3 Восстановление исходного кода и структуры программы

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

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

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

---

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

Внутри программы

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

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

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

Восстановление исходного кода

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

---

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

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

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

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

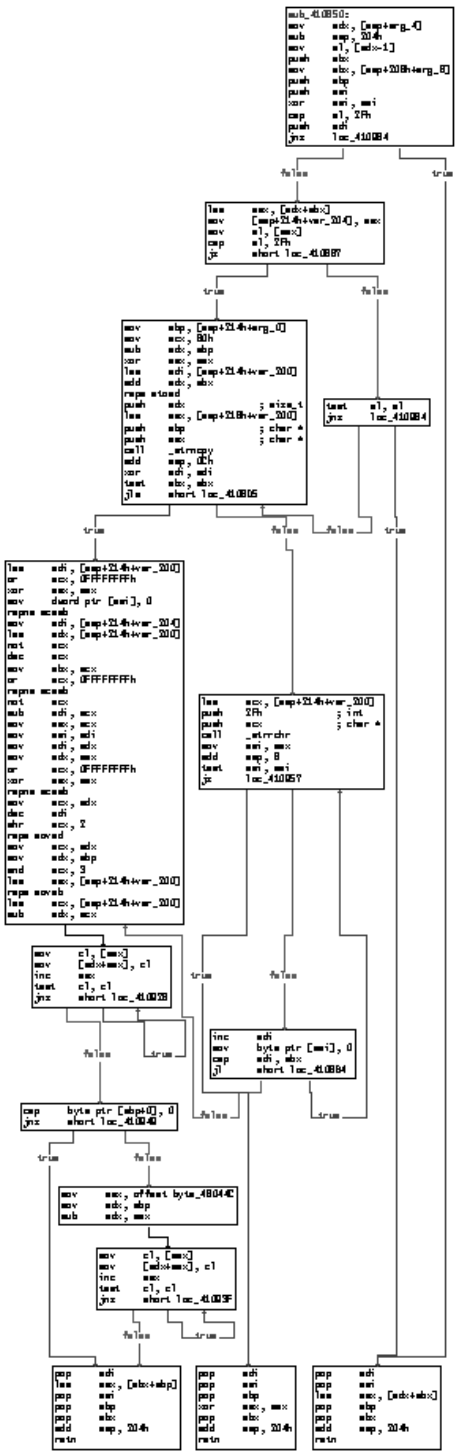


Рис. 3.1. На этом рисунке показана блок-схема выполнения программы стандартного FTP-сервера. Каждый блок представляет собой набор инструкций, которые запускаются в едином пакете, одна за другой. Линии между блоками отображают передачу управления. Во многих случаях решение относительно того, какая ветвь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

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

Зачем это нужно?

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

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

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

Насколько это легально?

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

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

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

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

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

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

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

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

Средства для восстановления исходного кода

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

Отладчик

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

Средства для внесения ошибок

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

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

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

Дизассемблер

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

Декомпилятор

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

Методы для восстановления исходного кода

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

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

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

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

Этот список первоочередных целей необходим при исследовании двоичного кода.

Исследование по методу "белого ящика"

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

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

Исследование по методу "черного ящика"

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

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

---

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

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

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

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

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

Исследование по методу “серого ящика”

