наведение порядка в хаосе атак
или классификация ошибок переполнения

крис касперски a.k.a. nezumi, a.k.a. souriz, a.k.a. elraton, no-email

дыра — это нора, а в норе — ароматный ужин, как правило, состряпанный из переполняющихся буферов, которым посвящены десятки тысяч статей и разработаны сотни защитных механизмов, от обилия которых рябит в глазах и без систематического руководства новичку очень легко свернуть голову или зажевать прокисший ужин, соблазнившись статьей двух-трех летней давности, забыв о том, что методики атак на ошибки переполнения — весьма скоропортящейся продукт. вот мыщъх и притаранил свежачок!

введение

Количество _типов_ локальных/удаленных атак, прямо или косвенно связанных с ошибками переполнения, неуклонно растет. Защитные механизмы так же не стоят на месте, но откровенно запаздывают, оставаясь в роли догоняющих. А на передовой линии хакерского фронта царит полный хаос, совершенно неподдающийся никакой классификации. Дыры (с чисто формальной точки зрения) делятся на семейства, типы и подтипы, но при внимательном анализе всякой классификации выявляются подтипы из различных семейств, описывающих одну и туже дыру, и классификация летит к черту!

Тем не менее, без систематизации не обойтись, потому как невозможно каждый раз описывать все характеристики дыры от начала и до конца. Мыщъх не предлагает своей собственной классификации и не придумывает новых терминов, а упорядочивает уже существующие, иногда (в силу сложившихся исторических ситуаций) достаточно нелепые, но ставшими общепринятыми де-факто.

переполняющиеся буфера

Возможность использования ошибок переполнения для хакерских атак была осознана и теоретически обоснована еще в 1972 году Джеймсом Андерсеном (James Anderson), а спустя десяток лет, 2 ноября 1988, впервые опробована в достопочтенном Черве Морриса, использовавшего ошибку переполнения в UNIX-демоне finger. После сокрушительной эпидемии на хакерском фронте наступило неожиданное затишье, но с конца 90х годов XX века атаки на переполняющиеся буфера вспыхнули с новой силой, да так вспыхнули, что едва не погрузили мир в средневековую тьму — хорошо, что ни один из червей не содержал в себе деструктивной начинки.

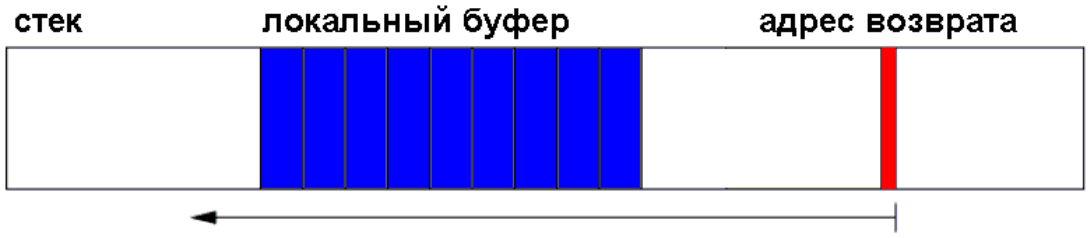

Рисунок 1 состояние стека до переполнения локального буфера

Какова же природа сатаны, с которым приходится иметь дело? А вот такая: локальные буфера находятся в стеке (см. рис. 1) и при их переполнении (традиционное отсутствие проверки длины перед копированием) происходит затирание адреса возврата из функции (вместе с остальными буферами, скалярными переменными и указателями, встретившимися на пути). Если только функция не грохнется еще до своего завершения, то произойдет переча управления по адресу, записанному поверх адреса возврата (см. рис. 2), и в зависимости от "настроения" хакера отправляющего процессор в "космос" (т. е. по случайному адресу, высаживающего жертву на DoS), либо же вызывающего shell-код, по обыкновению

расположенный непосредственно в переполняющемся буфере, а в исключительных случаях — где-то в другом месте. Техника передачи управления кратко описана в одноименной врезке.

