**Shellcode**

Table of Contents

[Execute of a Shellcode 2](#_Toc30597484)

[Type of shellcode 3](#_Toc30597485)

[Local shellcode 3](#_Toc30597486)

[Remote shellcode 3](#_Toc30597487)

[Connect back shellcode 3](#_Toc30597488)

[Bind shell shellcode 3](#_Toc30597489)

[Socket re-use shellcode 3](#_Toc30597490)

[Staged Shellcode 4](#_Toc30597491)

[Egg-hunt shellcode 4](#_Toc30597492)

[Omelet shellcode 4](#_Toc30597493)

[Download and execute shellcodes 4](#_Toc30597494)

[Encoding of shellcode 4](#_Toc30597495)

[Debugging a shellcode 6](#_Toc30597496)

[Create our first shellcode 7](#_Toc30597497)

[Simple sleep shellcode 7](#_Toc30597498)

[Shellcode goal 7](#_Toc30597499)

[Function needed 7](#_Toc30597500)

[1st step: dissembler 7](#_Toc30597501)

[2nd step: build up ASM 7](#_Toc30597502)

[3rd step: Compile 7](#_Toc30597503)

[4th step: Disassembler 7](#_Toc30597504)

[5th step: handling byte code 7](#_Toc30597505)

[6th step: try out shellcode 8](#_Toc30597506)

[Advanced shellcode 8](#_Toc30597507)

[Advanced shellcode in C++ 8](#_Toc30597508)

[Shellcode goals 8](#_Toc30597509)

[Function needed 8](#_Toc30597510)

[1st step: write shellcode in C++ 9](#_Toc30597511)

[2nd step: compile 9](#_Toc30597512)

[3rd step: inspect the program 9](#_Toc30597513)

[Advantage of this shellcode 9](#_Toc30597514)

[Analysis inspection 9](#_Toc30597515)

[4th step: build our shellcode in ASM 10](#_Toc30597516)

[5th step: try our shellcode in debugger simple C program 11](#_Toc30597517)

[Tips and tricks 11](#_Toc30597518)

[String terminators 11](#_Toc30597519)

[NULL-Free shellcode 12](#_Toc30597520)

[Manually edit 12](#_Toc30597521)

[Case: manual editing method 1 12](#_Toc30597522)

[Case: manual editing method 2 12](#_Toc30597523)

[Decode and Encode 13](#_Toc30597524)

[Msfvenom encoder tool 13](#_Toc30597525)

[Bad Characters 14](#_Toc30597526)

[Shellcode and payload generators 14](#_Toc30597527)

[Msfvenom 14](#_Toc30597528)

# Execute 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.

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 activities. 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.

# Type of shellcode

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

* Local shellcode.
* Remote shellcode.

## 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.

## Remote shellcode

is sent through the network a long with an exploit. The exploit will allow the shell code to be injected into the process and executed. Remote code execution is another name for this kind of exploitation.

- The goal of remote shellcode 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.

### Connect back shellcode

initiates a connection back to the attacker’s machine.

### Bind shell shellcode

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

### Socket re-use shellcode

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

# Staged Shellcode

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.

Tagged 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 into 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 the point, the execution of the bigger shellcode begins.

## Omelet shellcode

Is similar to the egg-hunter 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

We introduced the meaning of Null-free shellcode. Shellcodes are generally encoded since most vulnerabilities have some form of restriction over data which is being overflowed.

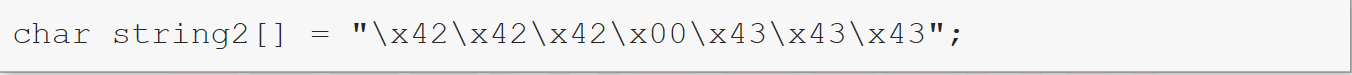
For example:



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

If everything works fine when the printf get executed the program should print the string”AAABBBCCC”

C language string functions work till a **Null**, or **0** byte is found. If the strings2 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 \42 and \x43



