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.

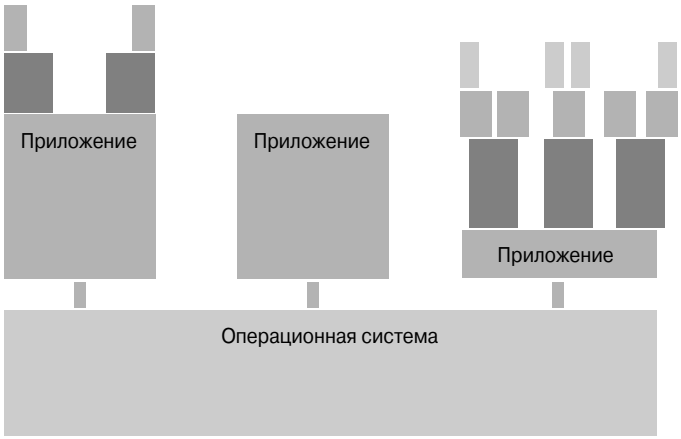

Рис. 2.1. Стандартное представление о работе приложений как отдельных иерархических структур. На самом деле эти приложения далеко не так четко "разложены по полочкам", как изображено на этом рисунке

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.

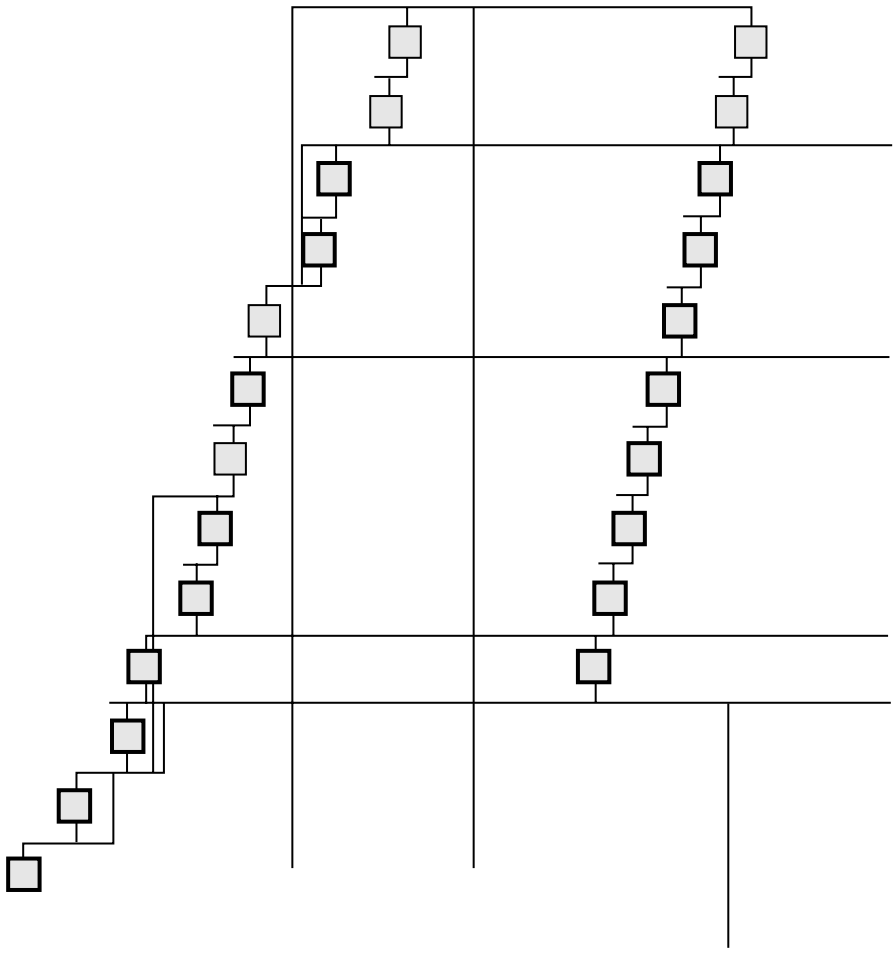

Рис. 2.2. На схеме изображена древовидная структура программного кода типичного FTP-сервера. Блоки представляют собой неразрывный код, а линии символизируют переходы и условные переходы между блоками кода. В выделенных жирными линиями блоках кода обрабатываются пользовательские данные

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.

