую в состав Binutils 2.17, в число его функциональных возможностей была добавлена и поддержка синтаксиса языка ассемблера Intel (см. **http://sourceware.org/cgi-bin/cvsweb.cgi/~checkout~/src/gas/NEWS?rev=1.82&content-type=text/plain&cvsroot=src&only_with_tag=binutils-2_17**).

Кроме GAS, для программирования на ассемблере под UNIX можно использовать и другие ассемблеры, рассмотренные ранее в этой главе: NASM, YASM или FASM. Следует заметить, что для разработки программ на "чистом" ассемблере (например, простых "заплаток" BIOS), использование этих трансляторов вполне оправданно. Причем если вы раньше уже программировали под MS-DOS/Windows, то, вероятно, их использование будет наилучшим выбором. Однако для разработки более сложного системного программного обеспечения одного только языка ассемблера уже недостаточно, и в этом случае лучше применять метод ассемблерных вставок, обсуждавшийся ранее в этой главе. И для этой цели гораздо лучше подходит GCC (GNU Compiler Collection), в комплект поставки которого входит и GAS.

GAS пользуется наибольшей популярностью, и знать его синтаксис необходимо уже хотя бы затем, чтобы разбираться с чужими программами. Обсуждением этого синтаксиса мы сейчас и займемся.

Синтаксис Intel и синтаксис AT&T

Синтаксис AT&T разрабатывался компанией AT&T в те далекие времена, когда никакой корпорации Intel вообще не существовало, процессоры менялись как перчатки, и знание нескольких ассемблеров было вполне нормальным явлением. По сравнению с синтаксисом Intel, AT&T-синтаксис намного более избыточен. Однако это сделано умышленно с целью сокращения ошибок (пример: на одном процессоре команда `MOV` может перемещать 32-бита, на другом 16, а на третьем — вообще 64).

Отличия синтаксиса AT&T от Intel следующие:

- Имена регистров предваряются префиксом `%`:

```
Intel:   eax,    ebx,    dl
AT&T:    %eax,   %ebx,   %dl
```

- Обратный порядок операндов: вначале источник, затем приёмник:

```
Intel:   mov    eax, ebx
AT&T:    movl   %ebx, %eax
```

- Размер операнда задается суффиксом, замыкающим инструкцию, всего есть три типа суффиксов: `b` — байт (8 бит), `w` — слово (16 бит) и `l` — двойное слово (32 бита):

```
Intel:   mov    ah, al
AT&T:    movb   %al, %ah
Intel:   mov    bx, ax
```

```
AT&T:    movw    %ax, %bx
Intel:   mov     eax, ebx
AT&T:    movl    %ebx, %eax
```

- В командах длинного косвенного перехода или вызова (indirect far jump/call), а также дальнего возврата из функции (`ret far`) префикс размера (`l`) ставится перед командой (сокращение от long jmp/call) независимо от физического размера операнда, равного 32-битам в 16-разрядном режиме и 48-битам — в 32-разрядном:

```
Intel:   jmp     large fword ptr ds:[666h]
AT&T:    ljmp    *0x666
Intel:   retf
AT&T:    lret
```

- Числовые константы записываются в соответствии с соглашением C:

```
Intel:   69h
AT&T:    0x69
```

- Для получения смещения метки используется префикс `$`, отсутствие которого приводит к чтению содержимого ячейки:

```
Intel:   mov     eax, offset label
AT&T:    movl    $label, %eax
Intel:   mov     eax, [label]
AT&T:    movl    label, %eax
```

- В тех случаях, когда метка является адресом перехода, префикс `$` опускается:

```
Intel:   jmp     label
AT&T:    jmp     label
Intel:   jmp     -
AT&T     jmp     0x69
```

- Для косвенного перехода по адресу используется префикс `*`:

```
Intel:   jmp     dword ptr ds:[69h]
AT&T:    jmp     *0x69
Intel:   jmp     dword ptr ds:[label]
AT&T:    jmp     *label
Intel:   jmp     eax
AT&T:    jmp     *%eax
Intel:   jmp     dword ptr ds: [eax]
AT&T:    jmp     *(%eax)
```

- Использование префикса `$` перед константой применяется для получения ее значения. Знак (если он есть) ставится после префикса. Константа без указателя трактуется как указатель:

```
Intel:   mov     eax, 69h
AT&T     movl    $0x69, %eax
Intel:   mov     eax, -69h
AT&T     movl    $-0x69, %eax
Intel:   mov     eax, [69h]
AT&T     movl    0x69, %eax
```