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

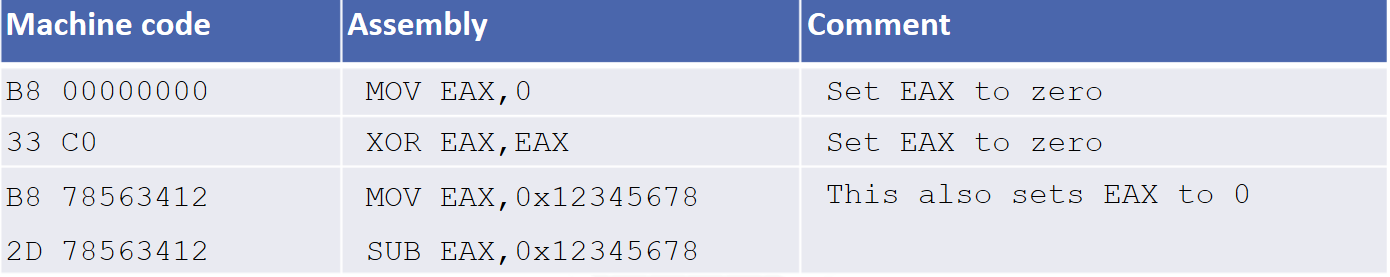
For our testing purposes, this is extremely important, if our shellcode contains a **NULL** character it will fail. **Shellcode should be free NULL-Free to guarantee the execution**. There are several types of shellcode encoding:

* Null-free encoding.
* Alphanumeric and printable encoding.

**Note:** 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**.

Let’s see an example. Let’s say you want to initialize a register to zero. We have different alternative.



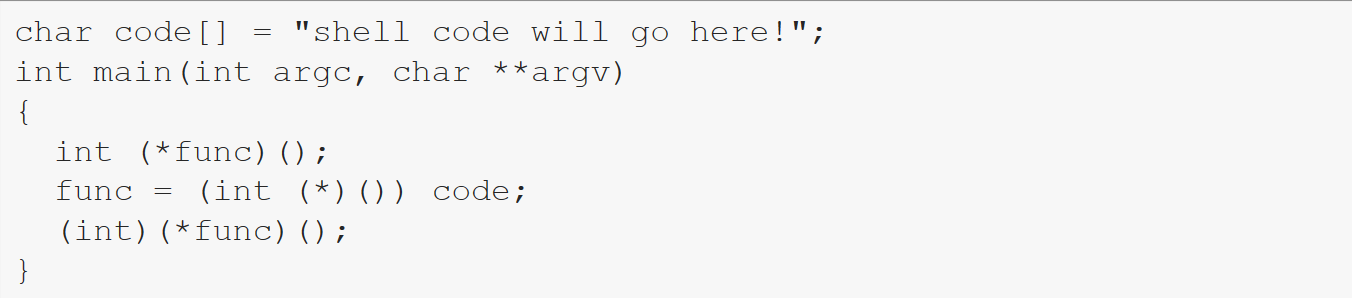
From this, you should notice that the first instruction (MOV EAX,0) should be avoided because it contain 00 within its machine code representation.

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

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

# Debugging a shellcode

Simple piece of code that will test to see if shellcode works as expected.



Shellcode to run the “windows calculator”



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.

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

# Create our first shellcode

There are many different tools and frameworks that we can use to generate shellcode 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.

