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.

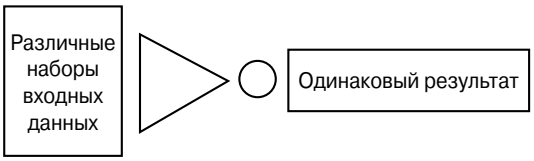
Это весьма интересный вид симбиоза. Пользователи систем сигнализации косвенно помогают с трудоустройством злоумышленникам, которые должны будут обновлять их системы сигнализации, которые в свою очередь предназначены для поимки этих злоумышленников.

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`.

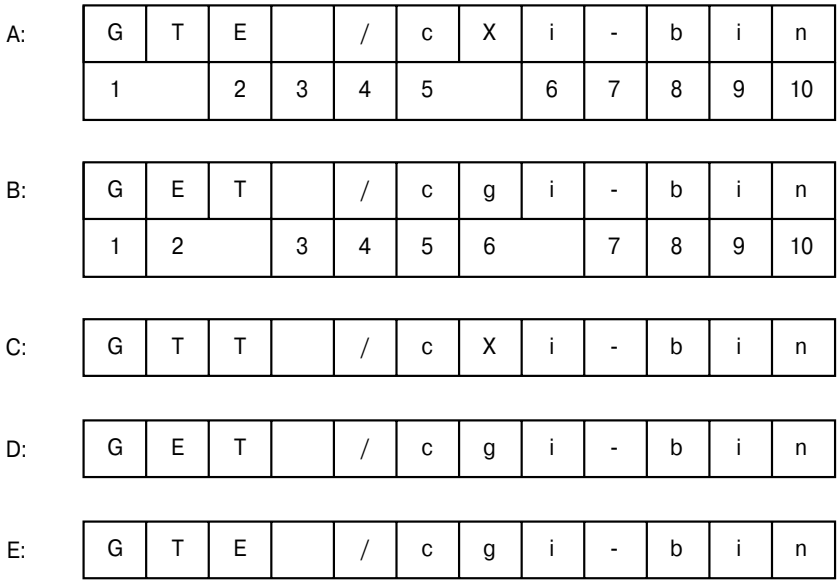


Рис. 6.1. Десинхронизация GET-запроса

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.

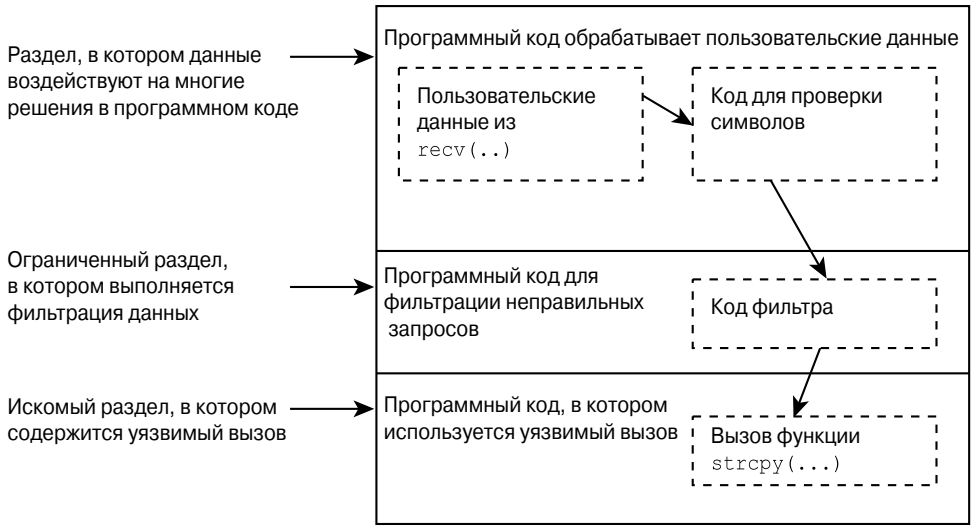


Рис. 6.2. Три раздела в программном коде атакуемой программы и их влияние на обратную трассировку

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.