- Для реализации косвенной адресации базовый регистр заключается в круглые скобки, перед которыми может присутствовать индекс, записанный в виде числовой константы или метки без префикса `$`:

```
Intel:    mov    eax, [ebx]
AT&T:     movl   (%ebx), %eax
Intel:    mov    eax, [ebx+69h]
AT&T:     movl   0x69(%ebx), %eax
Intel:    mov    eax, [ebx+label]
AT&T:     movl   label(%ebx), %eax
```

- Если регистров несколько, то они разделяются запятыми:

```
Intel:    mov    eax, [ebx+ecx];
AT&T:     movl   (%ebx, %ecx), %eax;
```

- Для задания коэффициента масштабирования (scale) перед первым регистром ставится ведущая запятая (при использовании базовой индексной адресации запятая опускается), а сам коэффициент отделяется другой запятой, без префикса `$`:

```
Intel:    mov    eax, [ebx*8]
AT&T      movl   (,%ebx, 8), %eax
Intel:    mov    eax, [ebx*8+label]
AT&T      movl   label(,%ebx, 8), %eax
Intel:    mov    eax, [ecx+ebx*8+label]
AT&T:     movl   label(%ecx, %ebx, 8)
Intel:    mov    eax, [ebx+ecx*8+label]
AT&T:     movl   label(%ebx, %ecx, 8)
```

- Сегментная адресация с использованием сегментных регистров отличается от применяемой в синтаксисе Intel использованием круглых скобок вместо квадратных:

```
Intel:    mov    eax, es:[ebx]
AT&T:     movl   %es:(%bx), %eax
```

- В командах переходов и вызовов функций непосредственные сегмент и смещение разделяются не двоеточием, а запятой:

```
Intel:    jmp far 10h:100000h (псевдоконструкция!)
AT&T:     jmp $0x10, $0x100000
Intel:    jmp far ptr 10:100000
AT&T:     jmp $10, $0100000 — транслируется в –> jmp far ptr 0:0F4240h
```

UNIX API

Как и в Windows, в UNIX имеется и API — высокоуровневые библиотеки и, в первую очередь, LIBC (условный аналог KERNEL32.DLL в Win32), пример использования которой был продемонстрирован в листинге 4.6.

Некоторые хакеры тяготеют к использованию системных вызовов (syscalls), представляющих собой своеобразный Native API, по-разному реализованный в различных системах. Это затрудняет написание программ, поддерживающих несколько различных аппаратных архитектур. Тем не менее, применение системных вызовов оправдано при написании внедряемого кода (shell-код), а также в червях и вирусах — в силу простоты и компактности их вызова.

Простейшая ассемблерная программа для UNIX/Linux

Простейшая ассемблерная программа, работающая через штатную библиотеку LIBC и выводящая "Hello, world!" на консоль, представлена в листинге 4.6.

Листинг 4.6. Исходный текст программы demo-asm-libc.S, работающей через штатную библиотеку LIBC

```
.text

// Объявляем глобальную метку main
.global main

main:
                pushl           $len                    // Длина строки
                pushl           $msg                    // Указатель на строку
                pushl           $1                      // stdout
                call            write                   // Функция записи
                addl            $12,%esp                // Выталкиваем аргументы из стека

                ret                                     // Возвращаемся в стартовый код

.data
                msg: .ascii "hello, world!\n"           // Строка для вывода
                len = . - msg                           // Вычисление длины строки
```

Как же оттранслировать эту программу? Проще и правильнее всего делать это с помощью компилятора GCC. Однако в этом случае файлу следует присвоить расширение .S, иначе компилятор не поймет, что это ассемблерная программа (листинг 4.7).

Листинг 4.7. Сборка ассемблерной программы при помощи gcc

```
# Компилируем и компонуем
$gcc demo-asm-libc.S -o demo-asm-libc

# Удаляем символьную информацию (для сокращения размеров файла)
$strip demo-asm-libc

# Запускаем на выполнение
$./demo-asm-libc
hello, world!
```

Рассмотрим реализацию той же программы (выводящей на консоль строку `hello, world!`), но работающей не через штатную библиотеку LIBC, а через системные вызовы (листинг 4.8).

