Reversing & Exploiting with free tools (part 2)

In the first part we have installed several tools that will be useful to complete this course, the main point is they are all free, we will not use any paid tool, and for those with paid versions like IDA or PYCHARM we will use the version FREE or COMMUNITY versions.

Let's look at some concepts before going into the exercises

Bug definition:

A bug is the result of a failure or deficiency during the process of creating computer programs (software). Such a failure can occur at any stage of the software's life cycle, although the most obvious ones occur at the development and programming stage.

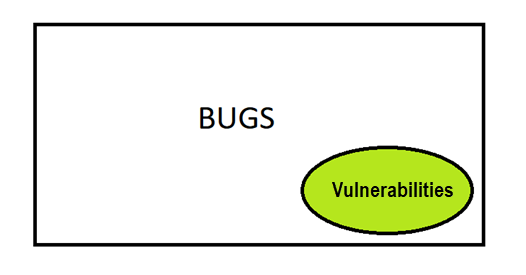
I always say the programmer can make mistakes and those errors can produce bugs (failures) in the program, this is a familiar thing.

The issue is to know the difference between a BUG and a VULNERABILITY so let's see what a VULNERABILITY is.

**Vulnerability definition:** is a certain type of bug in a piece of software that allows, through its exploitation, to compromise the security of a computer system

Vulnerabilities allow to perform actions for which the software was not intended and abuse them.

So, vulnerabilities are one type of bugs, a subset of them.



Of course, there are many types of vulnerabilities, but we are going to focus on studying Windows exploits, and their exploitation techniques.

**Exploit definition**: is software code that tries to use a vulnerability of another software.

The goal of the exploit could be malicious as destroying or disable the attacked system.

although it is normally a matter of violating security measures in order to be able to access information in an unauthorized manner and use it for one self’s benefit or as a source of other attacks on third parties.

* Abusing a vulnerability can allow from crashing an application or even the whole system, to executing its own code on local or remote machines and its exploitation and difficulty varies depending on the same vulnerability, the environment and the mitigations that the target has at the moment of exploitation.

The first type of vulnerability we are going to study will be the buffer overflow, we will begin with the simplest examples and then increasing knowledge step by step. For this reason, at the beginning we’ll not have all the mitigations or system protections activated and bit by bit we’ll be activating them to learn how we can handle ourselves in those situations.

* **Buffer definition:** a memory space of a certain size reserved for storing and managing data

A simple example can be that of a 20 liters tank that is initially empty, there you can store up to 20 liters, but no more. if you want to store more in a single tank you should find a way to have a larger buffer. Otherwise, when trying to store, for example, 40 liters in that 20-liter tank the liquid will overflow.

**· Buffer Overflow definition:** buffer overflow occurs when a computer program exceeds the amount of memory reserved for it by writing to the contiguous memory block.

* A buffer overflow occurs in a computer application when it doesn’t have the necessary security checks in its programming code, such as measuring the amount of data that will be copied into a buffer ensuring that it does not exceed its size.
* The most common types of buffer overflows are stack buffer overflows and heap buffer overflows.

Well, here we have seen the definition of buffer overflow, and in our previous example if I try to save 40 liters in a tank of 20 liters it will overflow as we saw. This overflowing is the *buffer overflow*, or the overflow of my tank to exceed the maximum capacity of it.

Now an explanation about the difference between **stack** and **heap**

**STACK**

The stack is used to store local variables of a function that are only needed while the function is executing. In most programming languages it is fundamental that we know in compilation time a variable size if we want to store it in the stack.

**HEAP**

The heap is used to reserve dynamic memory, whose useful life is not known in advance, but is expected to last a while. If we do not know its size or the same is decided in runtime it should be calculated and reserved in the heap.

It is also used for objects that vary in size, because we do not know at compile time how long they will last or cover.

In our company I have worked for more than 13 years as exploit writer, and the first thing we do all the people who are hired, even myself when I joined, is to try to solve the famous Gerardo Richarte’s *stacks* and *abos*, one of the founders of Core Security and one exploit analyzing guru.

We will start step by step with the stacks that are the simplest, of course as I said they are compiled for now with minimal protection to make it easier to start exploitation.

Let's see the source code of stack1.

<https://drive.google.com/open?id=16btJAetpa1V5yHDZE2bnnFWTQNpsUR4H>

Here we have the exercises folder and inside it is the source code of the stack1 called STACK1\_VS\_2017.cpp.

---------------------------------------------------------------------------------------------------------------------

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_DEPRECATE

#include <stdlib.h>

#include  <stdio.h>

#include "Windows.h"

**int main(int argc, char \*\*argv) {**

MessageBoxA((HWND)-0, (LPCSTR) "print You win...\n", (LPCSTR)"Great!", (UINT)0);

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

**printf("you win!\n");**

}

--------------------------------------------------------------------------------------------------------------------

Let's try to understand this code and see where the buffer overflow can be produced, and if it will be an overflow buffer in the stack or in the heap.

To the code of the original stack1 it has been added a call to MessageBoxA to show us a text that encourages us to solve it. It is just an addition that does not influence anything, is a standard call to Windows function that we will not analyze here.

We know that within a function, if there are local variables you must reserve space for them before starting with the instructions themselves.

So we are left with this which is the original code created by Gera.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

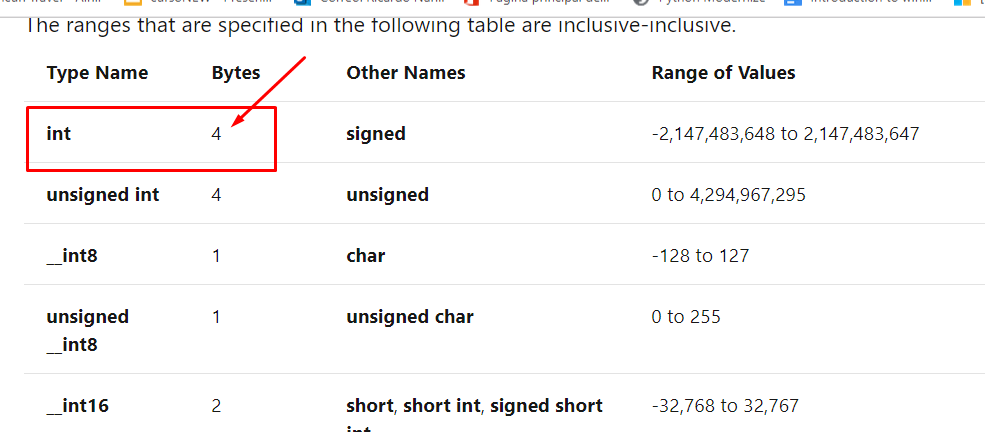
**printf("you win!\n");**

}

We can see in red the first part where it reserves the space for local variables, in this case there are two local variables, *cookie* and *buf*.

You can find its sizes, in this table.

<https://docs.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=vs-2019>



In the case of *buf*, we see that it is an array of char or string so as a char it has one-byte size.

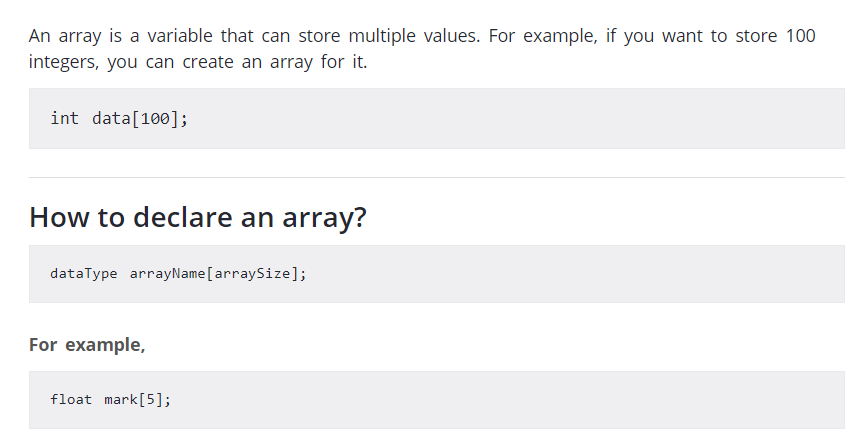


It will be an array with 80 chars. It will be 80 bytes length.

If you don't know what an array is, you can look at this data

<https://www.programiz.com/c-programming/c-arrays>

Summarizing an array can store many values of the same type of data, just tell it what type the data is and how many will be.