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

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

---

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

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

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

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

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

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

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

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

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

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

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

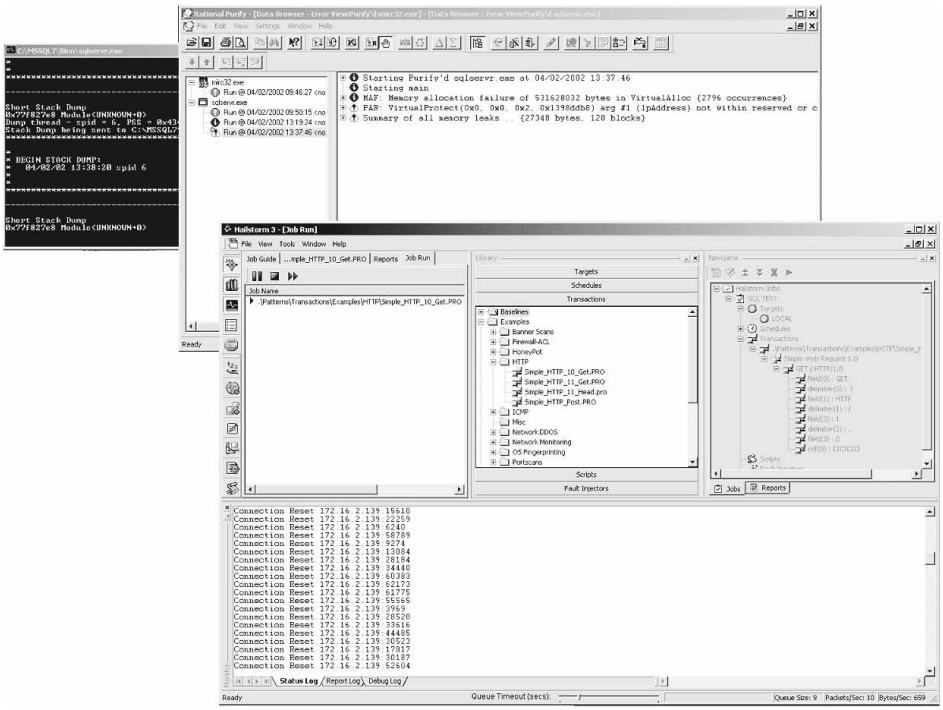


Рис. 3.2. Снимки экранов с отчетами программ Hailstorm и Purify, которые были сделаны в результате тестирования программного обеспечения SQL-сервера

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

Методы исследования

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

Отслеживание обработки входных данных

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Использование отличий в версиях программ