Листинг 4.8. Исходный текст программы demo-asm-80h.S, работающей через системные вызовы

```
.text

// Точка входа, которую ищет компоновщик по умолчанию
.globl          _start

_start:
                movl            $4,%eax         // Системный вызов #4 "write"
                movl            $1,%ebx         // 1 -- stdout
                movl            $msg,%ecx       // Смещение выводимой строки
                movl            $len,%edx       // Длина строки
```

```
                int             $0x80           // write(1, msg, len);

                movl            $1, %eax        // Системный вызов #1 "exit"
                xorl            %ebx,%ebx       // Код возврата
                int             $0x80           // exit(0);

.data
                msg: .ascii "hello, world\n"
                len = . - msg
```

Пример, иллюстрирующий трансляцию и сборку этой программы, приведен в листинге 4.9.

Листинг 4.9. Трансляция и сборка программы, приведенной в листинге 4.8 (без помощи gcc)

```
# Транслируем
$as -o demo-asm-80h.o demo-asm-80h.S

# Компонуем
$ld -s -o demo-asm-80h demo-asm-80h.o

# Удаляем символьную информацию (для сокращения размеров файла)
$strip demo-asm-80h

# Запускаем на linux
./demo-asm-80h
hello, world!

# Запускам на xBSD (через эмулятор системных вызовов)
brandelf -t Linux demo-asm-80h
./demo-asm-80h
hello, world!
```

Заключение

Попробуем подвести итог, обобщив все вышесказанное в нескольких словах (по одному слову для каждого транслятора):

- MASM (Macro Assembler) — стандарт де-факто при программировании под Windows 9x и линейку Windows NT.
- TASM (Turbo Assembler) — мертвый ассемблер, пригодный только для MS-DOS.
- Lazy Assembler — реинкарнация TASM с поддержкой новых команд процессора.
- FASM (Flat Assembler) — неординарный и весьма самобытный, но увы, игрушечный ассемблер.
- NASM (Netwide Assembler) — хороший ассемблер под Linux/BSD с Intel-синтаксисом.
- YASM (Yet another assembler) — усовершенствованный вариант NASM.
- HLA (High Level Assembly Language) — очень высокоуровневый ассемблер, на любителя.
- GAS (GNU Assembler) — стандартный ассемблер для UNIX и Linux, входящий в состав наборов GNU Binutils и GCC.

Сравнительный анализ характеристик рассмотренных ассемблеров приведен в табл. 4.1.

Таблица 4.1. *Сравнительный анализ характеристик различных ассемблеров*

Критерий	MASM	TASM	FASM	NASM	YASM	GAS
Цена	Бесплатный	—	Бесплатный	Бесплатный	Бесплатный	Бесплатный
Открытость	Закрытый	Закрытый	Открытый	Открытый	Открытый	Открытый
Владелец	Microsoft	Borland	Tomasz Grysz-tar	Сообщество	Сообщество	Сообщество
Популярность	Огромная	Низкая	Высокая	Умеренная	Умеренная	Умеренная
Совместимость с MASM	:-)	Хорошая	—	Низкая	Низкая	—
Архитектуры	x86 16/32, x86-64	x86 16/32	x86 16/32, x86-64	x86 16/32	x86 16/32, x86-64	IA32, IA64, Alpha, PPC, SPARC, PDP-11
Поддержка расширений SSE/SSEII/ SSEIII	поддерживает	не поддержи-вает	поддерживает	поддерживает	поддерживает	поддерживает
Платформы	DOS, WIN	DOS, WIN	DOS, WIN, Linux, BSD	DOS, WIN, Linux, BSD	DOS, WIN, Linux, BSD	DOS, WIN, Linux, BSD BeOS, Unix-Ware, Solaris
Отладочная информация	CodeView, PDB	Borland	—	Borland, STABS, DWARF2	Borland, Code-View, STABS, DWARF2	STABS, ECOFF, DWARF2
Выходные файлы	COFF, MS OMF	MS OMF, IBM OMF, PharLap	BIN, MZ, PE, COFF, ELF	BIN, Aout, Aoutb, COFF, ELF, as86, OBJ, Win32, RDF, IEEE	BIN, COFF, ELF	Aout, COFF, ECOFF, XCOFF, ELF, BIN
Поддержка Unicode	поддерживает	не поддержи-вает	поддерживает	не поддержи-вает	не поддержи-вает	Не поддержи-вает
Документация	Отличная	Отличная	Приемлемая	Хорошая	Хорошая	Хорошая
Количество багов	Огромное	Умеренное	Низкое	Высокое	Умеренное	Умеренное
Комбини-руется с	DDK, VC, IDA	Borland C++	—	—	—	GCC