First example is an array of int 100 ints and as one int occupies 4 bytes the total length would be 100 x 4= 400 bytes

In the second example one float occupies 4 bytes, so it would be an array of 5 floats. The final size would be 5 x 4 = 20 bytes.

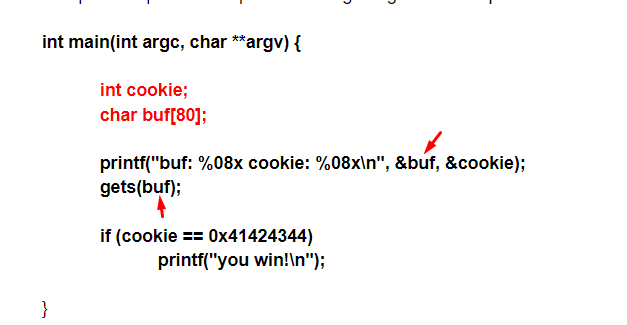
When we analyze an array at a low level, we will see that it is a reserved memory space or buffer, it is not the only way to reserve space. There are other types of data variables that also require to reserve a space in the memory that will be buffers to keep its content.

Returning to our exercise

**char buf[80];**

It is a char array of 80 x 1 = 80 bytes long, so this is like our tank of 20 liters, if we try to keep more than 80 bytes it will overflow.

Next, we’ll discuss how the buffer or variable called buf (indicated below with red arrows) is used.



We see that it is used in the two places marked with red arrows.

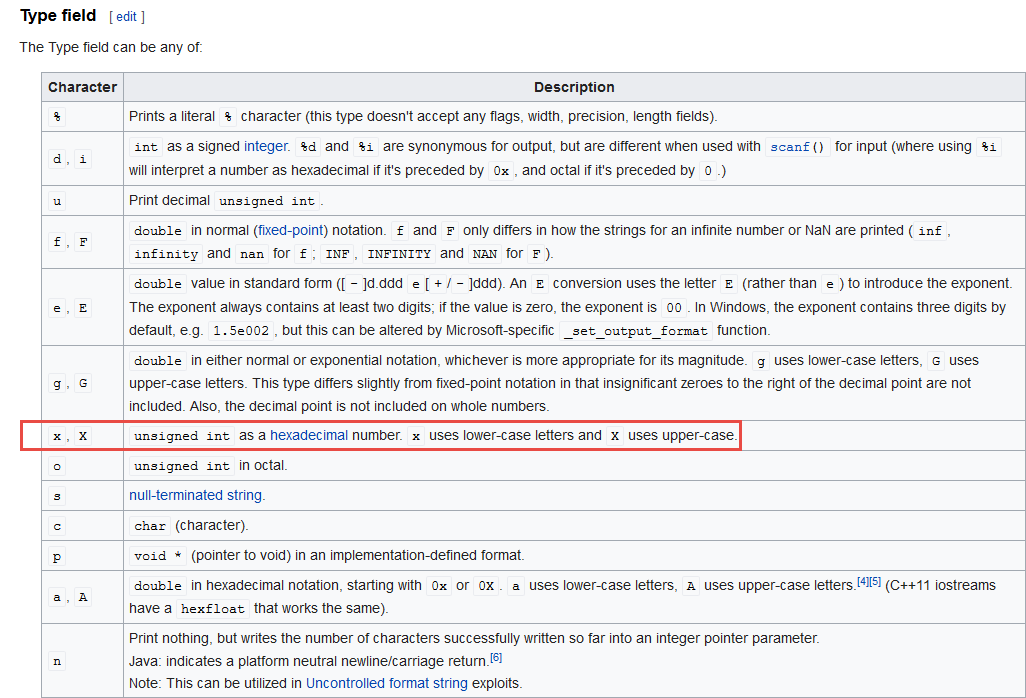
In the first instruction there is a printf that is used to show the string between the quotes as a console message

**"buf: %08x cookie: %08x\n"**

But printf not only prints the quoted string but also prints with format, the percentages chars inside tell us that it will form an output string. As we can see the string is just the first argument of the function and is the output format; there can be several other arguments (there will be one for each % that is in format) in this case two.

https://lh4.googleusercontent.com/DCB7q78lCEe4_F6tXpHQefdbKy2fBy7KjxlmvsuSn-e2JO8jMaSBiuyGer5KN2HV-tetqNVI13w8nilaRUZWow15egXYvNhmsDVUUAyGjIeYcGBiGknL1VubXnRfrwH0AdelAAON

in this case they are two %x therefore if I check the table of formats for printf.



We see that it will take those integers (int) and insert them into the output string in radix 16 (i.e. in hexadecimal) The 08 refers to the fact that if the number has less than 8 digits it will be filled with zeroes.[[1]](#footnote-0)

Output for **"buf: %31x”,%buf**

**buf:             19fed4**

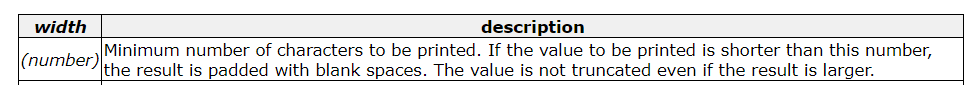
In this example it is filled with spaces before the number, there are several modifiers to change the output.

All the possibilities are listed here:

[**http://www.cplusplus.com/reference/cstdio/printf/**](http://www.cplusplus.com/reference/cstdio/printf/)

Our case was this one

https://lh3.googleusercontent.com/_9J1dhLm4L_pW1M5DjnvKCUSHOOIzk6XfO0aaUum8zmThRA9iMFirDrs3aCGkcSqSygTnZ5vkrf06FwbwMb4YZ1mCL3Zi0CYxgwdVKxTIhvuI9UhBRtIfgBk24jOVZQ0K5p6dfgy



We see that the result is not truncated, it only fills in with spaces if the length of the argument to insert is less than the value in front of the x.

Therefore, we know that it prints two hexadecimal numbers that come from the two arguments.

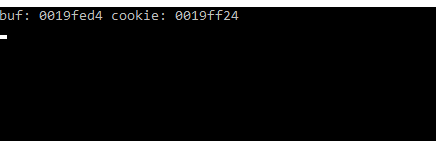
**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

We know that a variable has a memory address and a value to store, it is like our 20-liter tank, it has its contents or value, which are the amount of liters it stores inside. But also, if I have a garage full of similar tanks, I need to have some way of knowing where the tank I want is located out of all the tanks I own.

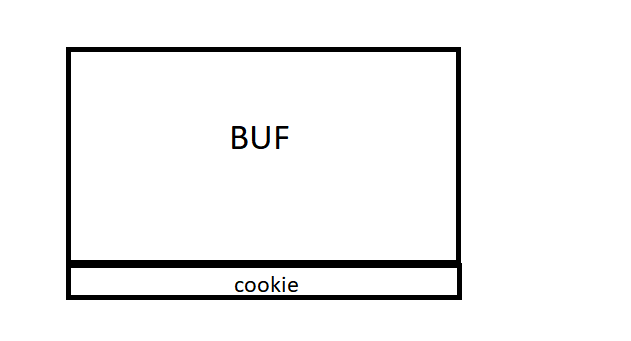
This is indicated by the & ampersand symbol, which tells us the direction or location of the tank, not its contents or value.

Ampersand definition: & ampersand is used to indicate the memory address of the variable where the data will be stored.

So, if I run the executable on a console I'll see for example that the printf prints.



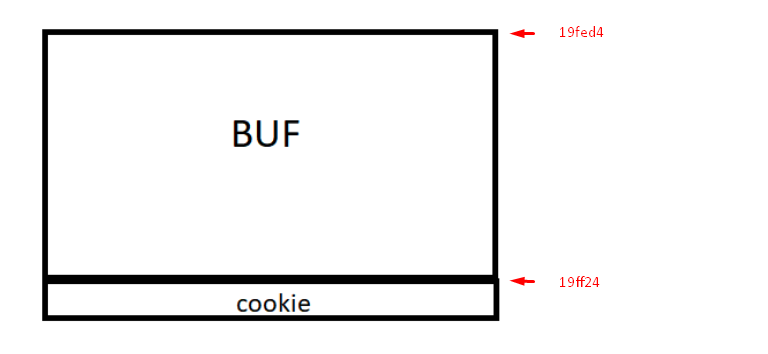
In your machines the directions can change but as the lower direction of both is the one of *buf* we can see that they are located this way.