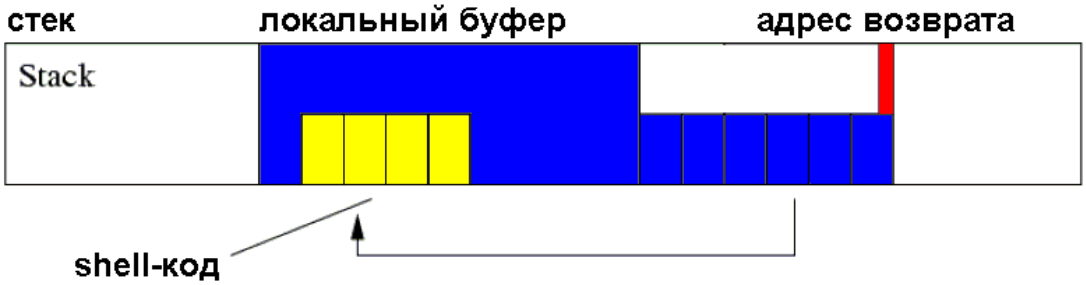

Рисунок 2 состояние стека после переполнения локального буфера

Динамические буфера размещаются в куче (heap), сезон переполнения которой открылся статьей "**Once upon a free()**", опубликованной 8 января 2001 года неизвестным хакером в #39h номере электронного журнала phrack со ссылкой на исследовательскую работу Solar'a Designer'a, восходящую к 25 июля 2000 года и описывающую уязвимость библиотеки glibc-2.2.3, допускающей передачу управления на произвольный код или (внимание!) **модификацию произвольных ячеек памяти** (например, указателей на функции), что открывает поистине безграничные возможности для атакующего.

Помимо стека и кучи, еще имеется и секция данных, где располагаются статические буфера, а так же большое количество указателей на функции (особенно в Си++ программах с их таблицами виртуальных функций), однако, в силу некоторых обстоятельств, атаки данного типа большого распространения так и не получили.

>>> врезка передача управления на shell-код

Кажется, если атакующий может перезаписывать адрес возврата (или любой другой указатель на функцию), то проблема передачи управления на shell-код решается сама собой, но все не так просто! Допустим, переполняющийся буфер расположен в стеке, а стек, как известно, растет снизу вверх (или сверху вниз — это уж кому как привычнее), и точное положение указателя вершины стека неизвестно. Следовательно, неизвестна и локация shell-кода. Так куда же передавать управление?!

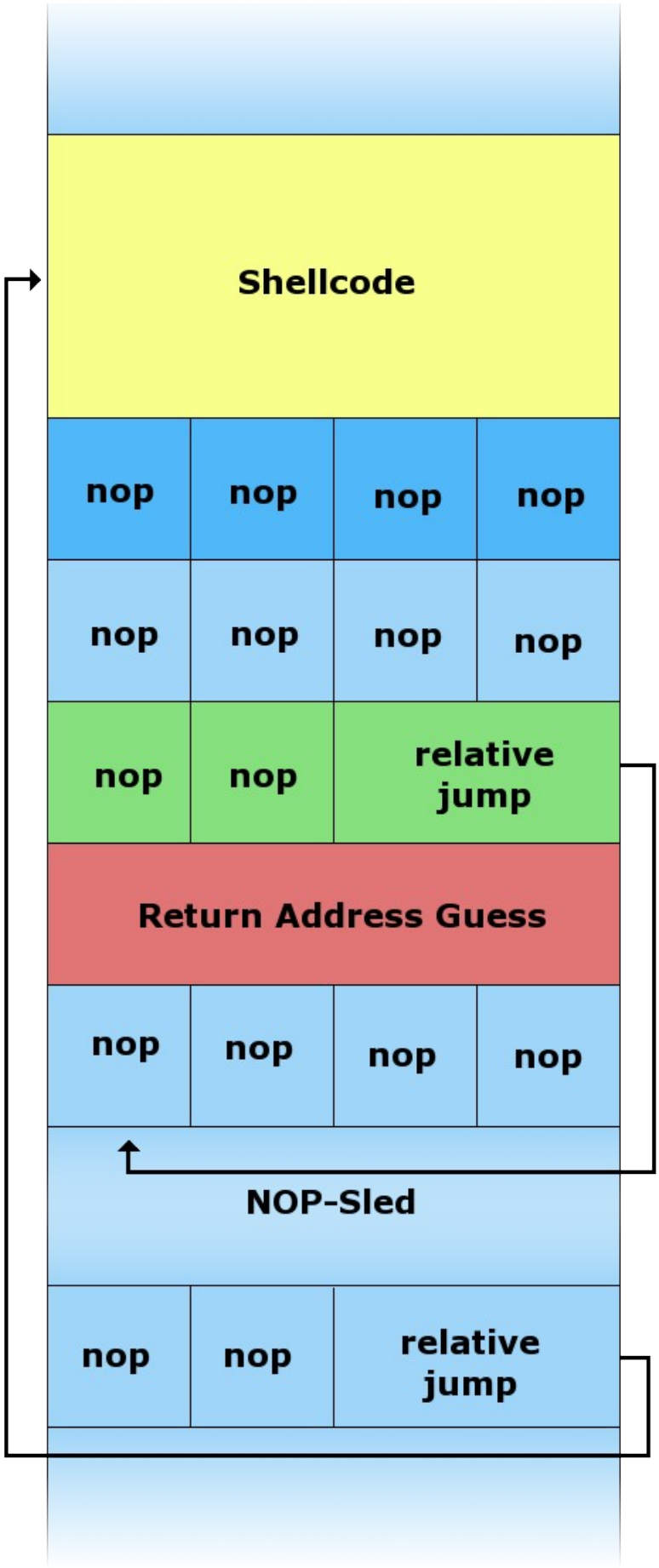
Shellcode
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
relative jump
Return Address Guess
nop
nop
nop
nop
NOP-Sled
nop
nop
relative jump