Ссылки на упомянутые продукты

- Kernel-Mode Driver Framework — среда разработки драйверов, бесплатно распространяемая Microsoft, в состав которой входит транслятор MASM со всеми необходимыми утилитами: **http://www.microsoft.com/whdc/driver/wdf/KMDF_pkg.mspx**.
- Visual Studio Express Edition — урезанная редакция Visual Studio, распространяемая на бесплатной основе с транслятором MASM и прочими необходимыми утилитами: **http://msdn.microsoft.com/vstudio/express**.
- Пакет Хатча — последние версии MASM плюс набор дополнительных утилит, документации и всего-всего, что может понадобиться при программировании под Windows: **http://www.movsd.com/** и **http://www.masm32.com/**.
- MASM docs — официальная документация по MASM на MSDN (на английском языке): **http://msdn2.microsoft.com/en-us/library/ms300951.aspx**.
- TASM 5+ by Borland Corp. and !tE — неофициальный пакет от хакера !tE, содержащий последние версии транслятора TASM со всеми доступными патчами, документаций и прочими утилитами: **http://www.wasm.ru/baixado.php?mode=tool&id=230**.

- FASM home — домашняя страница FASM, откуда можно скачать сам транслятор, документацию на английском языке и сопроводительные примеры к нему: **http://flatassembler.net**.
- Выбор — FASM: небольшая подборка статей, посвященная FASM (на русском языке): **http://www.softoplanet.ru/lofiversion/index.php/t4404.html**.
- NASM home — домашняя страница NASM, откуда можно скачать сам транслятор, документацию на английском языке и сопроводительные примеры к нему: **https://sourceforge.net/projects/nasm**.
- Расширенный ассемблер — NASM — неофициальный перевод руководства по NASM (на русском языке): **http://www.codenet.ru/progr/asm/nasm**.
- YASM home — домашняя страница YASM, откуда можно скачать сам транслятор, документацию на английском языке и сопроводительные примеры к нему: **http://www.tortall.net/projects/yasm**.
- LAZY ASSEMBLER home — домашняя страница LAZY ASM, откуда можно скачать сам транслятор, документацию на английском языке и сопроводительные примеры к нему: **http://lzasm.hotbox.ru/**.
- HLA — "академический" проект очень высокоуровневого ассемблера с транслятором, документацией на английском языке и огромным количеством готовых примеров: **http://webster.cs.ucr.edu**.
- GNU Binutils — набор утилит для компиляции и построения программ, написанных на ассемблере GAS, главными среди которых являются as (GNU Assembler) и ld (GNU linker): **http://www.gnu.org/software/binutils/**.
- GCC (GNU Compiler Collection) — набор компиляторов GNU, с помощью которого также можно компилировать программы, написанные на ассемблере под UNIX и Linux: **http://gcc.gnu.org/**.
- LinuxAssembly.org — сайт, целиком посвященный программированию на ассемблере для UNIX и Linux. Помимо документации и обучающих материалов, здесь можно найти и примеры программирования на языке ассемблера под UNIX и Linux: **http://asm.sourceforge.net/**.
- Using as, the GNU Assembler — подробное руководство по использованию GAS (на английском языке): **http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/pdf/rhel-as-en.pdf**.

ЧАСТЬ II

БАЗОВЫЕ ТЕХНИКИ ХАКЕРСТВА

Глава 5

Введение в защитные механизмы

Всемогущи ли хакеры? Можно ли взломать любую защиту? Основная цель, с которой создаются механизмы защиты ПО — предотвращение пиратского копирования и несанкционированного использования программ. В основе большинства защитных механизмов лежит проверка *подлинности*. В любом случае разработчики программного обеспечения должны убедиться в том, что человек, запустивший программу, действительно тот, за кого себя выдает, и этот человек действительно является легальным пользователем. Иными словами, защита требует от пользователей предъявить доказательство того, что они честно купили ПО (или что на компьютере установлена лицензионная копия программы). Таким образом, защитные механизмы подразделяются на две основные категории (рис. 5.1):

- Защитные механизмы, основанные на *знании* (пароля, серийного номера и т. д.).
- Защитные механизмы, основанные на *владении* (ключевым диском, документацией и т. д.).