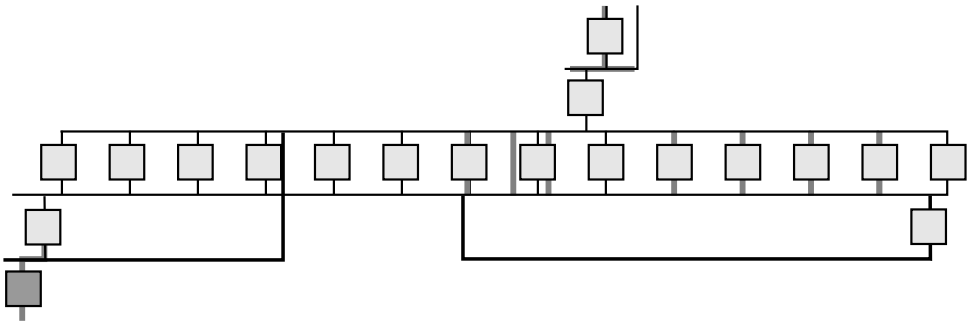


Рис. 6.3. Квадратик темно-серого цвета на рисунке — это область уязвимого кода в библиотеке `irc.dll` программы Trillian. Управление передается через большой по размеру оператор `switch` "по дороге" к интересующей области кода. Для составления этого рисунка мы использовали программу IDA-Pro

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

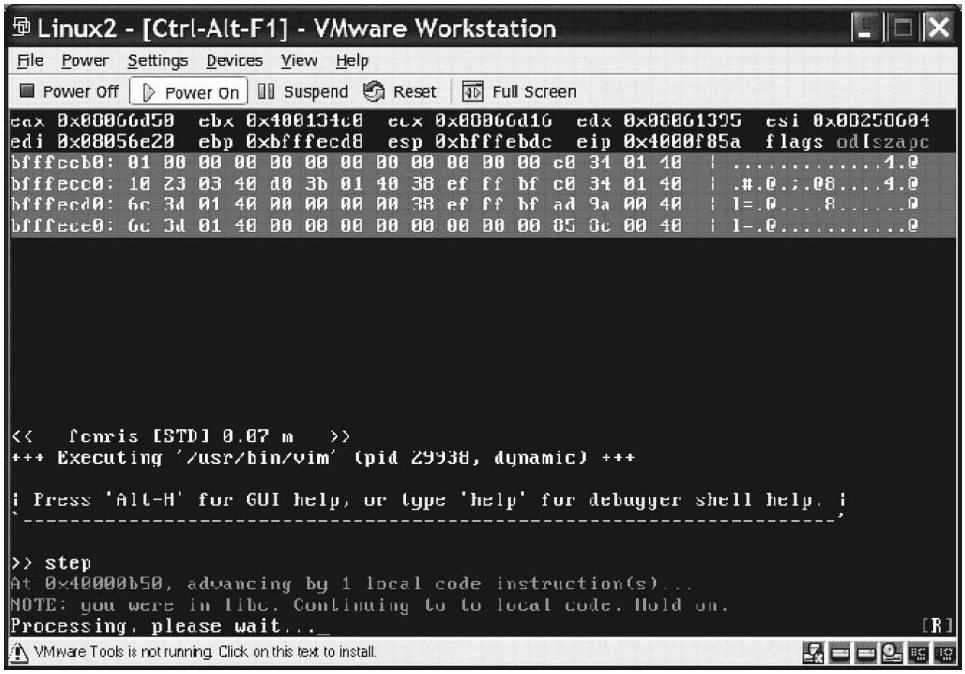


Рис. 6.4. Экран программы Fenris, запущенной в виртуальной машине для проведения анализа кода во время выполнения

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.

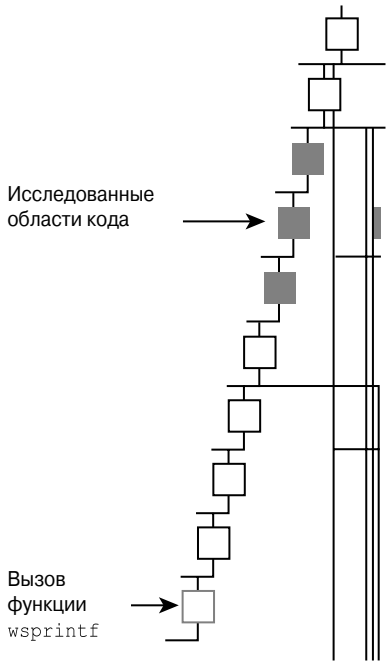


Рис. 6.5. Результаты исследования кода с помощью нашей простой программы анализа охвата кода. Исследованные блоки кода выделены серым цветом. Нам не удалось найти пути к уязвимому блоку кода, в котором содержится вызов функции `wsprintf()`