Рисунок 3 иллюстрация NOP SLED техники

Одно из решений проблемы (известное под именем **NOP SLED** техники) заключается в дописывании в конец буфера большого количества незначащих инструкций NOP (которым на x86 процессорах соответствует опкод 90h, тождественный операции XCHG EAX,EAX — обмен содержимого регистра EAX с регистром EAX), в конце которых стоит команда относительного (relative) перехода на начало shell-кода, не требующая знания абсолютных адресов (неизвестных атакующему).

При этом, NOP'ы оказываются расположены как до адреса возврата, так и после. Естественно, если управление будет передано "вперед", то цепочка управления, докатившись до адреса возврата, попытается интерпретировать его как машинную команду, со всеми вытекающими отсюда последствиями типа непредсказуемого поведения, поэтому, перед адресом возврата вставляется еще одна команда относительного перехода (см. рис. 3).

Однако, для реализации NOP SLED-техники, хакеру должен быть известен хотя бы _приблизительный_ адрес буфера с shell-кодом, а известен он далеко не всегда и тогда приходится прибегать к другой технике, передающей управление на вершину стека через команду **JMP ESP**, в x86-процессорах представляющую собой двухбайтовую машинную инструкцию с опкодом **FFh E4h**. Вся хитрость в том, чтобы найти такую последовательность байт в памяти и подсунуть ее адрес на место адреса возврата из функции. Тогда в момент стягивания последнего со стека, регистр ESP будет смотреть на двойное слово, следующее за адресом возврата, где может быть либо сам shell-код, либо команда перехода к нему.

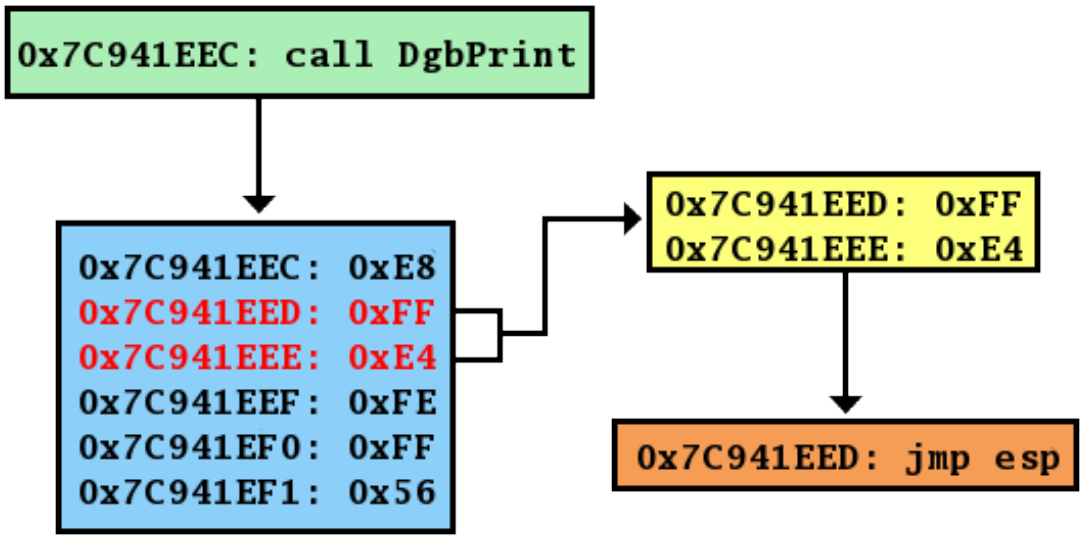

Рисунок 4 иллюстрация техники JMP ESP

Если целевая операционная система (или атакуемое приложение) известна с точностью до версии — найти двухбайтовую последовательность не проблема. Не обязательно искать именно JMP ESP. FFh E4h вполне может быть и частью совсем другой команды, например, инструкции CALL DbgPrint с опкодом E8h **FFh E4h** FEh FFh 56h (см. рис. 4). На машинах с неисполняемым стеком/кучей, последовательность FFh E4h необходимо искать только в кодовых секциях динамических библиотек или исполняемом файле атакуемого приложения. Если же защиты нет (или отключена) подойдет и область данных.