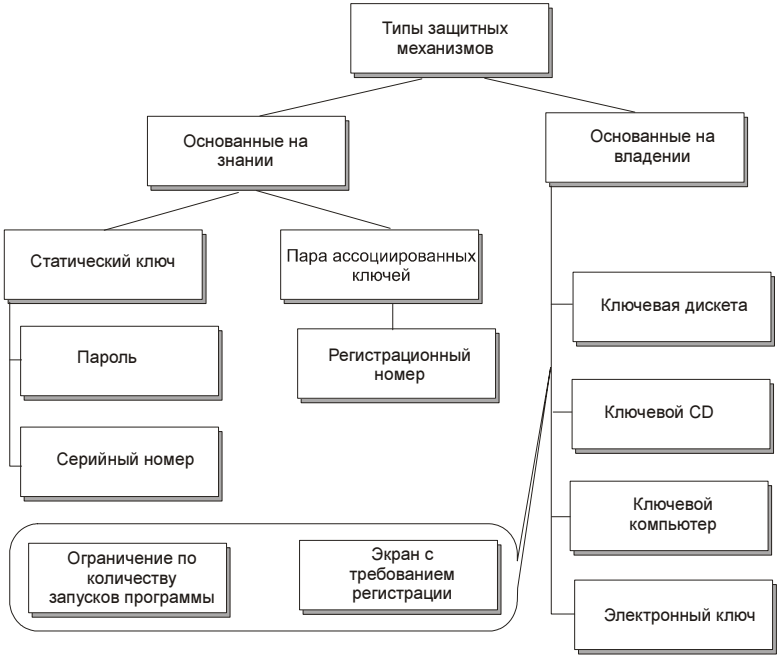

Рис. 5.1. Основные типы защитных механизмов

Защитные механизмы, основанные на знании, становятся бесполезными, если легальные пользователи не заинтересованы в том, чтобы хранить это знание в секрете. Владельцы могут сообщить пароли и/или серийные номера кому угодно, и этот кто угодно сможет запустить программу на своем компьютере. Хотя стоит отметить, что серийные номера все-таки полезны, так как незарегистрированные пользователи не могут получать техническую поддержку. Это вполне может подтолкнуть их к покупке лицензионных программ.

Более надежной является защита, основанная на владении некоторым носителем, который сложно (в идеале — невозможно) воспроизвести. Впервые защита этого типа появилась в форме ключевых дискет. Информация записывалась на ключевые дискеты таким образом, чтобы сделать их копирование невозможным. Простейшим (но далеко не лучшим) вариантом было нанесение на дискету небольших повреждений (например, шилом или перочинным ножом). Затем, определив сектор, в котором располагается дефект (путем чтения и записи тестовой информации вплоть до того момента, когда операция чтения начнет выдавать бессмысленный мусор), создатель защиты "привязывал" к нему программу. Каждый раз, когда программа запускалась, ее защитный механизм проверял, находится ли дефект на прежнем месте. Когда дискеты утратили популярность, тот же подход начал использоваться применительно к компакт-дискам. Наиболее продвинутые разработчики защит наносили повреждения лазером, а большинство простых смертных по-прежнему пользовались гвоздями и перочинными ножами.

В случае применения этой защиты, программа жестко привязывалась к диску и отказывалась работать в случае его отсутствия. Поскольку копирование такого диска невозможно (вследствие невозможности создания на копии идентичных дефектов), защитный механизм предотвращает пиратское распространение программы.

Другие защитные механизмы часто ограничивают количество запусков программы или продолжительность ее использования. Они часто применяются в инсталляторах. При этом ключ защищенной программы (ключевой диск, регистрационный номер и т. п.) запрашивается только один раз, при ее установке, а дальнейшая работа возможна и без него. Однако, если количество установок ограничено, легальные пользователи могут понести ущерб от несанкционированной установки программы не несколько компьютеров. Это нарушает права легальных пользователей, ведь некоторые из них часто переустанавливают операционные системы и приложения. Наконец, ключевые диски распознаются не всеми типами приводов и часто недоступны через сеть. Если защитный механизм получает доступ к оборудованию непосредственно, то такая программа определенно не будет работать под Windows NT/2000/XP и, вероятно, под Windows 9*x*. Разумеется, это произойдет только в тех случаях, когда защитный механизм разрабатывался без учета особенностей этих ОС. Однако если это так, то ситуация еще хуже, потому что защитный механизм, исполняющийся на высшем уровне привилегий, может нанести системе непоправимый вред. Кроме того, ключевой носитель может быть потерян, украден или просто испорчен. Например, дискеты имеют тенденцию к размагничиванию и появлению плохих кластеров, компакт-диски могут быть поцарапаны, а электронные ключи могут "выгорать".