Debuggers and disassemblers represent the lowest directions up, growing down.

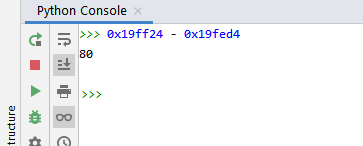
The address of buf is smaller than the address of the cookie, so we represent it above in the image.

And what do the addresses of both variables tell us (in my case they were &buf=0x19fed4 and &cookie= 0x19ff24)?



Both are represented in hexadecimal (let's remember that their format was %x) therefore I place the 0x in front of them to differentiate them from the decimal numbers that we will represent them with no added.

If I do the subtraction in a Python console or in Pycharm itself



The result of the buffer size is 80 because *cookie* starts right where the buffer ends, so the difference gives us the buffer size.

Often, when we do this type of accounts based on the source code it can happen that it gives us a bigger size than the one reserved in the original code. The compiler ensures that it will reserve at least 80 bytes; it can reserve more, never less.

The point is that we already know some things about the code, the sizes of the variables and their location thanks to the printf

Now let's see the other place where the variable *buf* is used because for now it only prints its address but has not been used for saving anything in it.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

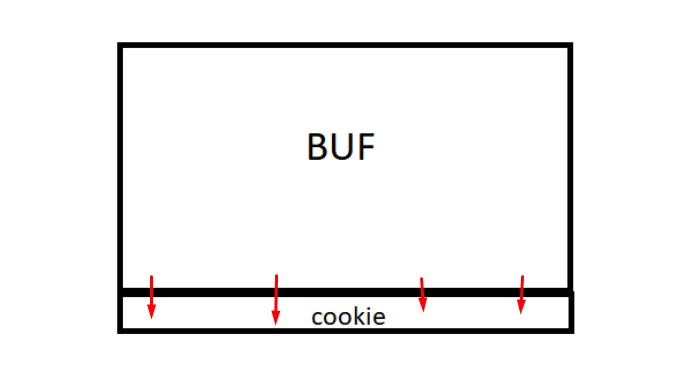
**printf("you win!\n");**

}

The *gets* red instruction is a keypad entry function, which will enter the amount that the user wants, until he presses the enter key.

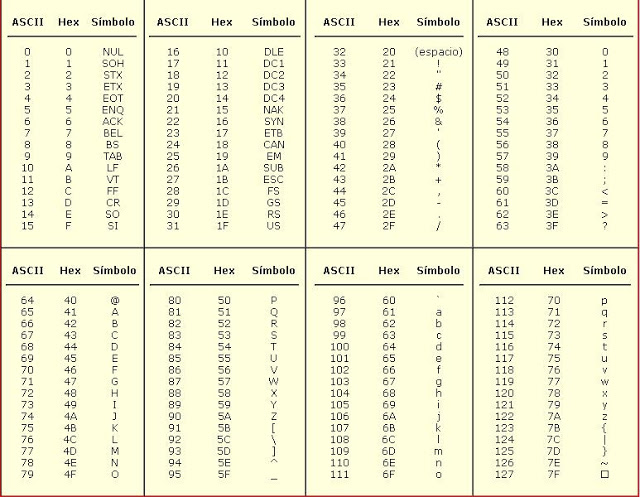
There is no way for the program to limit the amount of data entered by the user, nor is there any way to check the data, as it is entered it is copied to the buffer *buf*.

This has a problem. We said that *buf* can only store 80 bytes maximum, so if we enter more we will produce a buffer overflow, and here are given all the conditions for it because if the user writes more than 80 bytes, will overflow our tank and begins to drip liquid down. :-)



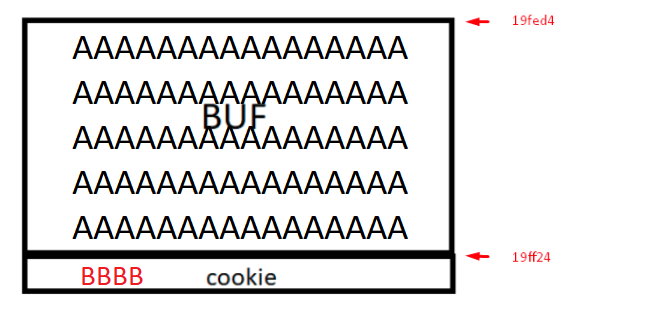
The point is that under *buf* is the *cookie* variable so whatever overflows will step on it and fill it with a value that the user can control.

For example, if the user writes 80 A letters and 4 Bs the 80 As will fill *buf* and the 4 Bs will fill *cookie*, and as we know when one types a character in the console, at low level it will be saved its ASCII value.



As *cookie* will have been filled with four letters B that are equivalent to the value 0x42, we can assure that the value of *cookie* will be 0x42424242 or in my machine the address of *cookie* 0x19ff24 will have as content 0x42424242.

0x19ff24 **42424242**



We've already seen how to overflow and control the *cookie* value.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

**printf("you win!\n");**

}

The point is that to succeed in the exercise you have to print "you win" and to achieve that *cookie* must be equal to 0x41424344. If there was no overflow it would not be possible because the value of *cookie* is never changed in the program code; if there is no overflow, we cannot get the "you win" message and here is complemented with the definition of buffer overflow, that I said that it can make the program perform actions different from which it was programmed.

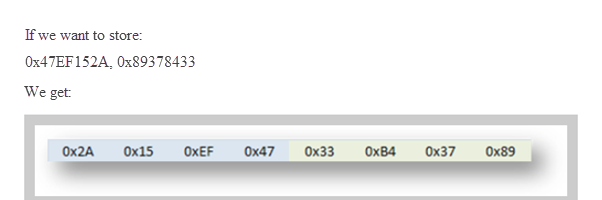
In this case it would never be possible to print “you win”, only the overflow will allow to do it.

So instead of passing for example 80 As and 4 Bs to print "you win" you should pass 80 As and DCBA because that will save the cookie values ASCII

44 43 42 41

<https://en.wikipedia.org/wiki/Endianness>[[2]](#footnote-1)

Here you can see the format in which the data is stored, in this case little endian i.e. in the memory they are stored backwards, haha, to say it in a simple way.

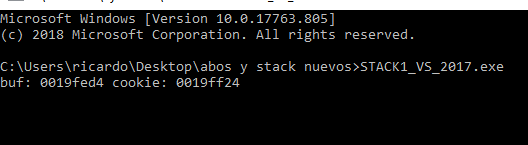


And if 0x41424344 is stored it will be stored in memory as

44 43 42 41

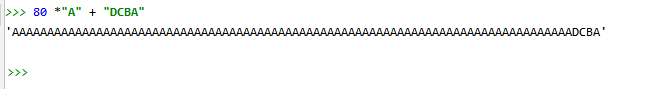
For that reason, when copying in memory, it will do it byte by byte, just as we type, we must write the value backwards so that when reading it from memory it does it correctly.

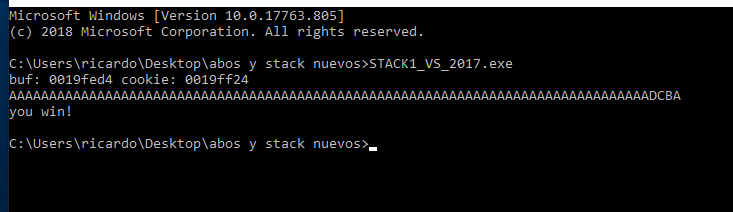
We can run the executable in a console.



And the cursor is flashing because the gets asks me to type the input data, I’ll type carefully 80 aes and then DCBA.

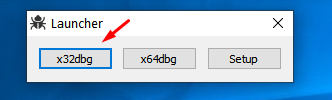
I can print the string in a Python or Pycharm console, copy it without quotes and paste it in the console to not type like crazy, then press ENTER to enter.