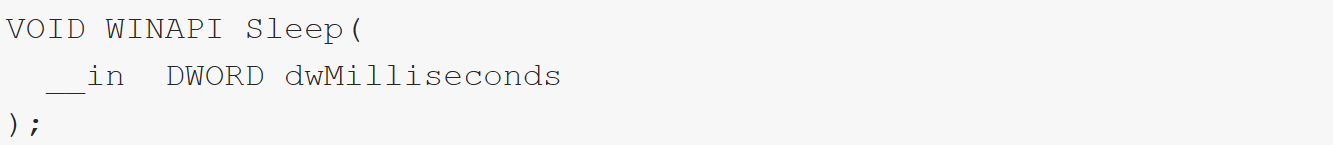
## Simple sleep shellcode

### 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:



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

#### 1st step: dissembler

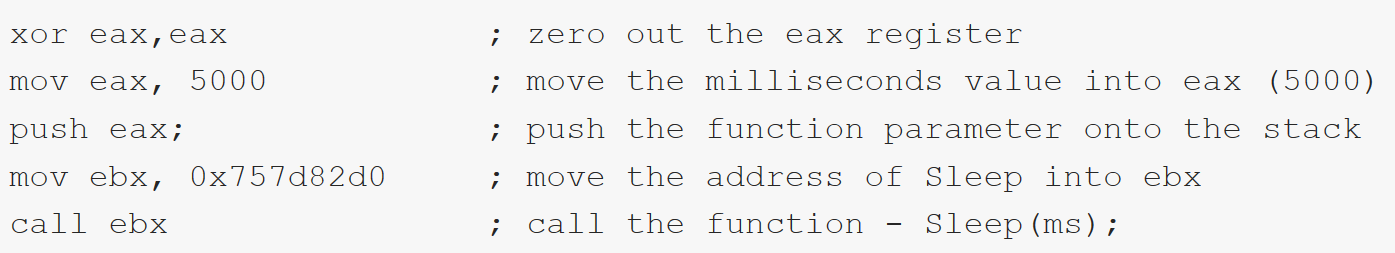
to obtain the address of the sleep function; this is required because we will create a small shellcode that calls this function.

We can find the address in different ways: by using immunity debugger or arwin tool.

#### 2nd step: build up ASM

as we know, when a function gets called, its parameters are pushed to the stack then call the function by using its address.

The ASM code:



**Note:** we can create many different versions of the same code. For example, we can push 5000 directly onto the stack, without the EAX register, and save one line of code.

#### 3rd step: Compile

we need to compile our ASM code.

#### 4th step: Disassembler

we have to disassemble it again.

#### 5th step: handling byte code

we have the byte code now, we need to edit it and remove the spaces and add the \x prefix.

#### 6th step: try out shellcode

finally, we can use our shellcode on debugger simple program to ensure it works fine.

**Note:** remember that not only different OS may have different addresses, if ASLR is enabled (like it is on our machine), the address is randomized. For those extra hard workers, an extra-mile would be to debug the program just created and see how it works.

# Advanced shellcode

Remember, 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 a study the function that does this.

You may want to use WinExec or ShellExecute to spwan the shell but you will need to set parameters for the function to work.

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

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

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

## Advanced shellcode in C++

### Shellcode goals

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.

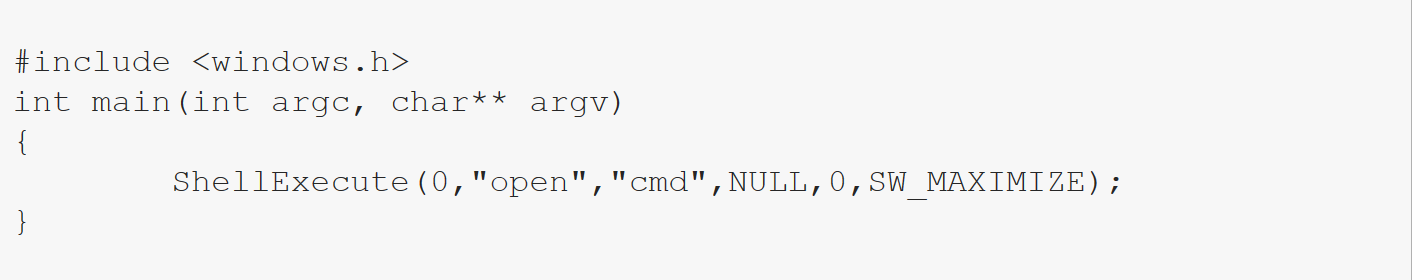
**Note:** remember that each compiler adds its own code inside.

The function we are going to use to spawn the command will be ShellExecute we can also use much simpler function such as WinExec.

### Function needed

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

#### 1st step: write shellcode in C++



#### 2nd step: compile

Once we have the source code ready, we just need to compile it. We also need to run the compiled program in order to see if it works.

#### 3rd step: inspect the program

Inspect the program with immunity debugger we should see something like this:



Once the main function starts, it sets the stack frame and 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…)

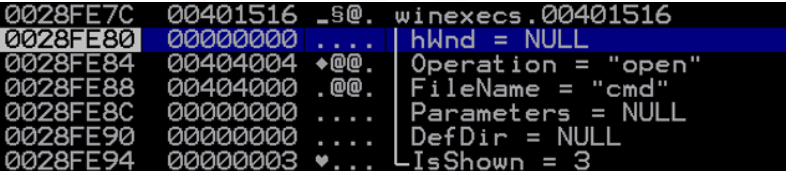
##### Advantage of this shellcode

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 as ‘cmd’ and ‘open’. Dealing with strings means that we have to:

* Calculate their hexadecimal value.
* Push the string.
* Push a pointer to the string into the stack.