Если защита вызывает неудобства, пользователи предпочтут пользоваться пиратским ПО, несмотря на все разговоры о морали и этике.

В настоящее время наиболее популярны защитные механизмы, основанные на регистрационных номерах. При первом запуске программы защитный механизм "привязывается" к компьютеру, запускает счетчик, а в некоторых случаях — блокирует ряд функциональных возможностей защищаемой программы. Чтобы работать с полнофункциональной версией программы, пользователь должен получить от разработчика пароль (разумеется, не бесплатно). Чтобы предотвратить пиратское копирование, часто используются пароли, являющиеся производными от ключевых параметров компьютера конечного пользователя (в простейшем случае — от пользовательского имени и/или других данных).

При всем своем многообразии защитные механизмы, окружающие нас, делятся на два типа: *криптозащиты*, называемые также защитами Кирхгофа (Kirchhoff), и *логические защиты*.

Согласно правилу Кирхгофа, стойкость криптозащит определяется исключительно стойкостью секретного ключа (secret key). Даже если алгоритм работы такой защиты становится известен, это не намного упрощает его взлом. При условии правильного выбора длины ключа, защиты Кирхгофа невозможно взломать в принципе (если, конечно, нет грубых ошибок в их реализации, но криптозащиты с подобными ошибками в категорию защит Кирхгофа просто не попадают).

Стойкость логических защит, напротив, определяется степенью секретности защитного *алгоритма*, но отнюдь не ключа, вследствие чего надежность защиты базируется на одном лишь предположении, что защитный код программы не может быть изучен и/или изменен.

Конечно, рядовым пользователям, не умеющим работать ни с дизассемблерами, ни с отладчиками, безразлично, каким путем осуществляется проверка вводимого ими регистрационного номера. С их точки зрения, защищенное приложение представляет собой "черный ящик", на вход которого подается некоторая ключевая информация, а на выход — сообщение об успехе или неудаче. Хакеры — другое дело. Если регистрационный номер используется для расшифровки критически важных модулей программы, то дела обстоят плохо. Если процедура шифрования реализована без ошибок, единственное, что остается — найти рабочую (то есть легально зарегистрированную) программу и снять с нее дамп. Если же защита тем или иным путем сравнивает введенный пользователем пароль с заложенным в нее эталонным паролем, — у хакера есть все шансы ее сломать. Как? Исследуя защитный код, хакер может:

- Найти эталонный пароль и "подсунуть" его взламываемой программе.
- Заставить защиту сравнивать веденный пароль не с эталоном, а… с самим собой.
- Выяснить, какой именно условный переход выполняется при вводе неверного пароля и скорректировать его так, чтобы он передавал управление на "легальную" ветку программы, а не на ветку, выводящую сообщение с требованием регистрации.

Подробный разговор о конкретной технике взлома ждет нас впереди, пока же просто учтем, что такой тип защит действительно может быть взломан. Причем не просто взломан, а взломан очень быстро. Порой расправа с защитой занимает всего лишь несколько минут, и только очень мощным защитам удается продержаться под осадой день или два.

Возникает вопрос: если логические защиты и вправду настолько слабы, то почему же их так широко используют? Во-первых, большинство разработчиков программного обеспечения совершенно не разбираются в защитах. Они просто не представляют себе, во что именно компилятор "перемалывает" исходный код (судя по всему, машинный код им представляется таким дремучим лесом, из которого живым никто выбраться не сможет). Во-вторых, в ПО массового назначения надежность защитных механизмов все равно ничего не решает. Как было сказано ранее, при наличии хотя бы одной-единственной зарегистрированной копии, хакер просто "снимет" с программы дамп, вот и все! Тем не менее, несмотря на то, что все программы принципиально могут быть взломаны, "хакнуть" демонстрационную программу, загруженную из Интернета или купленную на CD, возможно далеко не всегда. Если критические участки приложения зашифрованы (или, что еще хуже, физически удалены из демонстрационного пакета), то взломать эту программу смогут немногие.