На системах с поддержкой ASLR данная техника не работает, поскольку невозможно заранее определить адрес искомой последовательности, однако, если атакуемый исполняемый файл не содержит перемещаемых элементов (а чаще всего так и бывает) или же одна из его динамических библиотек имеет сброшенный бит рандомизации (состояние по умолчанию), шансы на успешную атаку многократно возрастают.

хронология технологий защиты стека

Еще в древних компиляторах, написанных в эпоху MS-DOS, была предусмотрена опция, отвечающая за контроль границ буферов, а в x86 процессоры встроена команда BOUND, генерирующая исключение в случае выхода за границы буфера, однако, все эти технологии по разным причинам остались невостребованными. Первое (и главное!) — среднестатистических

программист не осведомлен об угрозе переполнения, а проверка границ увеличивает размеры программы и тормозит ее выполнение, к тому же, ошибку переполнения надо как-то обрабатывать, иначе компилятор просто вызовет функцию аварийного завершения программы.

Даже сегодня, когда процессоры летают со скоростью пули, подавляющее большинство программ компилируются с отключенным контролем границ буферов, впрочем, некоторые языки программирования (и, в первую очередь, Си/Си++) никакие проверки не спасают, поскольку, полноценной поддержки массивов в них нет и программистам приходится оперировать указателями на безразмерные блоки памяти. Как следствие — ошибки переполнения носят характер фундаментальной проблемы, не имеющей общего решения.

Рисунок 5 реализация защиты адреса возврата в GCC и MS VC

Разработчикам компиляторов приходится извращаться и ходить совсем другим путем. Некогда популярное расширение для компилятора GCC (уже давно интегрированное в него) со скромным называнием Stack-Guard, модифицирует стековый фрейм путем помещения специального "сторожевого" слова перед адресом возврата (сначала представляющего собой константу, а затем случайно генерируемое значение). В код эпилога добавляется проверка целостность сторожевого слова на предмет его затирания хакером. Аналогичная техника используется и в последних компиляторах от Microsoft, поддерживающих ключ /GS, форсирующий проверку целостности адреса возврата (см. рис. 5).

Рисунок 6 pro-police — расширение для GCC, защищающее адрес возврата специальным сторожевым словом

Недостаток защит подобного типа в том, что они защищают лишь сам адрес возврата, но не препятствуют затиранию предшествующих ему переменных, среди которых часто встречаются указатели на функции, позволяющие хакеру передавать управление по любому адресу, которому ему только вздумается. Последние версии GCC поддерживают множество дополнительных расширений (см. рис. 6), "оборачивающих" буфера страницами памяти с атрибутами NO_ACCESS, всякая попытка доступа к которым вызывает исключение, а так же

шифрующих указатели, хранящиеся в памяти случайно сгенерированной константой по XOR. Накладные расходы на защиту (оверхих), конечно же, существенно возрастают, однако, вместе с этим затрудняется и сама атака. Впрочем, к счастью (для хакеров) подавляющее большинство программ поставляются в незащищенном виде.

хронология защиты кучи

Борьба с переполнением динамических буферов осложняется иерархическим обустройством кучи. На самом нижнем уровне находится базовый аллокатор, встроенный в операционную систему, однако, прикладные программы обращаются к нему редко, предпочитая действовать через библиотечные вызовы конкретного компилятора, оптимизированные под выделение небольших блоков памяти. Защита кучи операционной системы без защиты библиотек _всех_ популярных компиляторов (как правило, прилинкованных статическим образом, т. е. требующих перекомпиляции уже существующих программ) _ничего_ не дает. А вот обратное утверждение неверно, хотя если в программе используются прямые вызовы базового аллокатора, а он не защищен — то это ласты (ну, кому и ласты, а кому радость от очередной удачно свершившейся атаки).