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

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

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

Использование охвата кода

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

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

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

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

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

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

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

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

Доступ к ядру

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

Утечка данных из совместно используемых буферов

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

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

---

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

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

Пример: ошибка в сетевых адаптерах Ethernet

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

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

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

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

Поиск недостатков в предоставлении прав доступа

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

---

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

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

Использование вызовов функций API

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

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

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

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

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

---

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

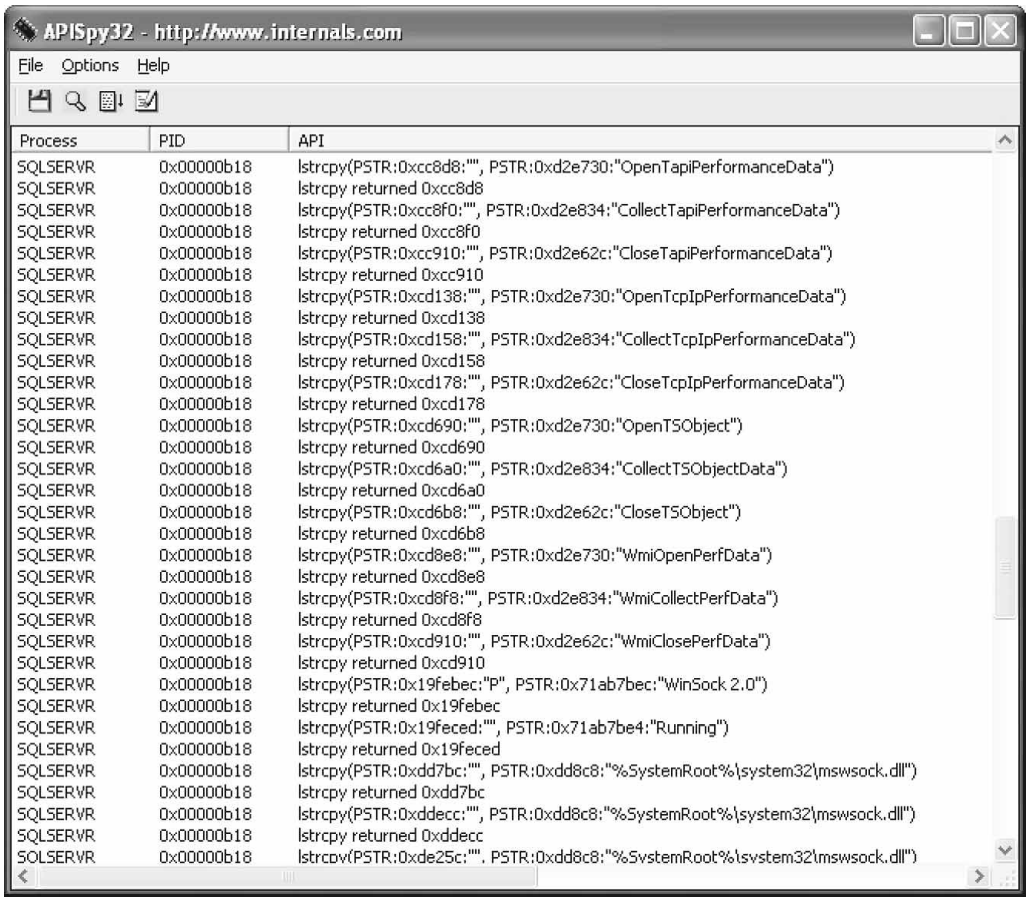


Рис. 3.3. Программа APISPY32 позволяет обнаружить вызовы в программном коде SQL-сервера. На снимке экрана показан отчет о выполнении запроса

Создание дополнительных модулей для IDA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

}

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

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

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

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

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

Добавим еще несколько заголовочных файлов и несколько глобальных переменных.