Классификация защит по роду секретного ключа

Одни защиты требуют ввода серийного номера, другие — установки ключевого диска, третьи же "привязываются" к конкретному компьютеру и наотрез отказываются работать на любом другом. Казалось бы — что может быть между ними общего? А вот что: для проверки легальности пользователя во всех случаях используется та или иная секретная информация, известная (и/или доступная) только этому пользователю, и лишь ему одному. В первом случае, в роли пароля выступает непосредственно сам серийный номер, во втором — информация, содержащаяся на ключевом диске, ну а в третьем — индивидуальные характеристики компьютера, которые воспринимаются защитным механизмом как последовательность чисел, и интерпретируются им точно так же, как и "настоящий" секретный пароль.

Правда, между секретным паролем и ключевым диском (ключевым компьютером) есть принципиальная разница. Пароль, вводимый вручную, пользователь *знает явно* и, при желании может поделиться им с друзьями без ущерба для себя. Ключевым диском (компьютером) пользователь *обладает*, но совершенно не представляет себе, что именно этот диск содержит и какие характеристики ключевого компьютера используются защитой. При условии, что ключевой диск не копируется автоматическими копировщиками, пользователь не сможет распространять защищенную программу до тех пор, пока не выяснит характер взаимодействия защиты с ключевым диском (компьютером) и не разберется, как эту защиту обойти. Это может быть сделано по меньшей мере тремя путями.

- Защитный механизм *нейтрализуется* (в особенности это относится к тем защитам, которые просто проверяют ключевой носитель на наличие неких уникальных характеристик, но реально никак их не используют).
- Ключевой носитель *дублируется* "один к одному" (весьма перспективный способ обхода защит, которые не только проверяют наличие ключевого носителя, но и некоторым сложным образом с ним взаимодействуют, скажем, динамически расшифровывают некоторые ветви программы, используя в качестве ключей номера сбойных секторов).
- Создается *эмулятор* ключевого носителя, обладающий всеми чертами оригинала, но реализованный на совершенно иных физических принципах. Этот подход актуален для тех случаев, когда скопировать ключевой носитель на имеющимся у хакера оборудовании невозможно или чрезвычайно затруднительно. Поэтому вместо того, чтобы послойно сканировать на электронном микроскопе всем хорошо известный HASP, хакер пишет специальную утилиту, которую защитный механизм воспринимает как настоящий HASP, но при этом данную утилиту можно свободно копировать.

Очевидно, что защиты, *основанные на знании*, полагаются исключительно на законодательство и законопослушность пользователей. Действительно, что помешает легальному пользователю поделиться паролем или сообщить серийный номер всем желающим? Конечно, подобное действие квалифицируется как "пиратство" и с недавнего времени преследуется по закону. Но точно так же преследуются (и наказываются!) все нелегальные распространители информации, охраняемой авторским правом, вне зависимости от наличия или отсутствия на ней защиты. Тем не менее, несмотря на резко ожесточившуюся борьбу с пиратами, нелегальное программное обеспечение по-прежнему свободно доступно. Практически под любую программу, распространяемую через Интернет как shareware, в том же самом Интернете можно найти готовый "крэк" (или бесплатный аналог требующейся утилиты).

В этих условиях "спасение утопающих — дело рук самих утопающих". Наивно, конечно, думать, что количество легальных продаж прямо пропорционально надежности вашей защиты, но… программа shareware без защиты рискует перестать продаваться вообще. Анализ программ, прилагаемых к журналу "Компьютер Пресс" на CD показал, что многие разработчики наконец-то вняли советам хакеров, и теперь программа требует для регистрации не пароль, а… неизвестно что. Это может быть и ключевой файл, и запись в реестре, и некоторая последовательность "вслепую" нажимаемых клавиш, и… еще много всего! Также исчезли текстовые сообщения об успехе или неудаче регистрации, в результате чего локализация защитного механизма в коде исследуемой программы значительно усложнилась (при наличии текстовых сообщений несложно по перекрестным ссылкам обнаружить код, который их выводит, после чего защитный механизм можно легко "раскрутить"). Из качественно новых отличий хотелось бы отметить использование Интернета для проверки лицензионной чистоты программы. В простейшем случае, защитный механизм периодически пытается соединиться через Интернет со специальным сервером, где хранится более или менее полная информация обо всех зарегистрированных клиентах. Если регистрационный номер, введенный пользователем, здесь действительно присутствует, то все ОК, в противном же случае защита дезактивирует флаг "зарегистрированности" программы, а то и удаляет защищаемую программу с диска. Естественно, разработчик программы может при желании удалять из базы регистрационные номера, которые кажутся подозрительными (например, растиражированные пиратами). Другие защиты скрытно устанавливают на компьютере сервер TCP/UDP,

предоставляющий разработчику защиты те или иные возможности удаленного управления программой (обычно дезактивацию ее нелегальной регистрации).