Разработчики всех операционных систем: BSD, Linux, Windows прилагают нехилые усилия по защите базового аллокатора, воздвигая многоуровневую линию обороны, призванную обеспечить контроль целостности кучи и не допустить затирания служебных структур данных. Microsoft отчаянно пропагандирует защиту кучи в Висле (впрочем, уже давно поломанную), забыв о том, что это никак не препятствует атакам. А проверка целостности кучи на уровне RTL конкретных компиляторов, ни в DELPHI, ни в MS VC, ни даже в последних версиях C# должным образом так и не реализована и все это хозяйство (неважно — работающее под W2K, XP или Вислой) атакуется влет.

Библиотека LIBC (стандартная библиотека в мире Linux/BSD) и GLIBC (стандартная библиотека компилятора GCC) защищена намного сильнее, но больше всего хакеров высаживает то, что в различных версиях этих библиотек применяются различные аллокаторы, а без точного знания схемы размещения служебных структур кучи ее не атакуешь — в лучшем случае получится отказ в обслуживании. Написание универсальных exploit'ов весьма затруднено и для удачной атаки необходимо знать точную версию библиотеки, используемой жертвой, а определить ее удаленно не так-то просто!

хронология неисполняемого стека/кучи

Запрет на исполнение кода в стеке/куче/сегменте данных когда-то казался весьма радикальным решением проблемы, гарантирующим мир и покой. А все потому, что руководящие работники не привыкли думать головой. Ни своей, ни чужой.

Неисполняемый стек/куча впервые появился в UNIX-системах, причем появился весьма давно. Парням из Microsoft для достижения аналогичного результата понадобилась специальная аппаратная поддержка со стороны процессоров, которая была предоставлена с большим запозданием. Проблема (если это можно назвать проблемой) в том, что UNIX (равно как и Windows) поддерживает линейное адресное пространство, выделяющее в распоряжение каждого процесса 4 Гбайта виртуальной памяти, в которых размещается: код операционной системы, код программы (со всеми динамическими библиотеками), секция данных, стек и куча. x86-процессоры поддерживают раздельные селекторы для кода, данных и стека — каждый со своими атрибутами, разрешающими (или не разрешающими): чтение, запись и исполнение, однако, для упрощения кода операционной системы разработчики Windows "распахнули" селекторы кода/стека/данных на все адресное пространство, присвоив им идентичные лимиты и атрибуты защиты. Так же поступили и разработчики первых версий Linux/BSD.

На уровне отдельных страниц, x86 процессоры поддерживают только два атрибута защиты: доступа и записи, при этом понятие "доступа" включает в себя как чтение, так и исполнение. И хотя, API-функции операционной системы формально поддерживают установку/снятие атрибута "исполняемый" со страниц памяти, вплоть до недавнего времени атрибуты чтения и исполнения были тождественны друг другу.

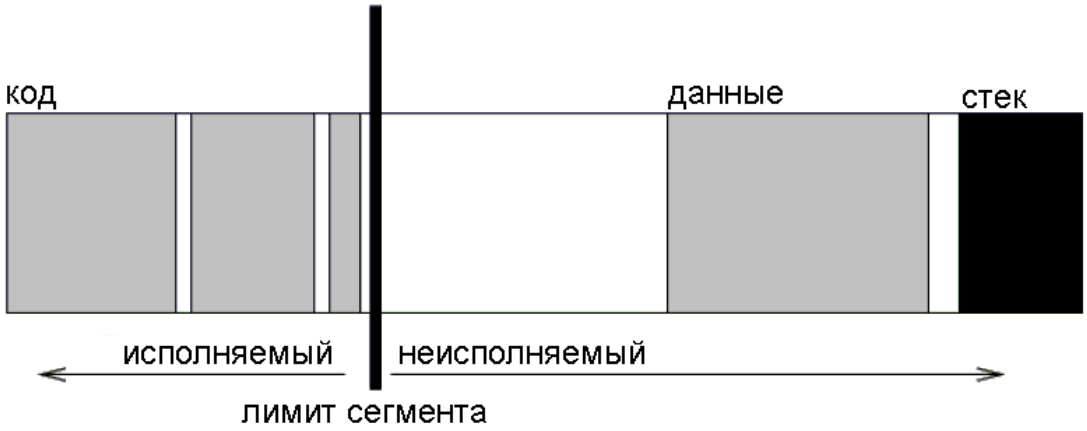

Рисунок 7 защита стека и кучи от исполнения, реализованная на древних x86-процессорах, поддерживающих атрибут "исполняемый" только на уровне селекторов

Первыми спохватились разработчики Linux/BSD. Они "разнесли" стек/кучу и код по разным концам адресного пространства, скорректировали лимиты селекторов (см. рис. 7), в результате чего стек/куча оказались совершенно неисполняемыми и хакеры конкретно приуныли. Однако, ряду честных программ (например, компиляторам, транслирующим код в оперативную память) пришлось либо не хило извратиться, чтобы преодолеть все "прелести" этих нововведений, либо объявить "забастовку", как большинство из них и поступило.