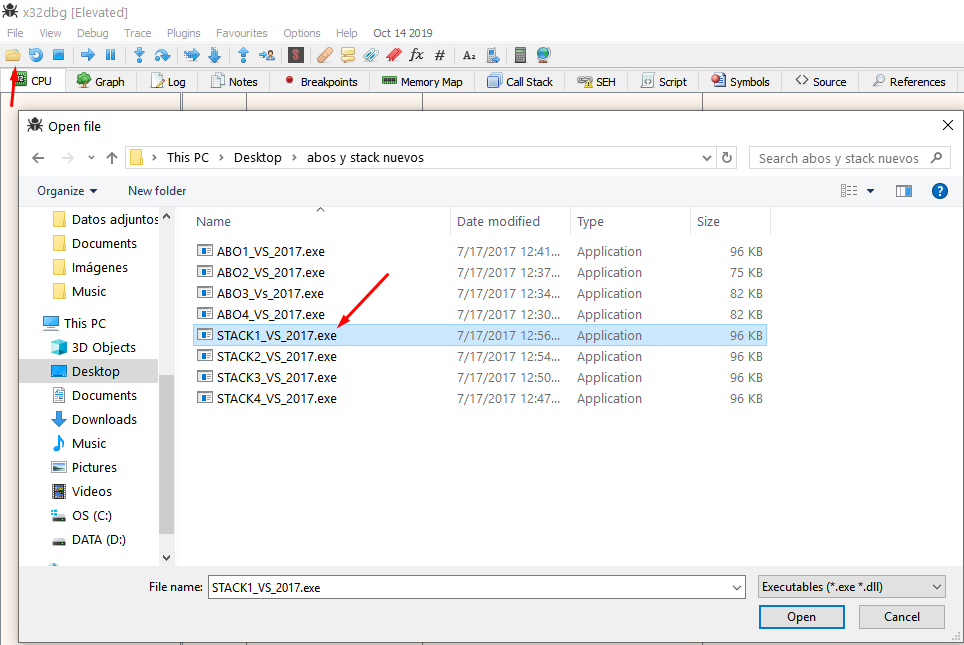


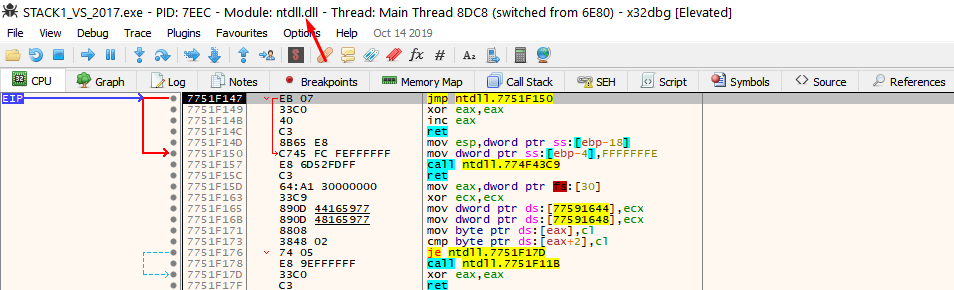
We see that we achieved, "YOU WIN", hehehe.

We can see this in a debugger, we will use X64DBG.



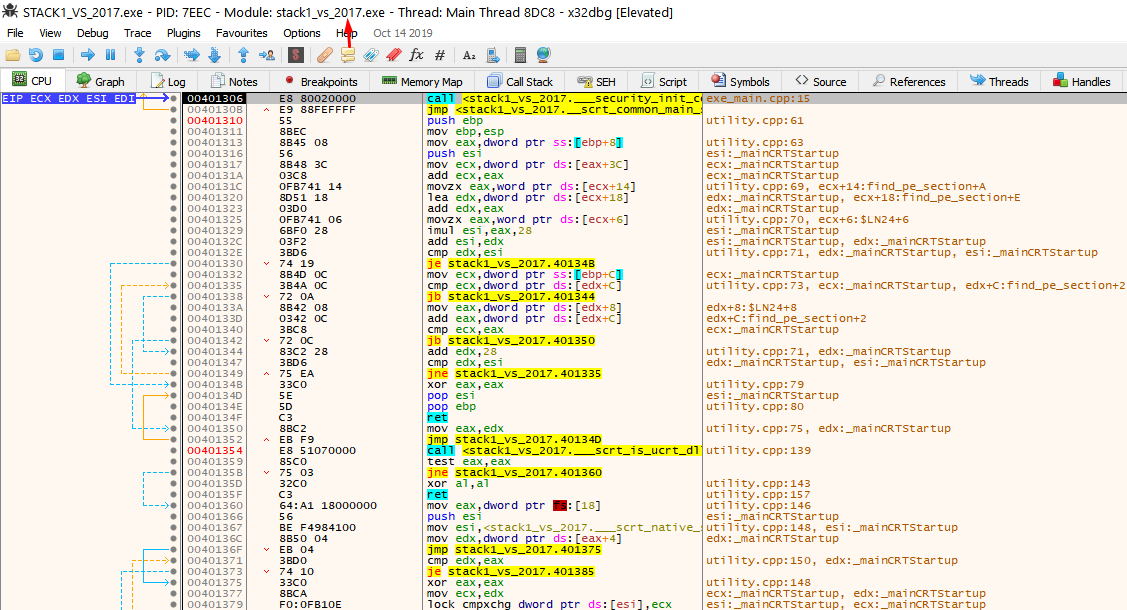
I’ll choose 32 bits version



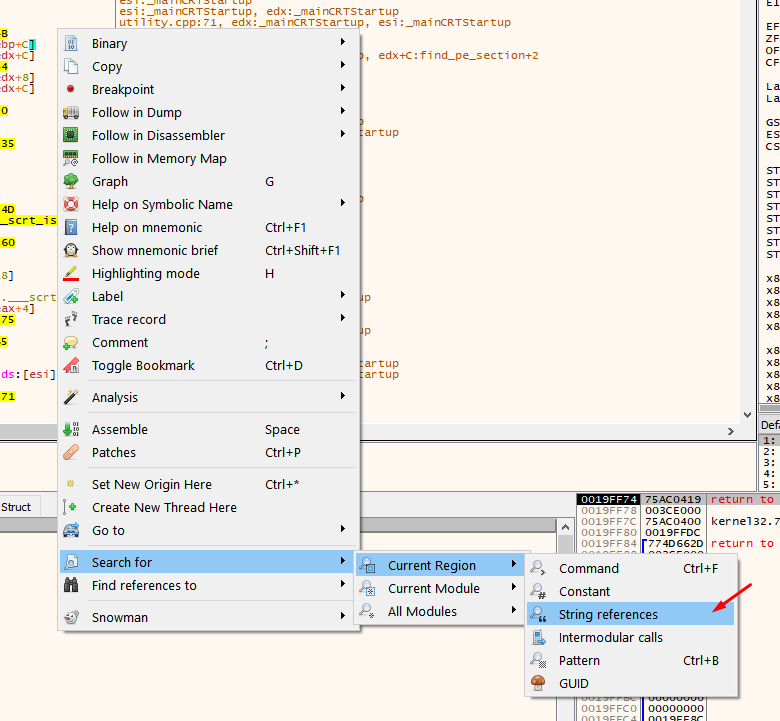


If it stops at ntdll we RUN it again by pressing F9.

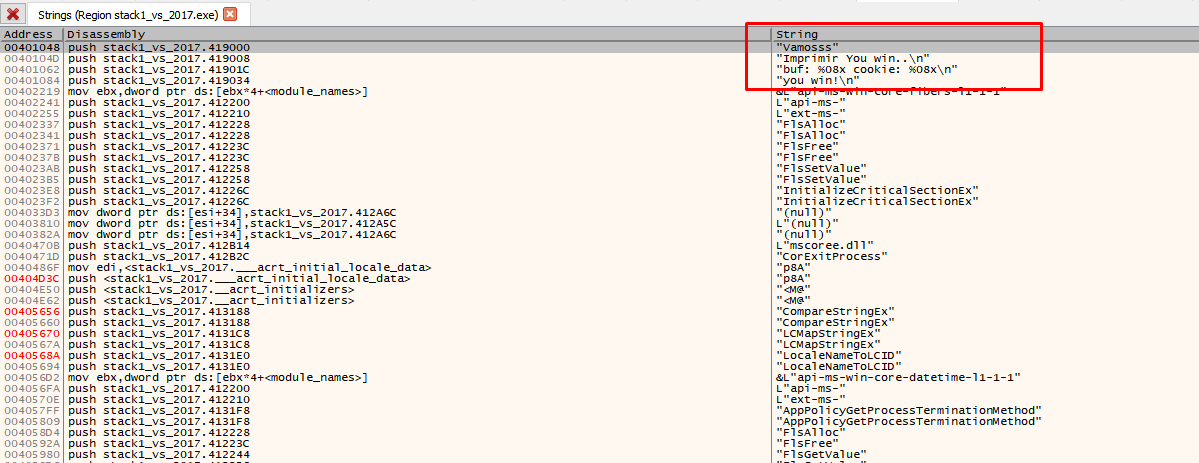
We see it stops in the first instruction of the exercise, what is known as ENTRY POINT or the first instruction executed by a module