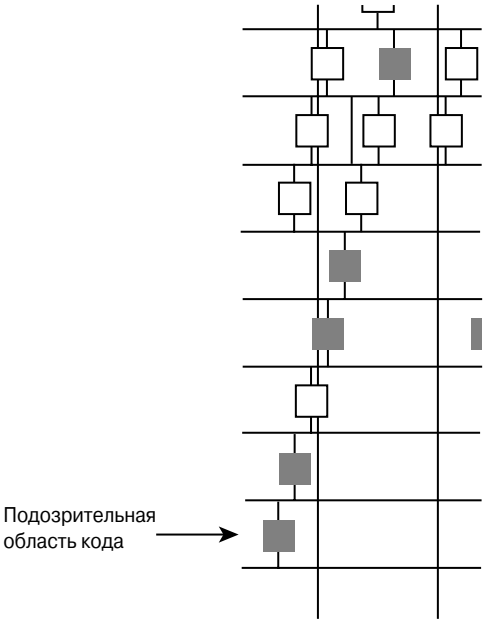


Рис. 6.6. В данном случае нам удалось добраться до уязвимой области кода с помощью специально подготовленных входных данных

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/).

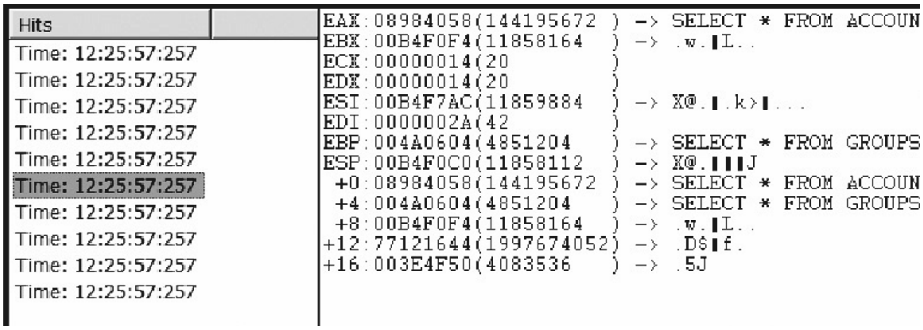


Рис. 6.7. Простая программа для проведения быстрых остановов использована для выборки данных из памяти, выделенной для FTP-сервера. В крайне левом столбце показано время, в которое была выполнена выборка

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.

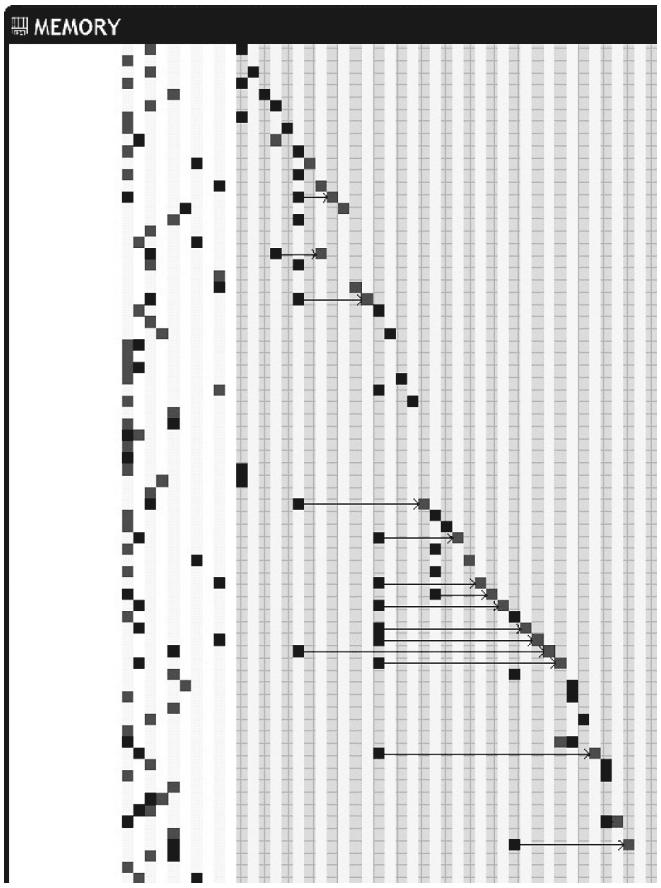


Рис. 6.8. Трассировка памяти показывает регистры (слева) и области памяти в стеке и куче (справа). Более темные квадратики — это источник (операция чтения), более светлые — это цель (операция записи). Стрелки указывают на отправителя и получателя при выполнении операции `move`*. Эта программа была создана 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.