##### Analysis inspection

1. The parameters are pushed in reverse order.
2. Representation of the stack right before calling the ShellExecuteA.



1. From dissemble panel we can also see that the module (.dll) that offers this function is in the shell32.dll file.



#### 4th step: build our shellcode in ASM

First thing to do is covert the strings (cmd and open) that we will push into the stack.

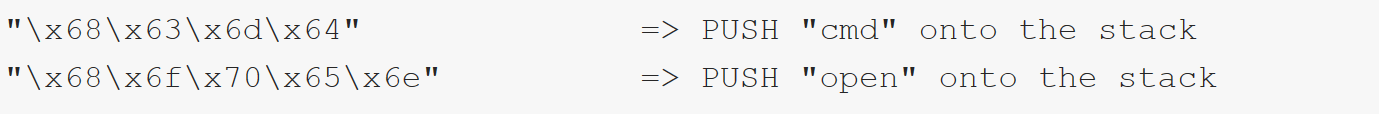
As in the compiled version of the program, these strings are taken from the .data section. As you imagine, this is something that we can not do while sending our shellcode (since the .data section will contain something different).

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).

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

* They must be exactly 4 bytes 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.
* Strings 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.

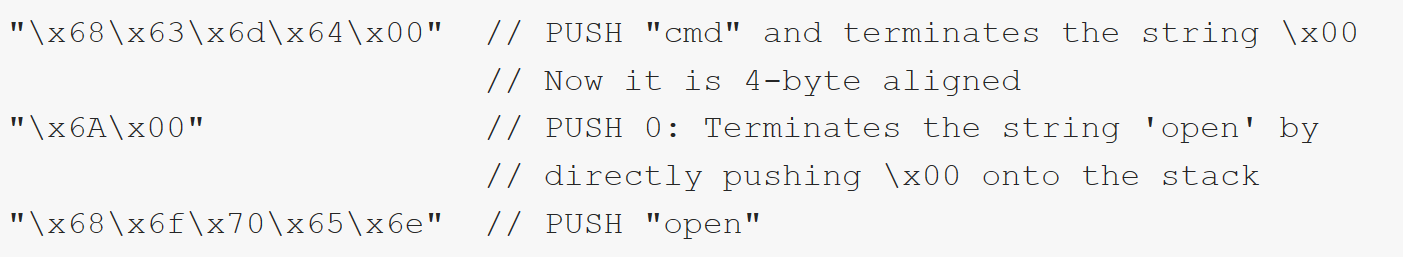
Convert cmd and open string with the following online tools: [asciitohex](https://www.asciitohex.com/), [rapidtables](https://www.rapidtables.com/convert/number/ascii-to-hex.html) and push hex in reverse order.



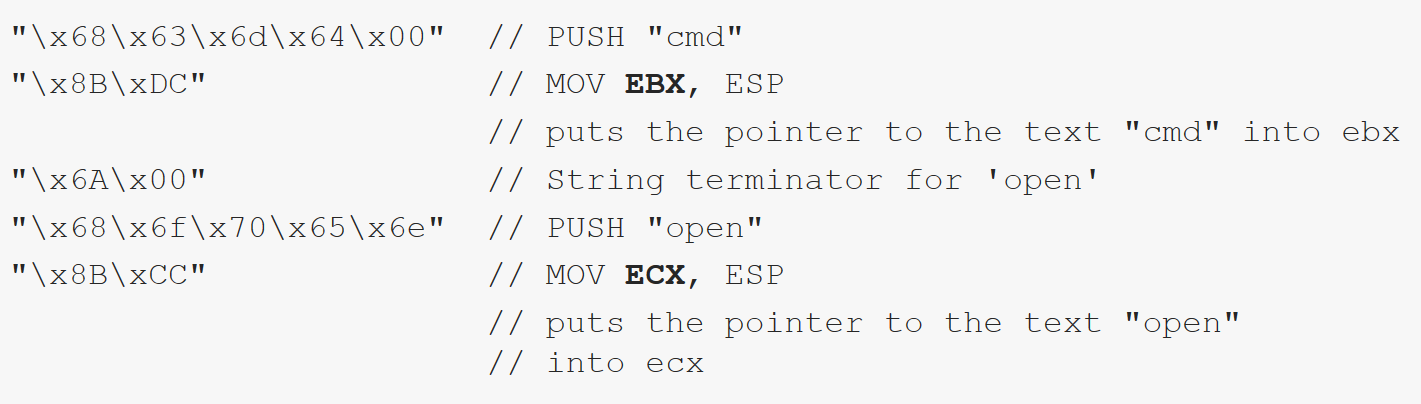
Note: \x68 is value of PUSH instruction.