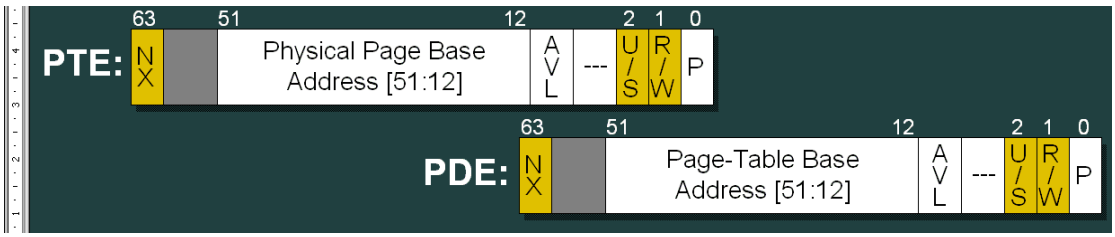

Рисунок 8 современные x86 и x86-64 процессоры поддерживают атрибут "исполняемый" на уровне отдельных страниц

Так что поддержка атрибута "исполняемый" на уровне отдельных страниц в последних версиях x86-процессоров (см. рис. 8) пришлась ко двору не только Windows, но и Linux/BSD. Но не успели разработчики опохмелиться после сдачи "защищенного" релиза, как хакеры уже изобрели атаку, получившую название **return2libc** и в общих чертах сводящуюся к засылке в стек указателей на функции, выделяющие блок памяти с атрибутами на запись+исполнение и тут же копирующие в него shell-код, с передачей на него управления классическим способом. При этом в стеке оказывался не код, а данные — указатели на функции, замещающие оригинальный адрес возврата. И хотя в Windows нет библиотеки LIBC, зато там есть KERNEL32.DLL и потому атака return2libc работает на ура даже с неисполняемым стеком (см. рис. 9).

