REVERSING and EXPLOITING USING FREE TOOLS (4th PART)

We have seen how to analyze the first two exercises, called stacks, which are the simplest ones using the three interactive disassemblers, IDA FREE, RADARE and GHIDRA.

In this part we will continue analyzing the exercises called stacks 3 and 4 but before that we need to introduce a new concept, that of invalid or bad chars.

## Bad char definition

*The badchars or invalid characters are characters received and filtered by the target program; they serve as delimiters or through the internal algorithms of the program, it eliminates or replaces them with other values, making our shellcode useless.*

*The search for badchars is a crucial part in the methodology of developing exploits, since, if these are not located and avoided during the generation of the payload, they make it useless because it would not be interpreted correctly in the target system.*

Explaining the previous definition, the key point is data that we send to a program, it processes that and in the process until arriving at the point of exploitation, it can happen that the program checks, filters, substitutes or blocks certain characters which prevents that we use them in our payload. Obviously, this complicates the exploitation and it will have to be considered at the moment of creating the exploit.

There are many ways to check if in the exploitation of a program, it produces invalid characters. One is analyzing the program from where they enter our data to the point of exploitation and see what it does with them. This method is the most accurate but sometimes it is extremely tedious if the program performs multiple checks, copies and manipulations of the entered data.

Another option is to pass a string containing all the possible characters and check what happens with it till, in several attempts, we’ll generate a string that passes complete and is fully copied to the destination buffer, by removing the invalid characters

Normally, in a first attempt, we avoid passing the 0xA, 0xD, 0x0 and 0xFF characters that are the main invalid characters. If the rest passes well and arrives at the zone of exploitation without problems these four can be added, one by one to see if there are no problems with them.

In the case of a buffer overflow, we could pass the string of test characters and see if it is copied to the buffer and if arrived all the ones sent.

A code in python 3 to set up a payload to test the invalid characters could be:

## BASIC SCRIPT TO TEST BAD CHARS

*payload = b""*

*excluded = (0x0, 0xa, 0xd, 0xff)*

*for i in range(256):*

*if i not in excluded:*

*payload += bytes([i])*

*print (payload)*

We can see that in excluded there are the invalid characters, which we will remove or add more, depending on the tests we are doing.

Here I made a modification to the stack 1 script to test the invalid characters, those that we find will be valid for the 4 exercises since they all process data in the same way, entering by means of the function gets, which copies them to the destination buffer buf

## ADAPTING BASIC SCRIPT TO TEST BAD CHARS IN STACK 1

*import sys*

*from subprocess import Popen, PIPE*

*payload = b""*

*excluded = (0x0, 0xa, 0xd, 0xff)*

*for i in range(256):*

*if i not in excluded:*

*payload += bytes([i])*

*print (payload)*

*p1 = Popen(r"STACK1\_VS\_2017.exe", stdin=PIPE)*

*print ("PID: %s" % hex(p1.pid))*

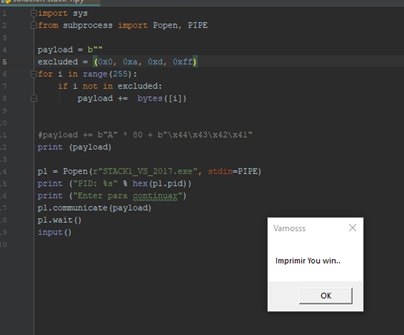
*print ("Enter para continuar")*

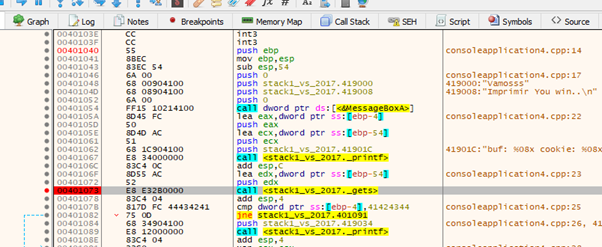
*p1.communicate(payload)*

*p1.wait()*

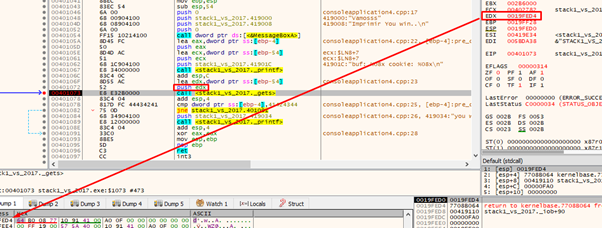
*input()*

I’ll execute this script attaching it to x64dbg when the MessageBox pops up and going till the gets() function. I’ll execute it and when I return from it, I’ll check if it copies all the characters to buf that is the destination buffer that is going to be overflowed.