The first PUSH open is not 4 bytes aligned and there is not the string terminator at the end. The second PUSH is 4 bytes, but we have to terminate the string.

Therefore, we will edit shellcode as following:

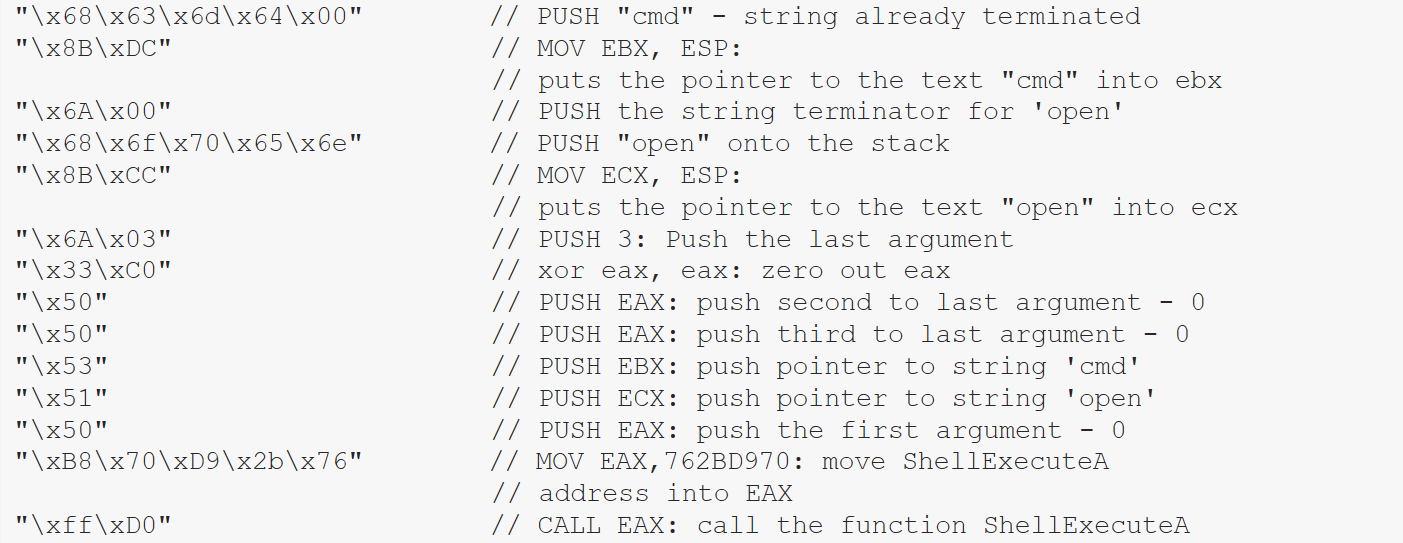


After pushing the strings to the stack, we will save the current stack pointer into a register. When we push the strings, **ESP** will be aligned to the top of the stack. Hence it will point to the string itself. Strong this value in a register such as **EBX** or **ECX** allow us to save a pointer to that string. Then we will just have to pass the pointer as an argument of the function. Let’s update our shellcode.



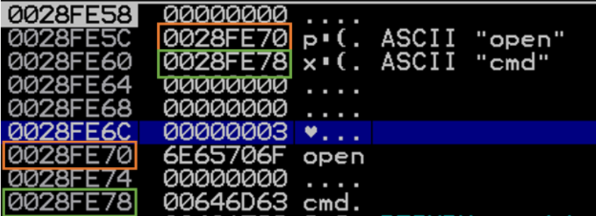
We still need to pass four other parameters to the function: three of them are 0 while one is 3.

Lets now combine the whole shellcode



#### 5th step: try our shellcode in debugger simple C program

* Stack frame right before call ShellExecutionA function.