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

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

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

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

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

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

```
   g_file_size = aFileSize;
   g_mapped_file = aFilePointer;

   return TRUE;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    strcpy( szFile, "");
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

run,          // Запустить дополнительный модуль.

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

help, // Многострочная помощь для дополнительного модуля


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

Декомпиляция и дизассемблирование программного обеспечения

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

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

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

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

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

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

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

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

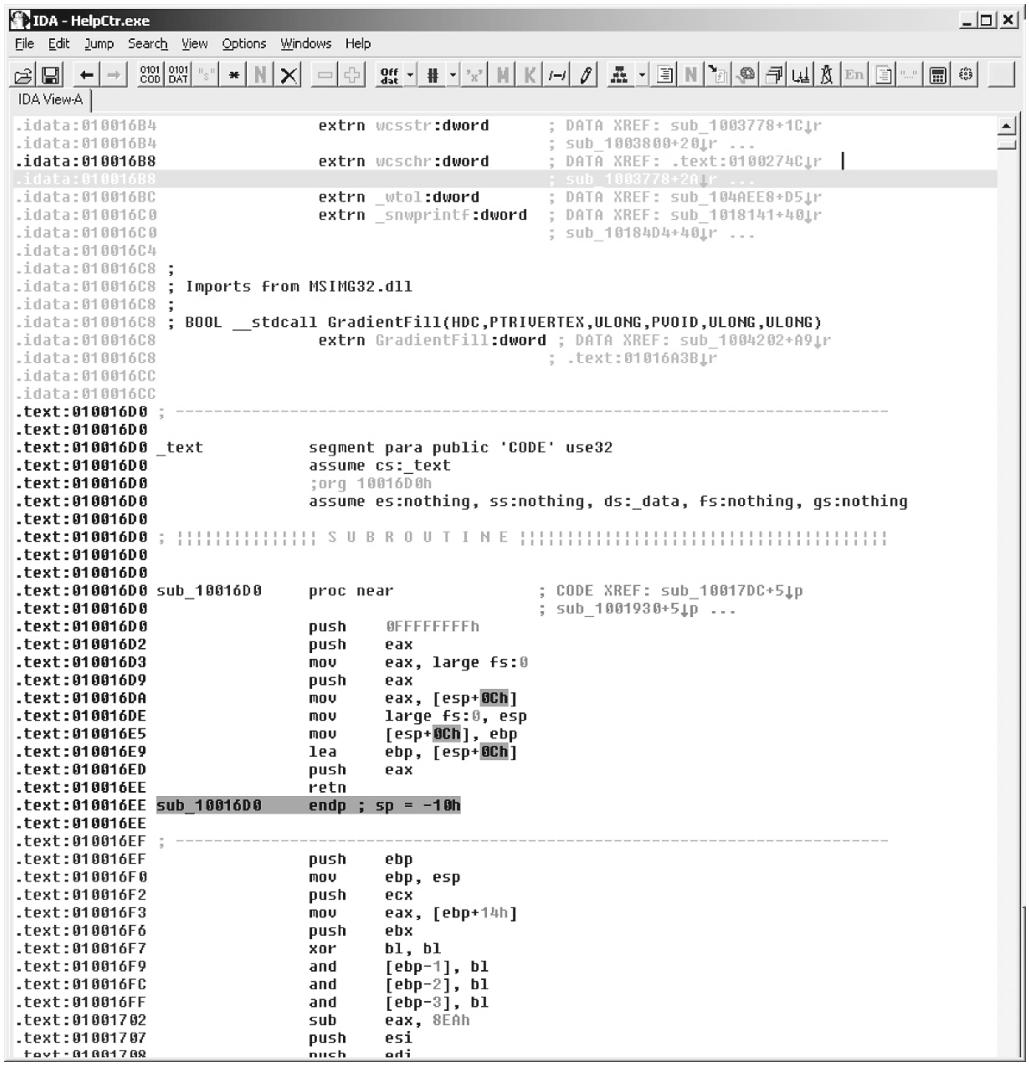


Рис. 3.4. Окно программы IDA-Pro при восстановлении исходного кода программы `helpctr.exe`*, которая является частью операционной системы Windows XP. В качестве примера мы пытаемся найти в* `helpctr.exe` *уязвимое место для атаки на переполнение буфера*

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

Журнал ошибок

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

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

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

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

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

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

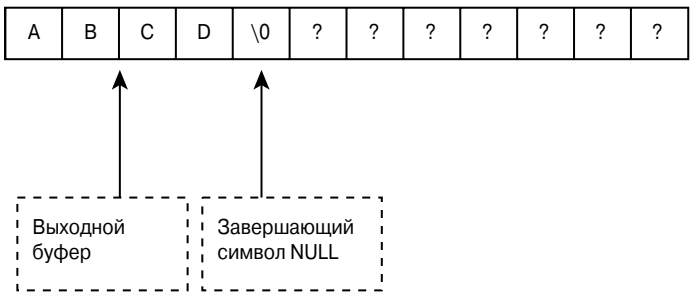
Правила работы с функцией `wcsncat` очевидны, к тому же получить описание можно из руководства пользователя. Вызову передается три параметра.

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

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