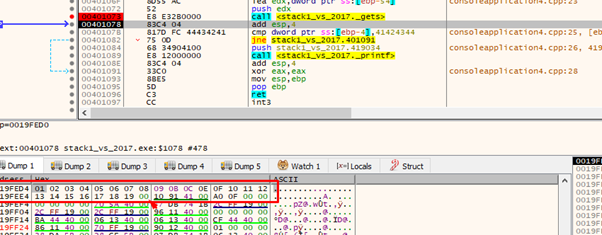


I put a breakpoint in the gets()and I accept the MessageBox, it will stop there.



As I see that EDX contains the address of buf, I select the EDX register and choose FOLLOW IN DUMP to see in the dump the area where it will copy the bytes that I send.

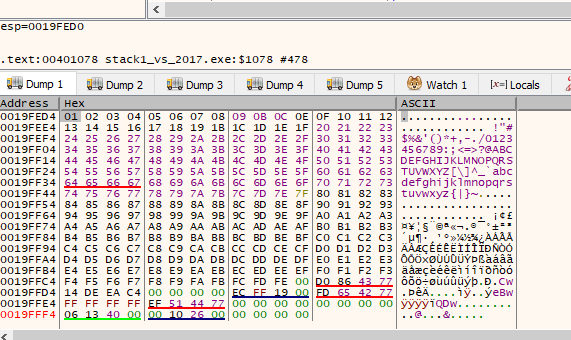
Now I press f8 to execute the gets().



I see that my string was cut at 0x1a character, let's add it to the list of invalid characters and try again

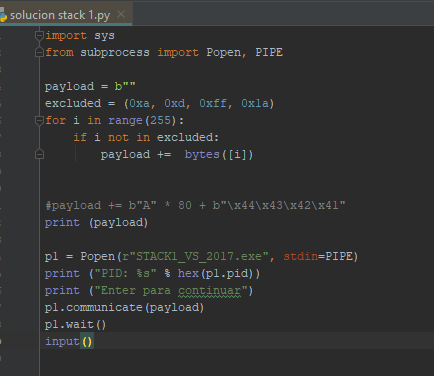
*excluded = (0x0, 0xa, 0xd, 0xff, 0x1a)*

I repeat the process: launch it, attach the program and step above the gets() with f8

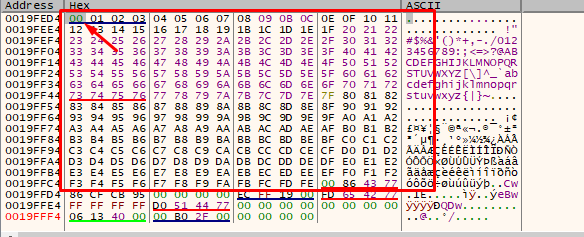


We already see that it entered from 0x1 to 0xfe. This way we already know that avoiding the 5 characters that we put in the list of invalids will pass our payload.

Now we will remove the initial invalid characters one by one, let's start with 0x0.

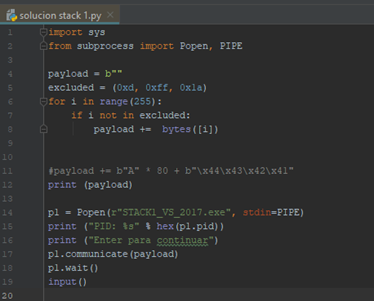


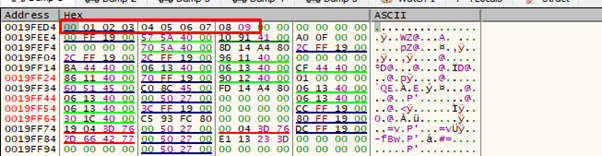
Let’s try again!



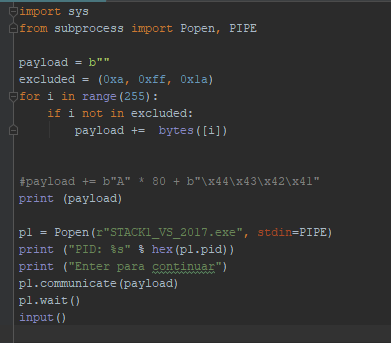
We can see that 0x0 is not an invalid character since it entered and didn't cut the payload, let's continue with the others.

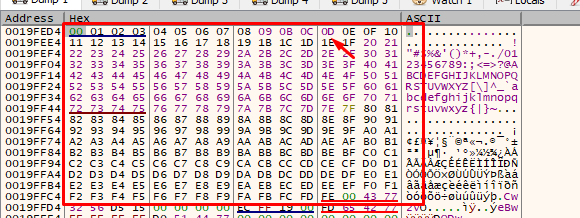
We remove the 0xa.