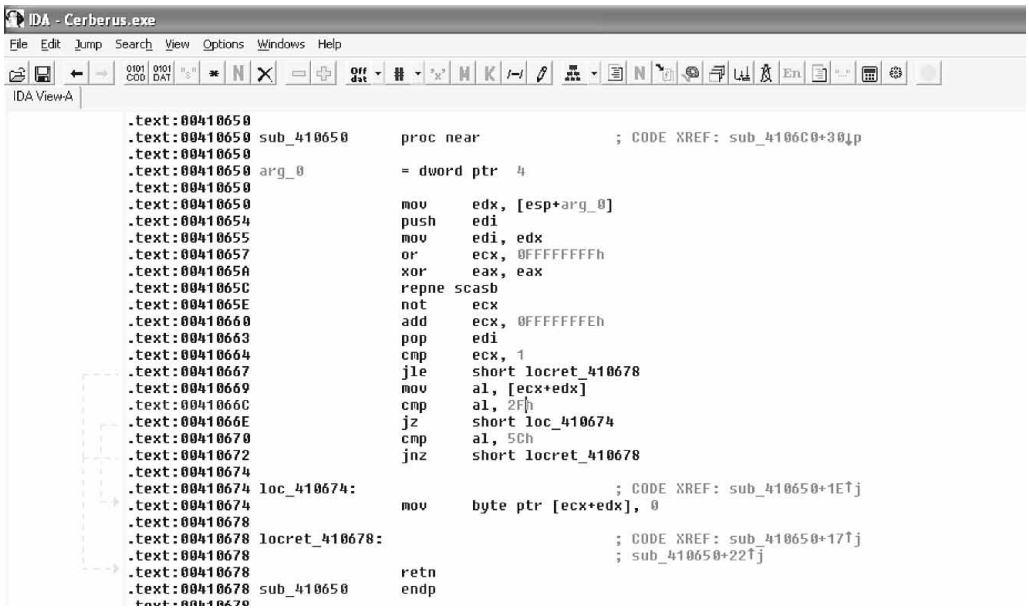


Рис. 6.9. В дизассемблированном с помощью IDA коде стандартного FTP-сервера выполняется поиск символов 2F и 5C

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

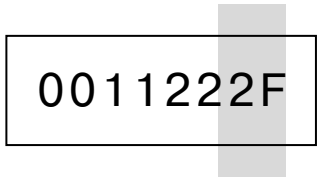


Рис. 6.11. Представление одного байта (2F) в 32-битовом регистре

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.