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

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

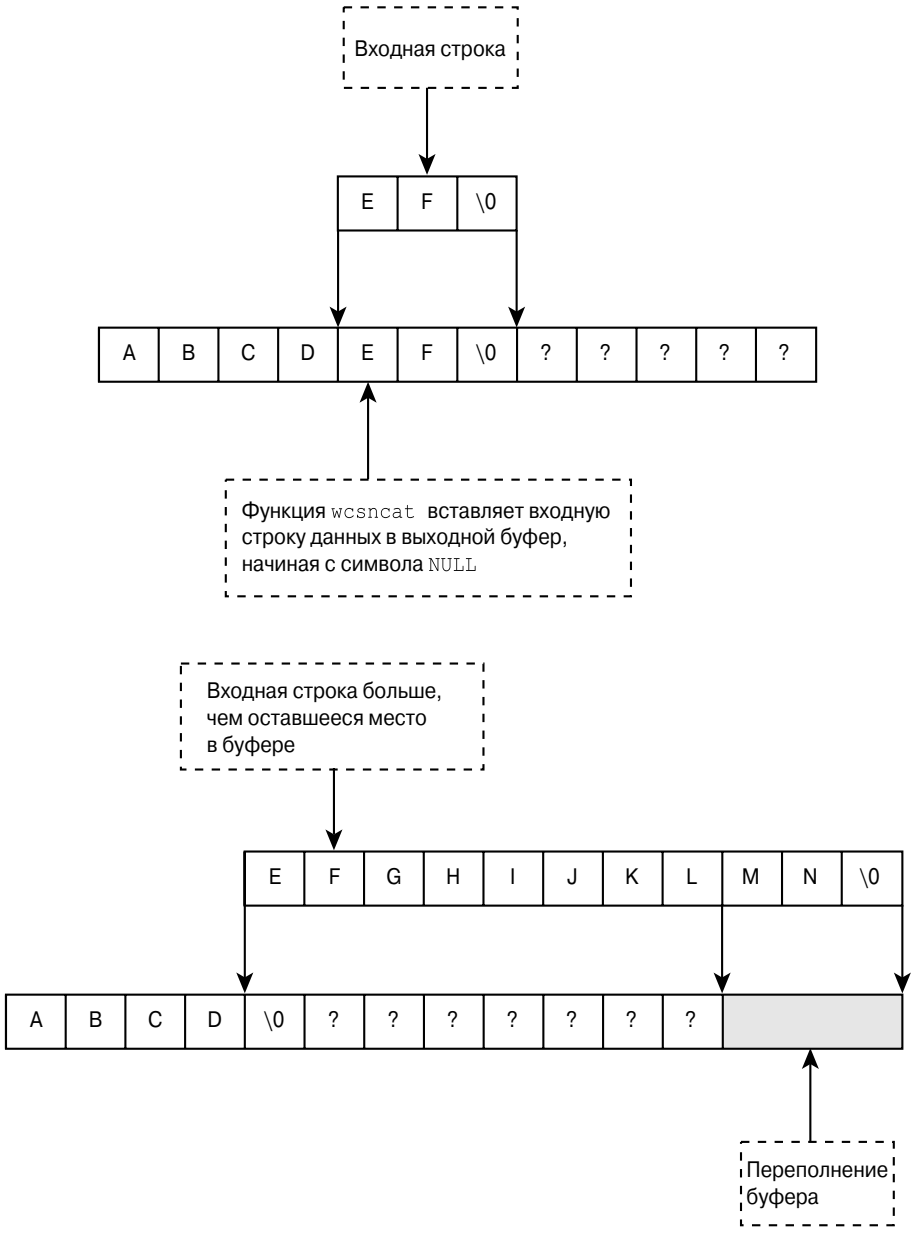


Теперь вызовем функцию `wcsncat()` и добавим строку `EF`. Как видно на следующей схеме, эта строка добавляется в выходной буфер, начиная с символа `NULL`. Для защиты выходного буфера мы должны указать, что максимальное количество добавляемых символов не должно превышать семи. С учетом завершающего символа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

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

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

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



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

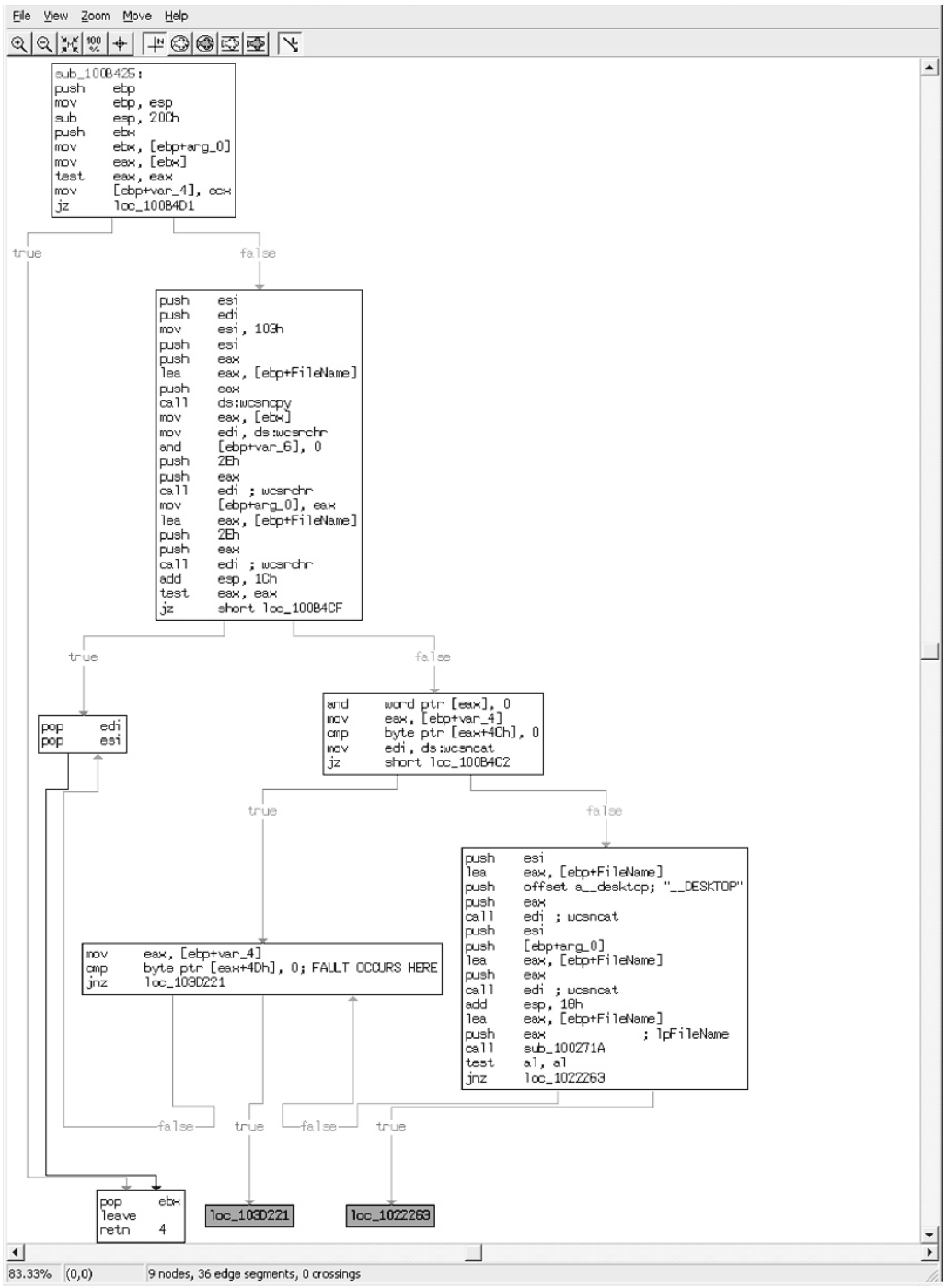


Рис. 3.5. Простая схема подпрограммы в `helpctr.exe`*, которая выполняет вызовы функции* `wcsncat()`

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

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

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

Автоматизированный глобальный аудит для выявления уязвимых мест

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

C:\temp>perl scan.pl

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

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

Глобальный анализ с помощью IDA-Pro

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

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

Ниже приведен элементарный файл IDC-сценария.

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

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

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

    system($command);

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

Мы используем сценарий `bulk_audit_sprintf.idc`.

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

#include <idc.idc>

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

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

   k = GetMnem(eb);

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

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

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

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