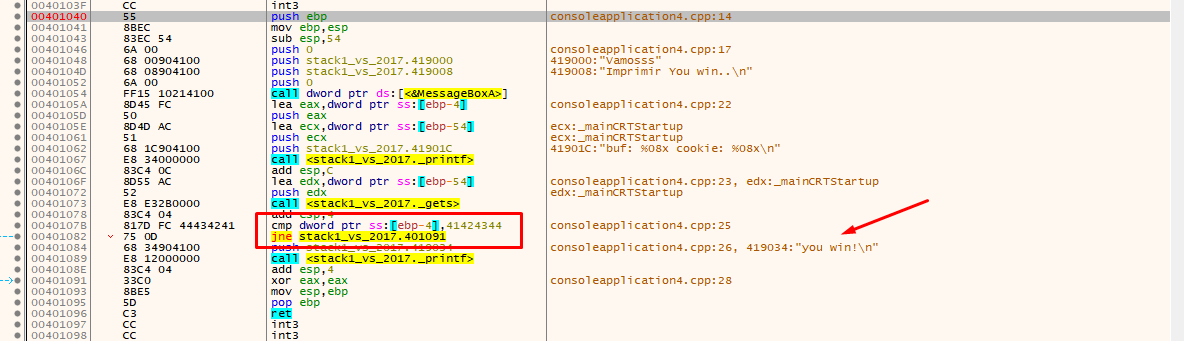
It obviously doesn't look like the original source code. We have to understand that the compiler adds a lot of code for the executable to work and start correctly. We’ll try to find our main function, we’ll be able to orientate ourselves looking at the strings of the program.



We choose only to search in the current region, because we know they will be in this same section.

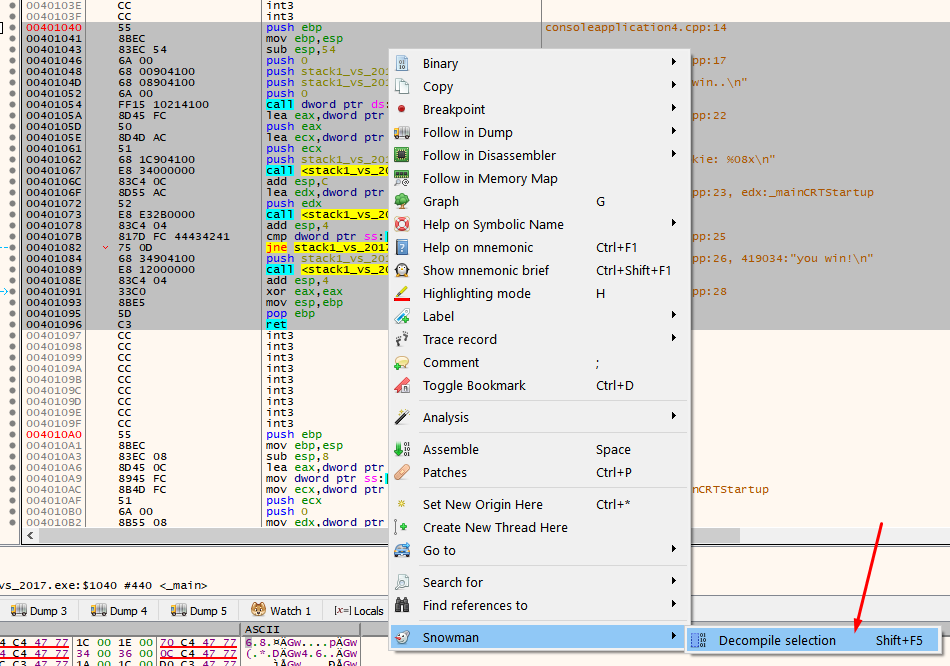


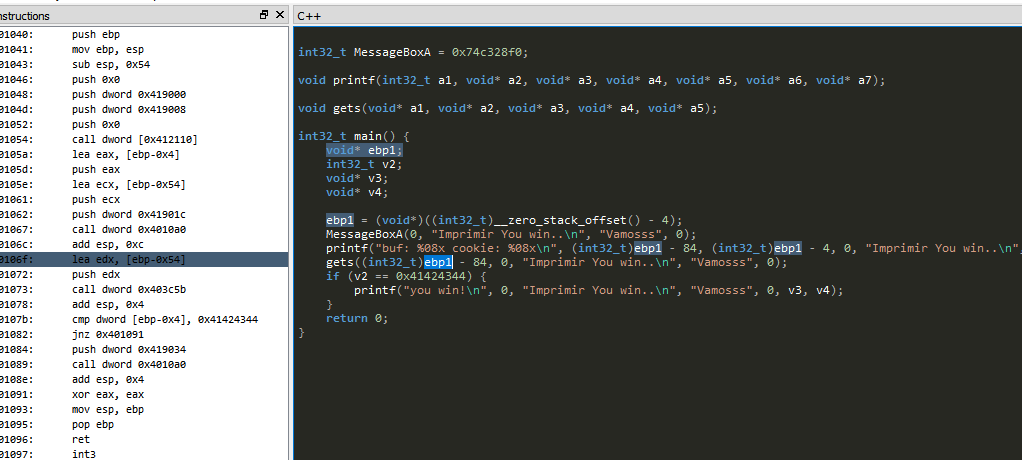
There we see that there are the strings of the program, and others that the compiler added. Let's double click in some of ours strings.



Now everything makes more sense. we see the call to MessageBoxA, the printf, the gets and the comparison with 0x41424344.

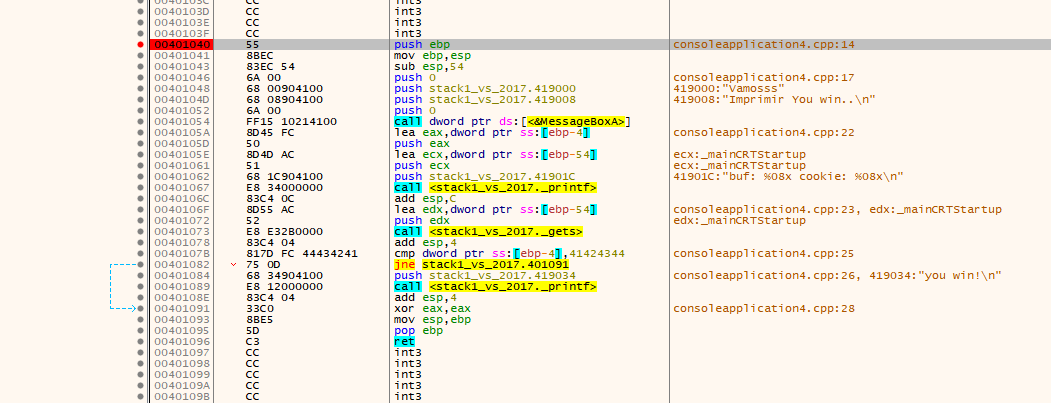
If we also add the SNOWMAN plugin to decompile we can try to see how it decompiles it, that is, how it tries to get the source code or something as similar as possible from the compiled file.

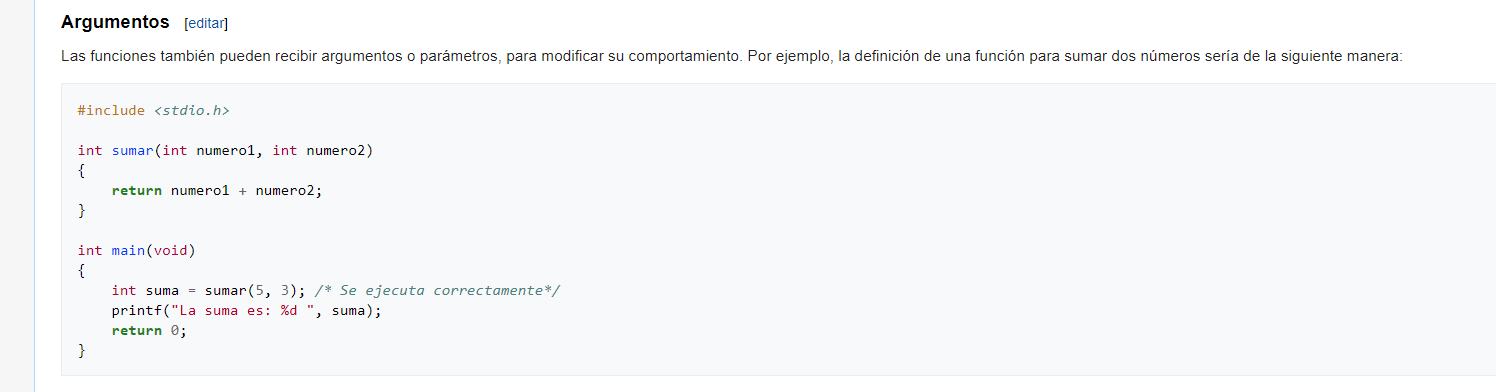




As we see it's not perfect, but there's the best it can be.