Тем не менее, такие защиты очень просто обнаружить и еще проще устранить. Обращение к Интернету не может пройти незаметным, — сам факт такого обращения легко распознается даже штатной утилитой `netstat`, входящий в комплект поставки операционных систем Windows 9*x* и линейки Windows NT. Эстеты могут воспользоваться бесплатной утилитой Марка Руссиновича TCPView (она доступна для скачивания по адресу **http://www.microsoft.com/technet/sysinternals/Networking/TcpView.mspx**). Локализовать код защитного механизма также не составит большого труда, — достаточно пойти по следу тех самых функций API, которые, собственно, и демаскируют защиту. Следует отметить, что все известные мне защиты этого типа пользовались исключительно библиотекой Winsock, и ни одна из них не отважилась взаимодействовать с сетевым драйвером напрямую (впрочем, это все равно не усложнило бы взлом).

Надежность защиты

Если защита основывается на предположении того, что ее код не может быть исследован и/или модифицирован, то это плохая защита. Закрытость исходного кода не является непреодолимым препятствием к исследованию и модификации приложения. Современные средства обратной разработки автоматически распознают библиотечные функции, локальные переменные, аргументы стека, типы данных, ветвления, циклы и т. д. По всей вероятности, вскоре дизассемблеры смогут и генерировать код, по виду подобный коду, написанному на языках высокого уровня.

Тем не менее, даже на сегодняшний день, анализ машинного кода не настолько сложен, чтобы долгое время противодействовать попыткам взлома. Лучшим доказательством этому служит постоянно растущее количество взломанных программ. В идеале, знание алгоритма работы защитного механизма не должно влиять на надежность защиты. Однако добиться этого возможно не всегда. Например, если демонстрационная версия сетевого приложения имеет ограничение на количество одновременных сетевых соединений (этот вариант ограничения встречается очень часто), то все, что требуется хакеру для взлома этой программы, — это найти инструкцию, которая выполняет данную проверку, и удалить ее. Модификация программы может быть предотвращена за счет регулярной проверки контрольных сумм. Однако и код, вычисляющий контрольную сумму и сравнивающий ее с эталоном, также может быть найден и нейтрализован.

Сколько бы уровней ни имела защита — один или миллион — защищенная программа все равно может быть взломана. Это — всего лишь вопрос времени и приложенных усилий. Существует распространенное мнение, в соответствии с которым никто не будет взламывать защиту, если стоимость легальной копии меньше затрат на взлом. Это далеко не всегда так! Материальная выгода — это не единственный мотив для хакера. Гораздо более мощными стимулами являются *интеллектуальная борьба с разработчиками защиты*, *дух соревнования между хакерами*, *любопытство*, *повышение квалификации*, и, наконец, взлом — это просто способ *интересно провести время*.

Таким образом, защитные механизмы просто обречены в их безнадежной борьбе за выживание (рис. 5.2). Наличие дополнительных уровней защиты может только замедлить взлом, по крайней мере в теории. Однако на практике создать защиту, которую невозможно взломать, по крайней мере, в течение жизненного цикла защищенной программы, все же возможно. Чтобы этого добиться, важно выбрать правильный подход. Например, бесполезно устанавливать сейфовый замок на картонную дверь. Линия защиты должна быть равномерно сильной во всех отношениях, так как общая надежность защитного механизма определяется его самым слабым компонентом. Итак, создать надежную защиту можно, и даже не столь уж сложно. Почему же тогда количество взломанных программ растет такими высокими темпами? Ответ предельно прост — из-за ошибок реализации, за которые разработчикам защиты некого винить, кроме себя.

Несмотря на разнообразие трюков и приемов, используемых создателями защит, большинство программ взламываются по одному и тому же набору стандартных шаблонов. Ошибки разработчиков удручающее однообразны — никакой тяги к творчеству. Никакого морального удовлетво-