# Tips and tricks

This tips and tricks help you to build you shellcode with effective and efficient way.

## String terminators

Are important makers 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 the mas punctuation marks like a “.”, or “;”

**Note:** if Null-byte \x00 is not exists, it stops as soon as it encounters the first \x00 value in the stack. The same things happen to the operation parameters.

### NULL-Free shellcode

If the shellcode is not free of Null-byte it will fail for example: in case of strcpy function, the function will copy only first 4 bytes of shellcode \x68\x63\x6d\x64.

There are two main techniques that we can use:

#### Manually edit

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.

##### Case: manual editing method 1

###### Goal

Push the bytecodes 00466d63 (Hex: \x68\63\x6d\x46\x00) without string terminator.

###### Solution

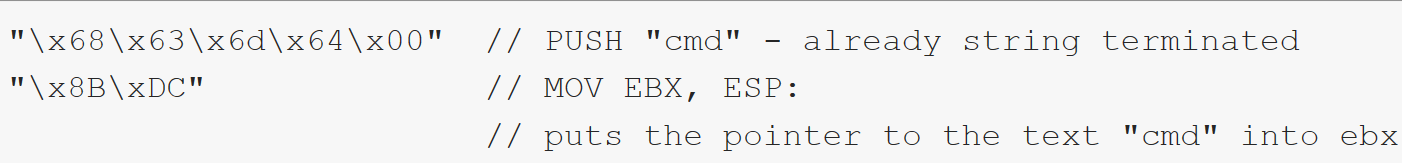
Subtract or (add) a specific value in order to remove 00

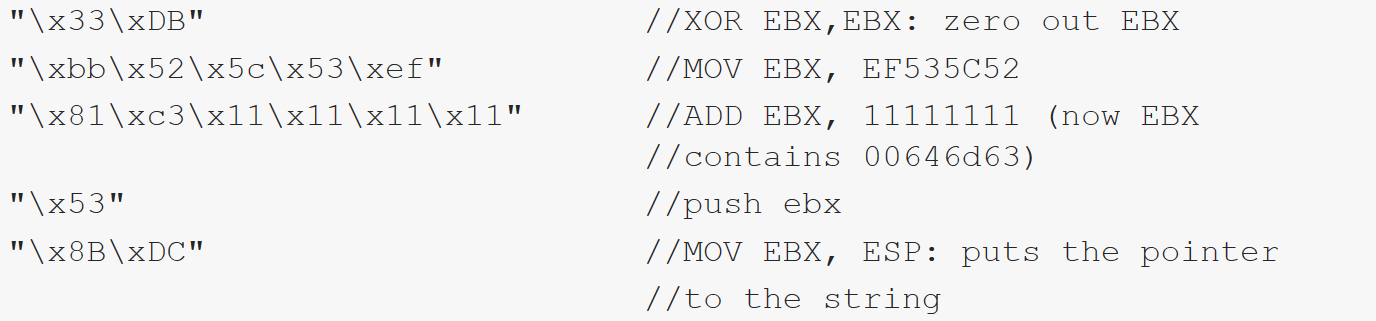
For example, let’s we subtract 11111111 from 00466d63 we will obtain EF535C52, which does not contain the string terminator.

**Note:** 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. Move EF53C52 into a register.
2. Add back 11111111 to the register (in order to obtain 00466d63).
3. Push the value of the register to the stack.

[shellcode contains Null-byte]  


[Shellcode: after null-free applied]  