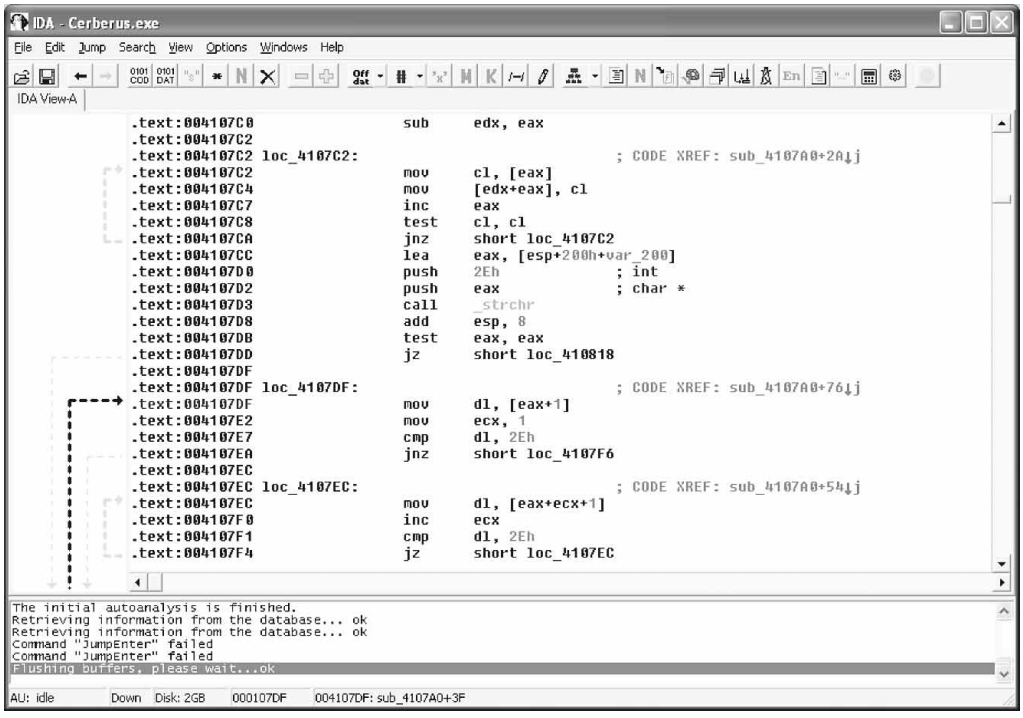


Рис. 6.12. Программный код, в котором содержатся некоторые интересные побайтовые операции

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.

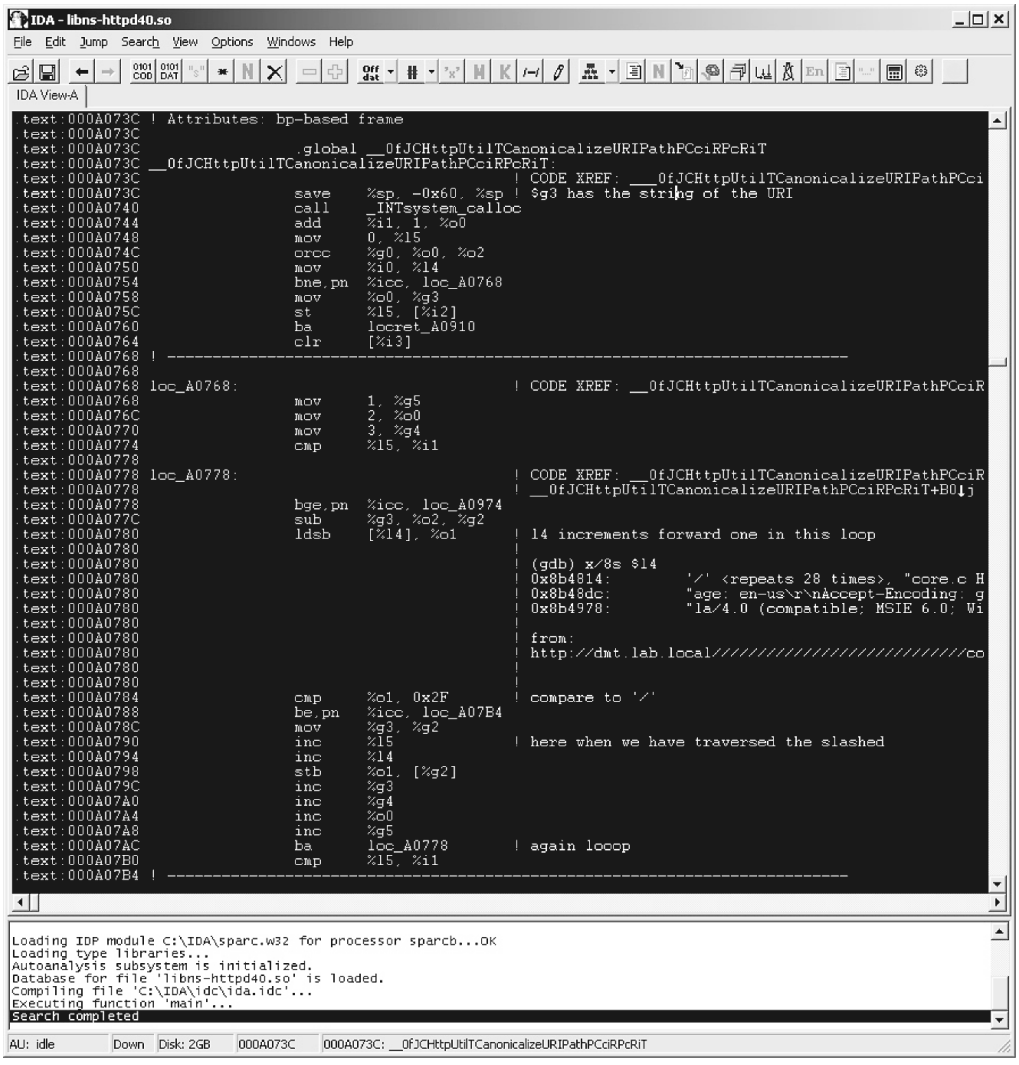


Рис. 6.13. Экран программы IDA с замечаниями, добавленными к коду

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.