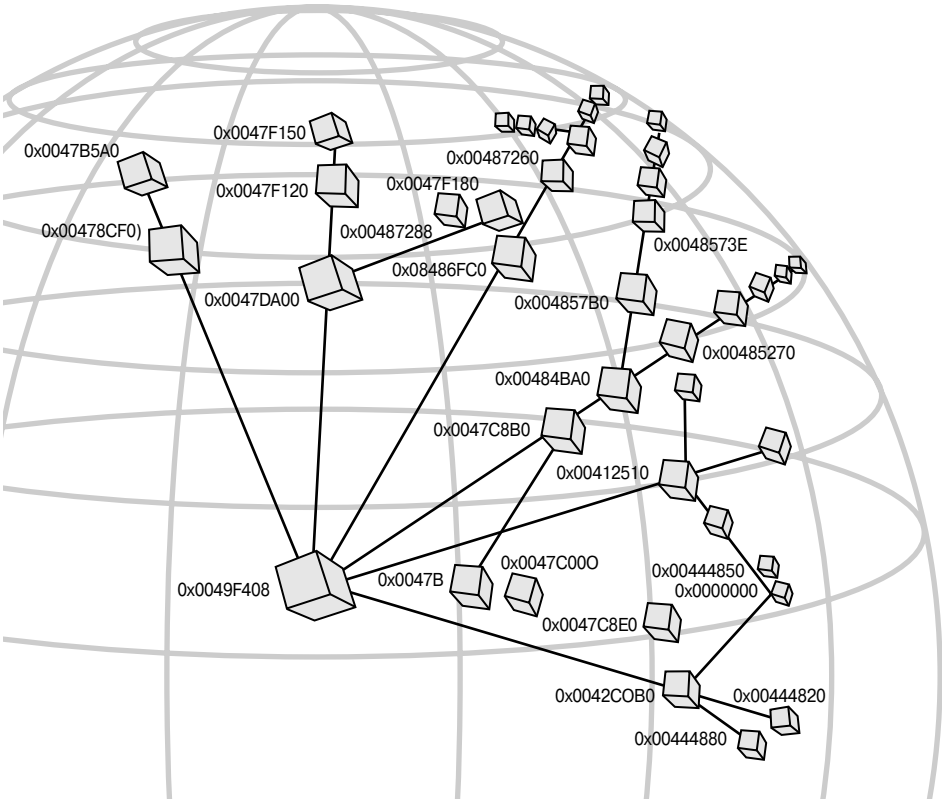

Рис. 2.3. Эта схема представлена в трех измерениях. Каждый фрагмент кода выглядит как отдельный куб. Мы использовали пакет OpenGL для иллюстрации всех путей в программном коде, ведущих к уязвимому вызову `sprintf` *в атакуемой программе*

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.

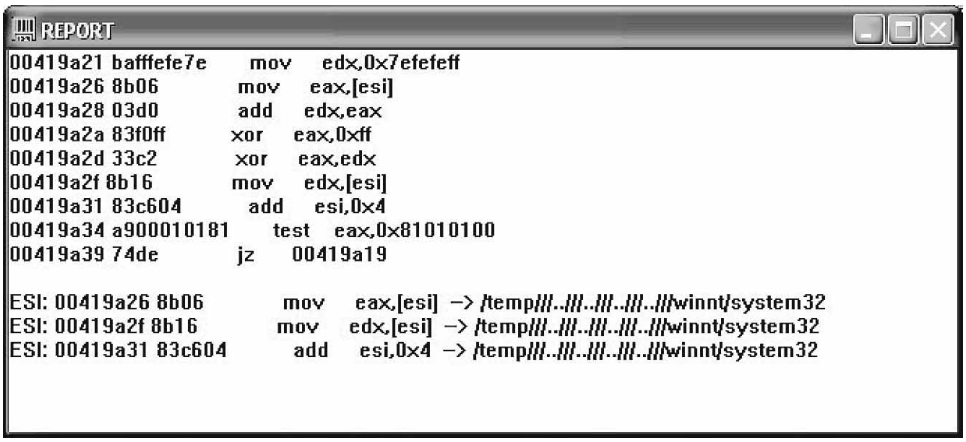

Рис. 2.4. Результат дизассемблирования одной "комнаты" в атакуемой программе. Первые строки листинга являются набором инструкций программы. Инструкции, которые обрабатывают введенные пользователями данные, показаны в конце листинга. При взломе программного обеспечения необходимо знать, как данные перемещаются 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.