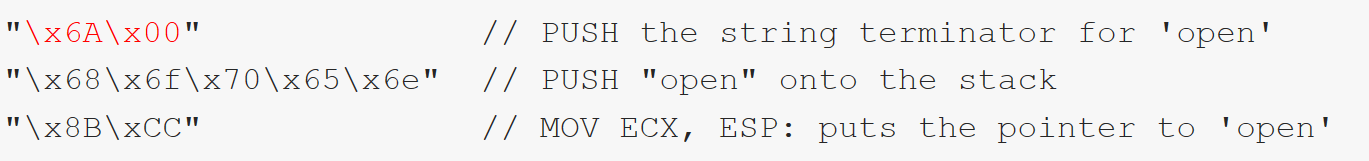
##### Case: manual editing method 2

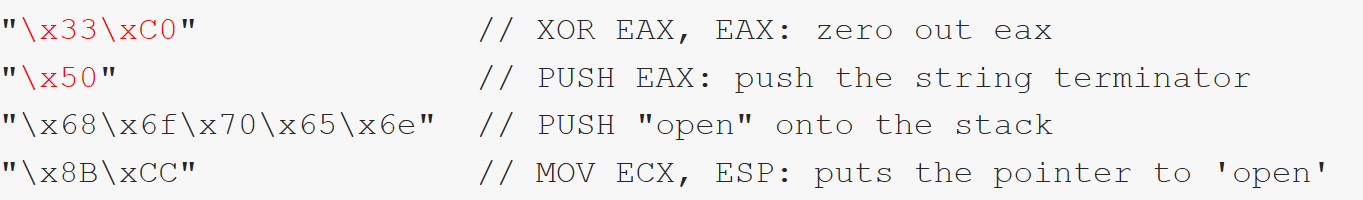
###### Goal

Delete string terminator for the string ‘open’.

###### Solution

This time is much easier, we can zero out the EAX register and push then its value into the stack.

[shellcode contains Null-byte]  


[Shellcode: after null-free applied]  


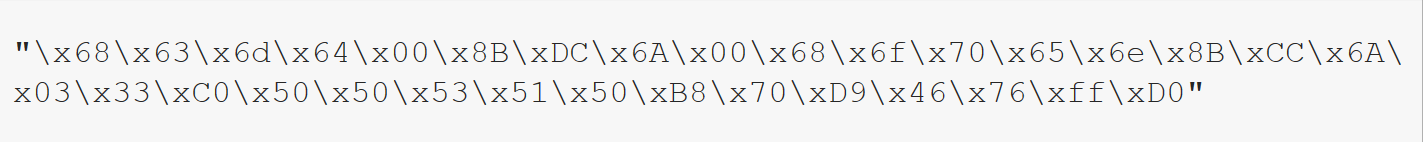
#### Decode and Encode

We can also encode and decode the shellcode.

##### Msfvenom encoder tool

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

###### Case: encoding custom payload

[Shellcode contain Null-bytes]  


Problem

Shellcode contains the null bytes \x00

Solution

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

* Command: echo -ne “\x86\x63\x6d\x64\x00….” > binshellcode.bin
  + -n: is used to not output the trailing newline.
  + -e: enable interpretation of backslash escapes.

Then using msfvenom to encode the binary file.

* Command: cat binshellcode.bin | msfvenom -p - -a -x86 –platform win -e x86/shikata)ga)nai -f c -b ‘\x00’
  + -b: is used to specify a list of bad characters to avid when generating the shellcode.
  + -a: specifies the architecture to use.
  + -p -: instructs msfvenom to read the custom payload from stdin.
  + –platform: is used to specify the platform.
  + -e: specifies the encoder to use.
  + -f: set the output format (in our case C).

## Bad Characters

We are able to determine the bad character for out vulnerable program to be a Null-byte or \x00, there are many cases where there are often more thane one bad character that we can’t use when developing our exploit. We may need to account for the “newline” \n or \x0A character fir instance. New line characters should be considered when developing your exploits.

# Shellcode and payload generators

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.

## Msfvenom

we can list all the available payloads. Each payload targets a specific OS or platform and has its own featues. For example, we have bind and reverse payloads, staged and stageless payloads and much more.

**Arwin commands:**

arwin.exe kernel32.dll Sleep [Enter].

**Immunity debugger:**

Right-click on the dissemble panel > search for > name in all modules > search for sleep > keep note of .txt Export Sleep address.

Converter tools: [asciitohex](https://www.asciitohex.com/), [rapidtables](https://www.rapidtables.com/convert/number/ascii-to-hex.html)

Encoder tool: msfvenom

* echo -ne “\x86\x63\x6d\x64\x00….” > binshellcode.bin
* cat binshellcode.bin | msfvenom -p - -a -x86 –platform win -e x86/shikata)ga)nai -f c -b ‘\x00’

Shellcode and payload generators: msfvenom, the backdoor factory and veil-framework.

Msfvenom

Command: msfvenom –list payloads

1st step: Command: msfvenom -p windows/messagebox --payload-options

2nd step: command: msfvenom -p windows/messagebox TEXT=”First msfvenom shellcode” -f c -a x86 –platform win -b ‘\x00’