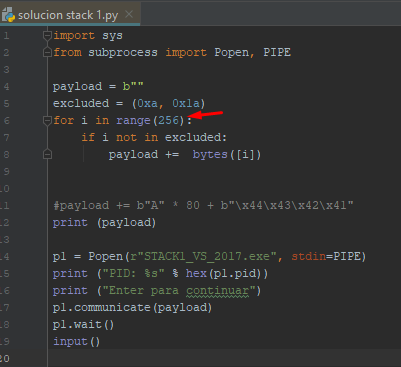


We see that the 0xa is an invalid character since it cuts the payload and does not enter complete, I add it again and remove the 0xd.

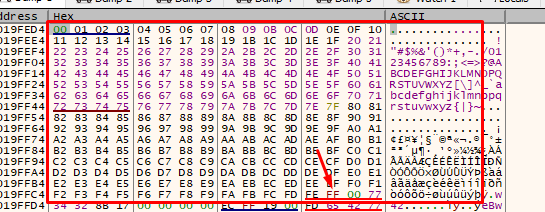




Now we see that the 0xd is not invalid character as it did not cut the payload. There is only 0xff left, I’ll try.



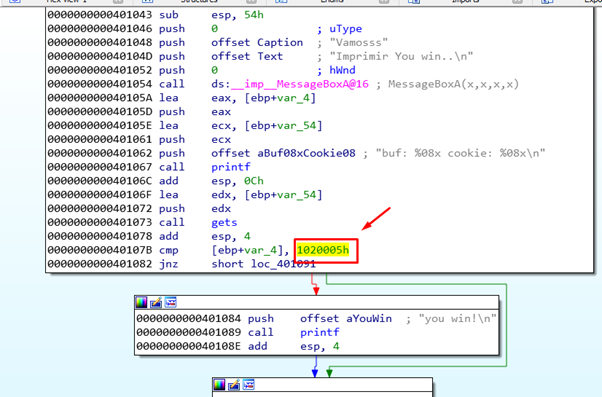
I see that in the loop, to enter the character 0xFF it must loop up to range(256) because it does not include the last one, so enters the 255 (0xFF)



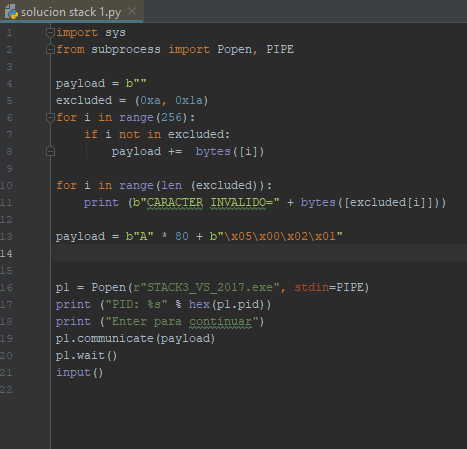
We see that already entered the whole string and we have tested all the characters, the invalids for the stacks are 0x1a and 0xa, the rest can be sent without problems.

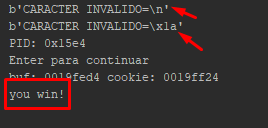
That's why stack 3, which in the cookie has a zero, is made for one to investigate if the zero is an invalid character. As it is not and I can send it in my payload, the script will be like stack 1 and 2 only changing the value so that it compares with the cookie

## STACK 3 SOLUTION



So, you must pass "05\x00\x02\x01" to compare with the cookie. No one is an invalid character so there is no problem.

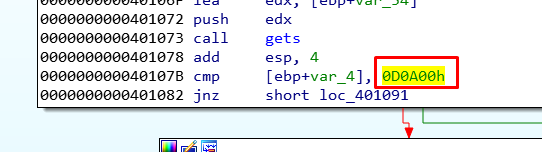




There we print the invalid characters and we see that with the same analysis that we did for stack 1 and 2, we can solve number 3 without problems.

It is convenient whenever we must perform an exploit, to do the job of detecting invalid characters to be sure that our payload will not be affected by them.

The stack 4 case is different as there is an invalid character in the cookie.



We see that we should pass over him to step on the cookie.

**payload = b"A" \* 80 + b"\x00\x0a\x0d\x00"**

We can see that our payload contains an invalid character, so we cannot pass the value we need to step on the cookie and take us to “YOU WIN”.

In the next part we will show the solutions.

If you want to try it you should know that you must continue to step below the cookie and try to step on the return address, forcefully returning to the address where the program calls the printf with the message “YOU WIN", or look for some way to achieve print what we want.

Of course, it is quite possible the program will break when you do that, so there are two valid solutions:

1) To print “YOU WIN” and breaks which is the simplest one.

2) To print “YOU WIN” and to avoid it breaks which implies much more difficulty, obviously

Let's see who is encouraged to do it either way (or both!!).

See you in 5th part

Ricardo Narvaja

Translated by @arrizen

11/15/2019