I'm going to put a breakpoint at the beginning of the function and press F9 until it stops there.



For those who don't know what a function argument is.

In our case the main function has arguments but these are not used within the function.

**int main(int argc, char \*\*argv) {**

**int cookie;**

**char buf[80];**

**printf("buf: %08x cookie: %08x\n", &buf, &cookie);**

**gets(buf);**

**if (cookie == 0x41424344)**

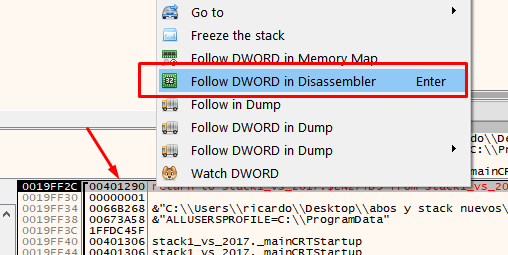
**printf("you win!\n");**

}

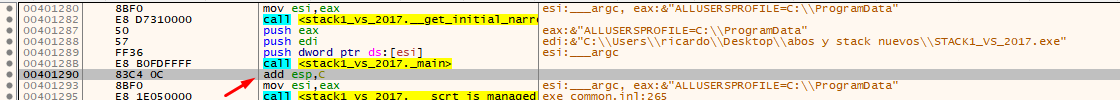
Here we can see the two arguments and in 32 bits compiled executables, these are passed through the stack, just before the function call, so they will be saved in the stack.

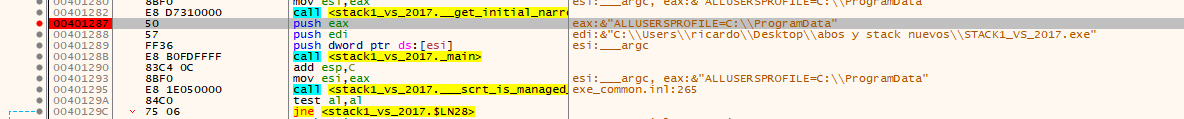
When we are stopped at the start of a function, the first value in the stack will be RETURN ADDRESS, which is where it will return after finishing executing the function and below will be its arguments.

If I right click on the RETURN ADDRESS and choose FOLLOW DWORD in DISASSEMBLER, I'll see where I should return after finishing the function.

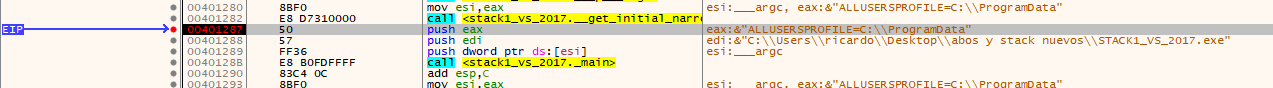


It would return there, that means that the main function was called from the call that is just above, I can put a breakpoint there, restart the exercise and verify that it is that way.



I put a breakpoint there and I restart.

He stopped there



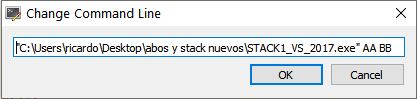
It is going to save the arguments from the main function using those PUSHs.

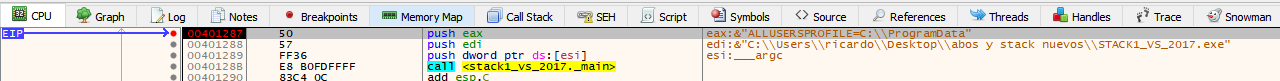
For those who want to know more about the main arguments

<https://publications.gbdirect.co.uk/c_book/chapter10/arguments_to_main.html>

But actually, it's not so complicated. The first argument, *argc*, is an int that indicates the number of parameters and the other, *argv*, is an array of pointers to strings.

We can see that if I change the command line passing more arguments and restart...





It stops there when he is going to save the function’s arguments using the PUSH. The first one that is saved is the furthest and the last one that saves will be the first argument of the function.

I trace and verify that in each PUSH it keeps the function’s arguments.(Do not confuse the function's arguments with the arguments passed to the console)



There he stored the function’s arguments.

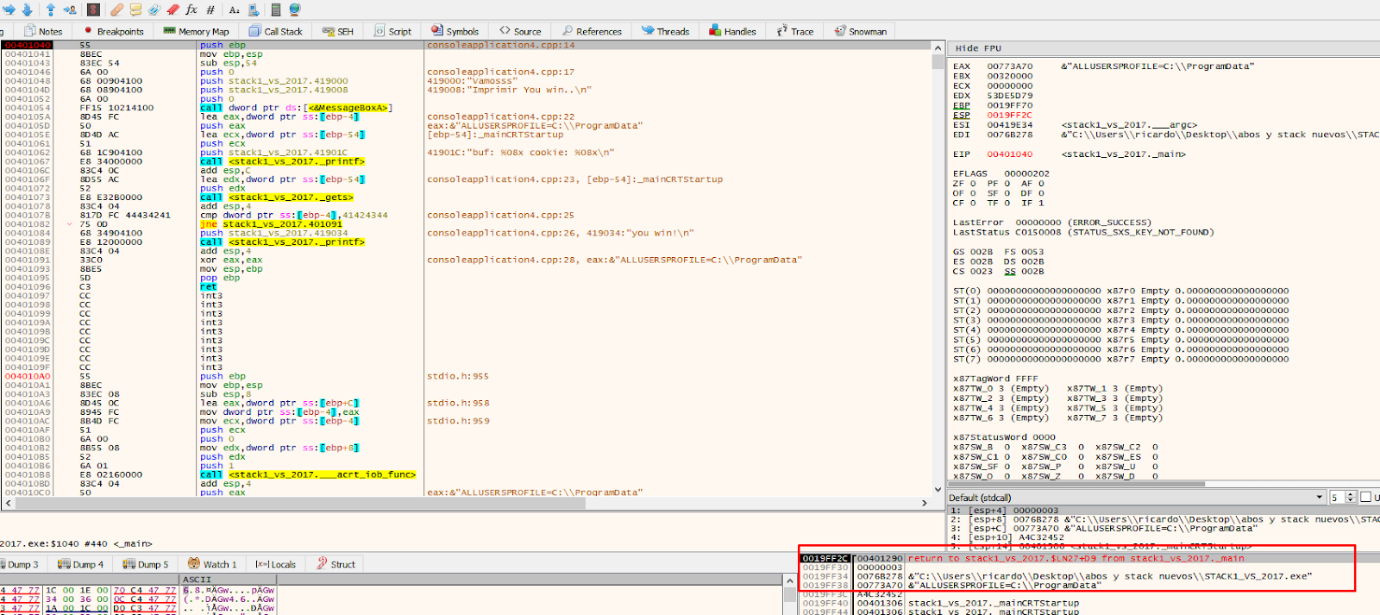
The previous one is the first, argc, which is worth 3, since it marks the number of console’s arguments and there were three counting the name of the executable.



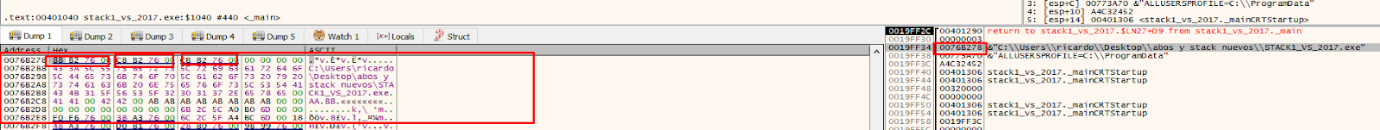
These are the three console’s arguments.

Now we press F7 to perform a STEP INTO and enter the function.