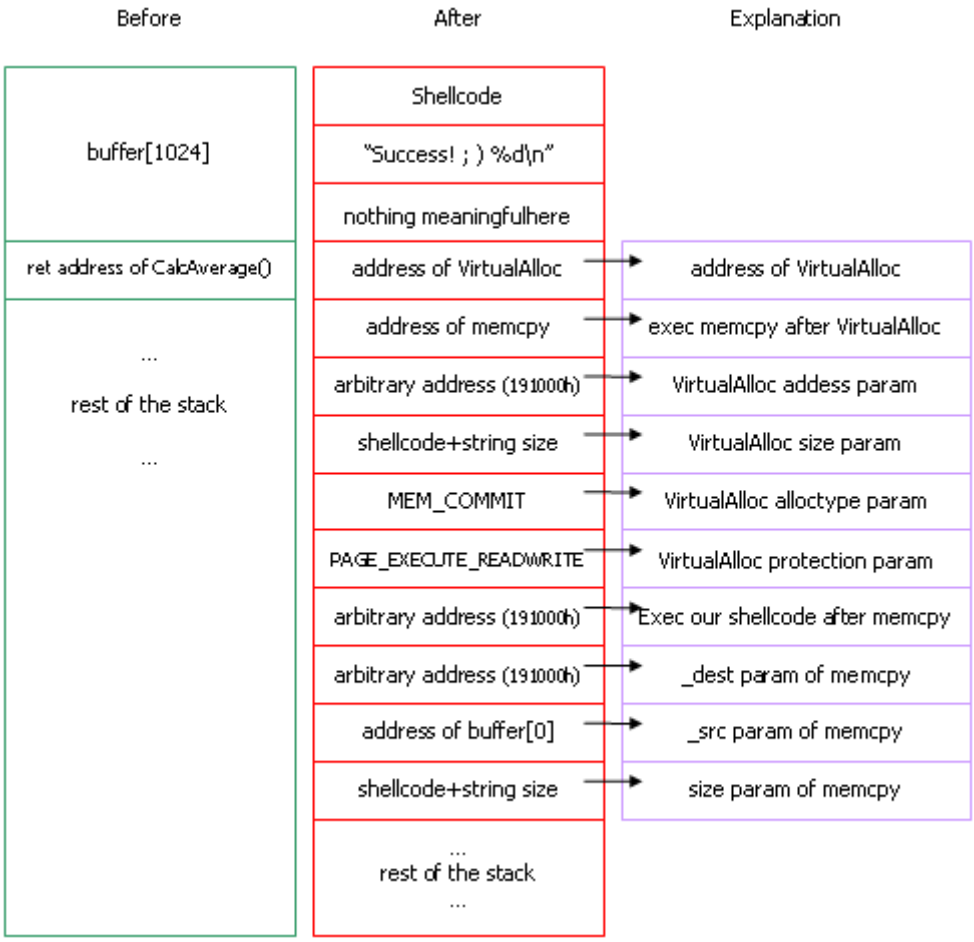

Рисунок 9 реализация атаки return2libc, пришедшей из мира UNIX, на Windows-системах с неисполняемым стеком

Как водится, первыми отреагировали разработчики Linux/BSD (толи пьют они меньше, толи трезвеют быстрее). Пакет PaX (кстати говоря, портированный и под Windows) выполняет рандомизацию адресного пространства (Address Space Layout Randomization или, сокращенно, ASLR), размещая стек, кучу и системные библиотеки по случайным адресам, в результате чего хакер уже не может просто так засунуть в стек указатели на необходимые ему функции, ведь их местоположение заранее неизвестно!

Разработчики Open-BSD пошли другим путем, внедрив технологию W^X (что расшифровывается как "W XOR X"), препятствующую одновременную установку атрибутов записи и исполнения, что существенно затрудняет атаку, однако, PaX все-таки круче, и потому спустя некоторое время коллектив Open-BSD дал ему добро, предоставив пользователю возможность выбора: какую защитную систему использовать.

ASLR, реализованный должным образом, действительно, представляет серьезное препятствие для атакующих, однако, даже в Linux/BSD часть критических структур данных по-прежнему располагается по вполне предсказуемым адресам. Что же касается Windows, то ASLR там поддерживается только начиная с Вислы и реализован настолько криво, насколько это только вообще возможно, к тому же (внимание!) ранее написанные программы с убитой таблицей перемещаемых элементов, всегда загружаются по одному и тому же базовому адресу и в принципе не поддаются рандомизации, так что для защиты от атак мало установить Вислу на своей компьютер. Как минимум требуется перекомпилировать все используемое программное обеспечение, а как его откомпилируешь, когда исходных текстов нет?!

А старые среды разработки DELPHI, Visual Basic вообще не поддерживают возможность установки бита рандомизации и помимо перекомпиляции, над сгенерированными файлами/динамическими библиотеками еще предстоит поработать руками (и головой) или же… полностью переписать проект на C#. Заманчивая, перспектива, не правда ли?! Так стоит ли удивляться, что существенного снижения хакерской активности ожидать не приходится, во всяком случае на ближайшие года два, а там… хакеры снова что-то придумают.

целочисленное переполнение

В большинстве языков программирования (и в языке Си в том числе) значение выражения (n + k) для целочисленных типов в общем случае неопределенно, то есть может быть равно арифметической сумме n и k, а… может быть и не равно!

При сложении двух беззнаковых типов x86-процессоры дают корректный результат лишь до тех пор, пока конечная сумма остается в пределах разрядной сетки, в противном же случае процессор выставляет знак переноса и мы имеем "заворот", то есть UCHAR_MAX + UCHAR_MAX == UCHAR_MAX-1 == FEh. Аналогичным образом дела обстоят и с UINT_MAX.

А вот со знаковыми типами все _гораздо_ интереснее. В x86-процессорах старший бит числа используется для задания знака (в некоторых процессорах за это отвечает младший бит, но разговор не о них). На 32-разрядных платформах INT_MAX = 2147483647, но (INT_MAX+1) == INT_MIN == -2147483648, то есть от наибольшего положительного до наименьшего отрицательного — всего один шаг! Ни процессор, ни компилятор _никак_ не реагируют на эту ситуацию и, если программист не озаботился рукотворными проверками, программа может выдать весьма неожиданный результат.

Но дальше еще интереснее. Обычно, то есть, по умолчанию, int представляет собой signed int, то есть знаковый тип, а вот функция malloc, выделяющая память (как и множество других функций подобного типа, включая memcpy), в качестве аргумента, задающего размер блока, принимает size_t, определенный в заголовочных файлах как unsigned int.