```
{
   auto my_reg;
   auto max_backtrace;

   ep = eb;

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

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

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

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

               my_loc = PrevFunction(eb);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Создание собственных средств взлома

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

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

Средства для платформы x86

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

Отладчик для платформ x86

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

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

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

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

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

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

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

Точки останова

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    if(!m_persistent)
    {
        m_refcount++;
    }

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

    // TODO: Flush instruction cache.

    return TRUE;
}
```

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

Чтение и запись в память

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

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

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

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

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

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

Отладка многопотоковых программ

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

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

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

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

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

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

    CloseHandle(hThread);

    return TRUE;
}
```

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

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

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

FOPENTHREAD fOpenThread=NULL;

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

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

Инвентаризация потоков или процессов

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

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

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

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

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

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

Пошаговый режим

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

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

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

```
bool SetSingleStep()
{
    CONTEXT ctx;

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

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

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

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

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

Установка заплат

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

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

Внесение ошибок

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

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

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

---

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

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

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

Фиксирование состояния процесса

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

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

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

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

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

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

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

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

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

std: :list<struct mb *> gMemList;

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

    while(start < 0xFFFFFFFF)
    {
        MEMORY_BASIC_INFORMATION mbi;

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

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

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

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

```
            gMemList.push_front(b);
        }

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

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

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

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

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

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

Дизассемблирование машинного кода

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

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

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

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

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

DWORD ulOffset = dp.context.Eip;

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

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

CloseHandle(hThread);
```

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

Создание базового средства для охвата кода

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

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

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

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

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

    std::list<struct item *> mChildren;

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

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

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

```
struct item *anItem = NULL;

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

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

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

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

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

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

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

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

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

Проверка тегов boron

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

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

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

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

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

return FALSE;
}
```

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

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

Резюме

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

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