Посмотрим к чему приводит такое несоответствие. Возьмем следующий (кстати говоря, _очень_ широко распространенный) код:

```
foo(int len, char *p)
{
        char buf[MAX_SIZE];
        if (len > MAX_SIZE) return -1;
        memcpy(buf, p, len);
        …
        return 1;
}
```

Листинг 1 наглядная демонстрация знакового переполнения

Что произойдет, если в качестве len передать отрицательное число? Поскольку любое отрицательное число больше всякого положительного (очень умную мысль сказал, да?!), то выражение (len > MAX_SIZE) окажется ложно и переменная len благополучно "докатится" до функции memcpy, где небольшое отрицательное знаковое число превратится в очень большое положительное беззнаковое (INT_MIN = 80000000h) – именно столько байт памяти будет скопировано функцией memcpy, точнее, она _попытается_ их скопировать, но, поскольку 80000000h — это половина адресного пространства, выделенная процессу, из которой ему реально доступно еще меньше, дело закончиться исключением типа "нарушение доступа" и хакер получит "всего лишь" отказ в обслуживании.

А вот еще один пример вполне типичного кода:

```
bar(int len, char *s)
{
        char *p;
        p = (char *) malloc(len+1);
        *(((char*)memcpy(p,s,0, len))+1) = 0;
        return 1;
}
```

Листинг 2 еще один пример кода, подверженного знаковому переполнению

Программист, копирующий строку, выделяет на один байт больше, куда и ставит завершающий нуль (на тот случай, если *s окажется без завершающего нуля). На первый взгляд все ОК, но если в качестве len передать UINT_MAX, то при добавлении к нему единицы, функция malloc в качестве аргумента получит… нуль! А по стандарту, попытка выделения блока нулевого размера является вполне допустимой операцией и функция malloc обязана возвратить валидный указатель на… ну, технически, создать блок нулевого размера в памяти невозможно, поэтому, обычно выделяется блок минимально возможного размера, который только поддерживает данная реализация malloc (что-то около 16 байт), а вот дальше… функция

memcpy попытается скопировать туда UINT_MAX байт (FFFFFFFFh), что опять-таки приведет к нарушению доступа.

А что на счет захвата управления?! Даже в примерах, рассмотренных выше, он вполне возможен, поскольку, прежде чем "врезаться" в невыделенный регион памяти или область памяти, принадлежащую операционной системе (и, естественно, защищенную от записи), функция _memccpy имеет хорошие шансы перезаписать обработчики структурных исключений (как правило, хранящиеся в стеке) и тогда, при генерации исключения, вместо отказа в обслуживании, управление подхватит хакерский код!

В Linux/BSD никакого SEH'a нет (там для этого используются сигналы, реализованные совсем иначе и неподвластные атаке), а в Windows, начиная с XP, предпринята попытка защиты SEH-обработчиков от хакерских домогательств и развернута компания под названием SafeSEH. Вышел Server 2003, Висла, Server 2008, а SafeSEH все еще улучшается и улучшается, но так до ума и не доведена.

Если же с целочисленными переменными осуществляются махинации в стиле memcpy(dst, src, x*y+z), что вовсе не редкость, то у хакера появляется _реальная_ возможность получить в результате переполнения именно то число, которое ему нужно – то есть, превышающее размер выделенного буфера, но не такое большое, чтобы "вылететь" за пределы адресного пространства.

В принципе, некоторые компиляторы (например, GCC) поддерживают специальный ключ, форсирующий проверку на целочисленные переполнения, но… во-первых, она довольно сильно тормозит (и в случае переполнения опять-таки высаживает на отказ в обслуживании), а, во-вторых, от кастинга, то есть явного/неявного преобразования типов она _не_ спасает и пример, приведенный в листинге 1, с точки зрения компилятора — вполне законное программистское творение, а потому атаки данного типа прекращаться не собираются (к тому же лишь немногие программисты способы провести надлежащий аудит кода на предмет поиска багов).

заключение

Разумеется, разновидности атак на этом не заканчиваются и за кадром нашего короткого обзора остались удары по памяти, использование освобожденных буферов, неинициализированные локальные переменные и указатели, неспецифические разрушения памяти, ошибки синхронизации потоков — малая часть того, что можно использовать для атаки с захватом управления или отказом в обслуживании. В рубрике "exploits review" мыщъх планирует планомерно и систематично окучить эту плодородную тему, детально описывая детали технической реализации в разделе full disclose.