There we see that when entering the CALL, it saves the RETURN ADDRESS

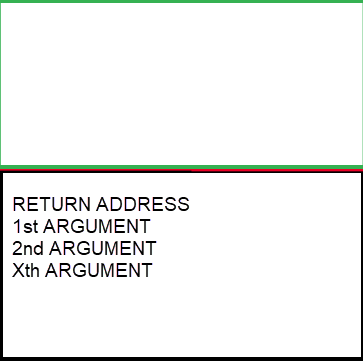


So, as we said when entering a function, the first thing that will be in the stack will be the RETURN ADDRESS (in 32 bits compilation) and below there will be the function’s arguments, first the first argument and successively down the rest.



The second argument, as we saw, is an array of pointers. We see that in memory there are three pointers to the three strings passed as console’s arguments.

1. 32 BITS



We are stopped at the beginning of the function, just below we have

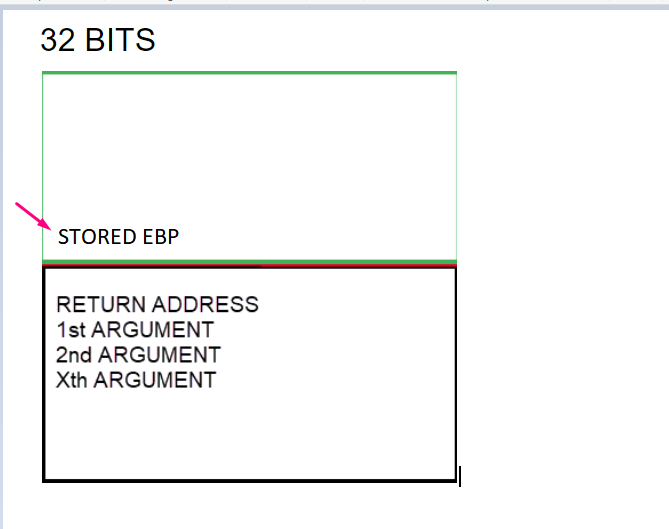
the RETURN ADDRESS and the function’s ARGUMENTS.

We clarify that it is a 32-bit compilation because in 64 bits the function’s arguments are passed in another way that we’ll see later.

Then the function begins to execute. The first thing in our function is the so-called PROLOGUE, which stores the EBP value of the function that called ours.

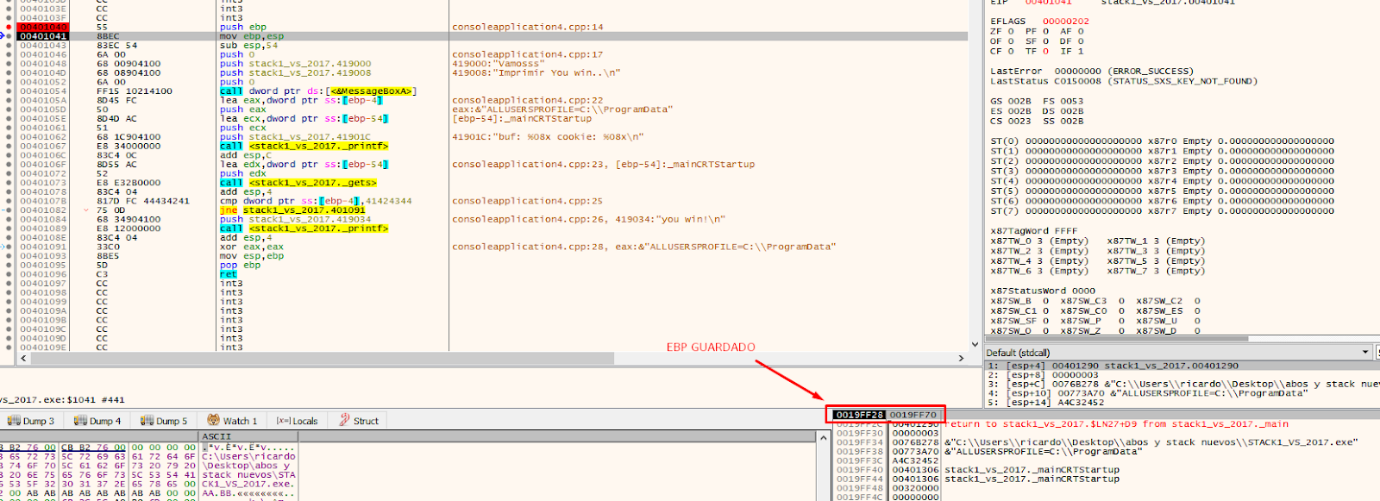
https://lh5.googleusercontent.com/awPsDT4_mg56uaQlVY_6eiujzBe0YO4jb37ozTaL8mDlCirm8po5dB9HxrIgPIGVBGiVR4FdOJ9nbKirxGPUJFL-CQMCi5SyQq6yE8q9m6KzGivcWTxYYccHluj3ib_jT6vMmBgh

That will cause the STORED EBP value to be stored just above the return address.



If I execute the instruction by pressing F7.

I can see that the STORED EBP value is located in the stack just above the return address.



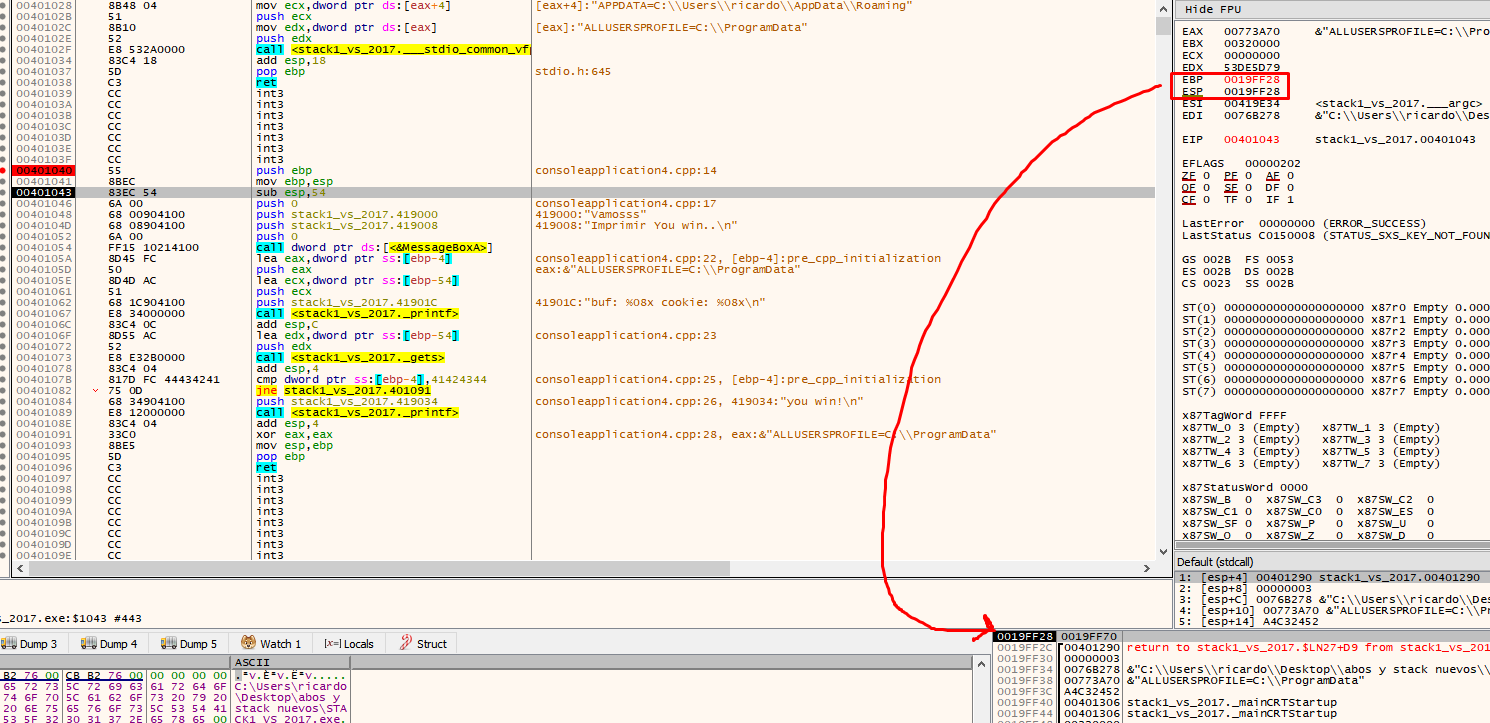
The next PROLOGUE instruction is:

mov ebp, esp

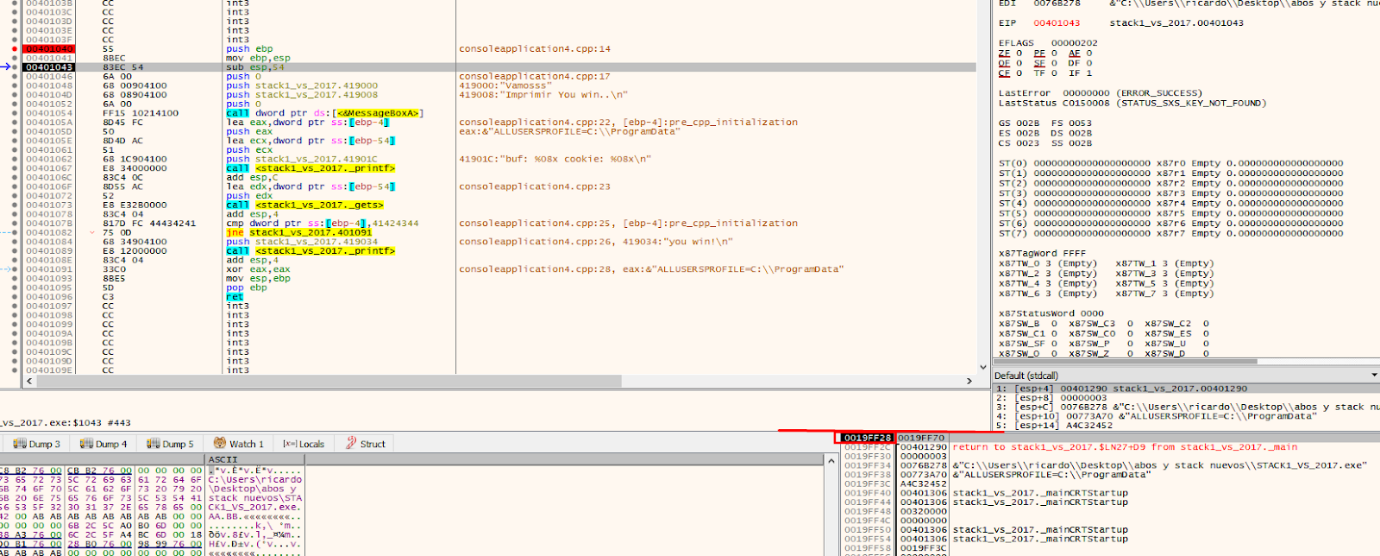
This sets the EBP for the current function. The one that was saved (stored EBP) corresponds to the parent function, that is, the one that called this function.

In this example, the function is based on EBP, in other cases it may differ.

By putting EBP at the current ESP value, we achieve to create the framework of our current function.



What happens is that being an EBP BASED function, from now on, within the function the EBP value will be maintained and the same will be taken as a reference, while ESP will vary.



A very important point is that the EBP value is taken as a reference. In the EBP BASED functions the variables and arguments can be named by their distance to this address that will be stored in the EBP value until the epilogue of it. In the list we can see several variables that are marked as EBP-4 or EBP-54, referring to the EBP value it takes at this point.

We can say that once EBP takes its value after the PROLOGUE, it will be like a watershed, so the arguments will always be down that direction, so EBP+XXX (add something to EBP), refers to arguments (the saved ebp and return address are also below but will have no references in the code), while the variables, as we will see, will be above this address so a reference to EBP - XXX (subtracting), refers to some local variable.

So, in general, in EBP BASED functions

EBP + XXXX  = arguments passed to the function

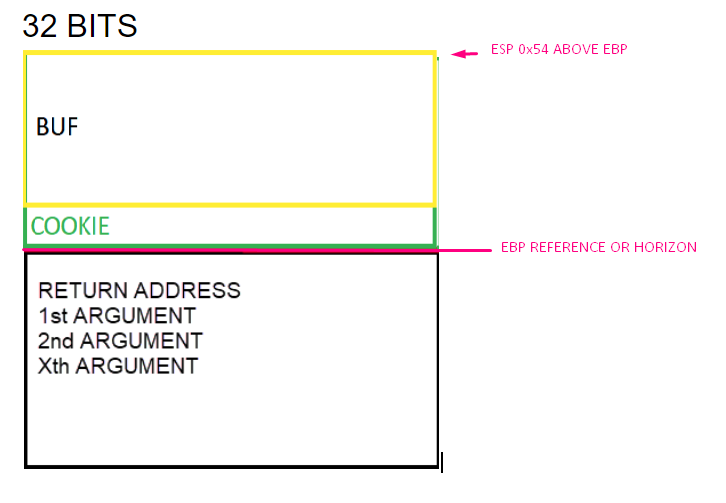
EBP - XXXX =  local variables of the function

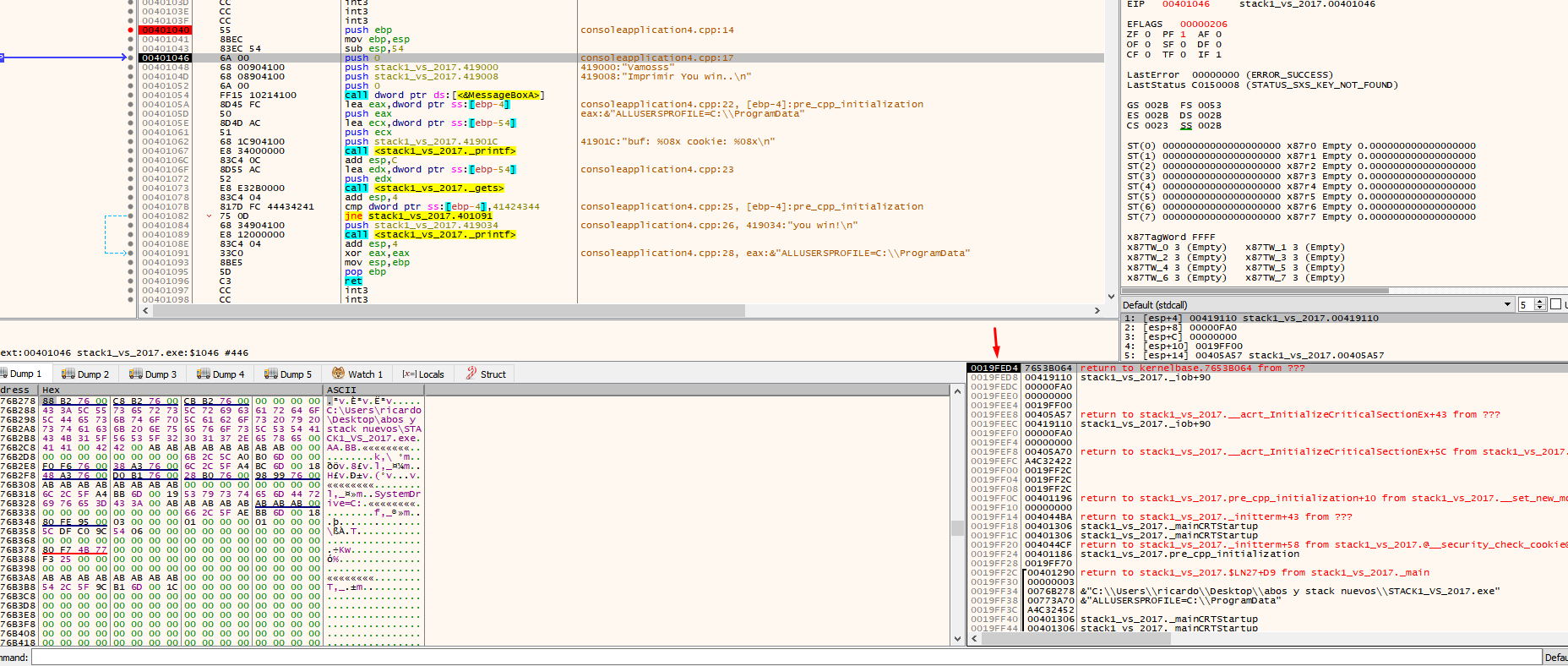
After the PROLOGUE there will be some way to reserve space for the variables, in our case, this is done by moving ESP up so that space left below is reserved for the sum of all the lengths of the variables and, sometimes, a little more just in case, which depends on the compiler.

00401043 | 83EC 54 | sub esp,54

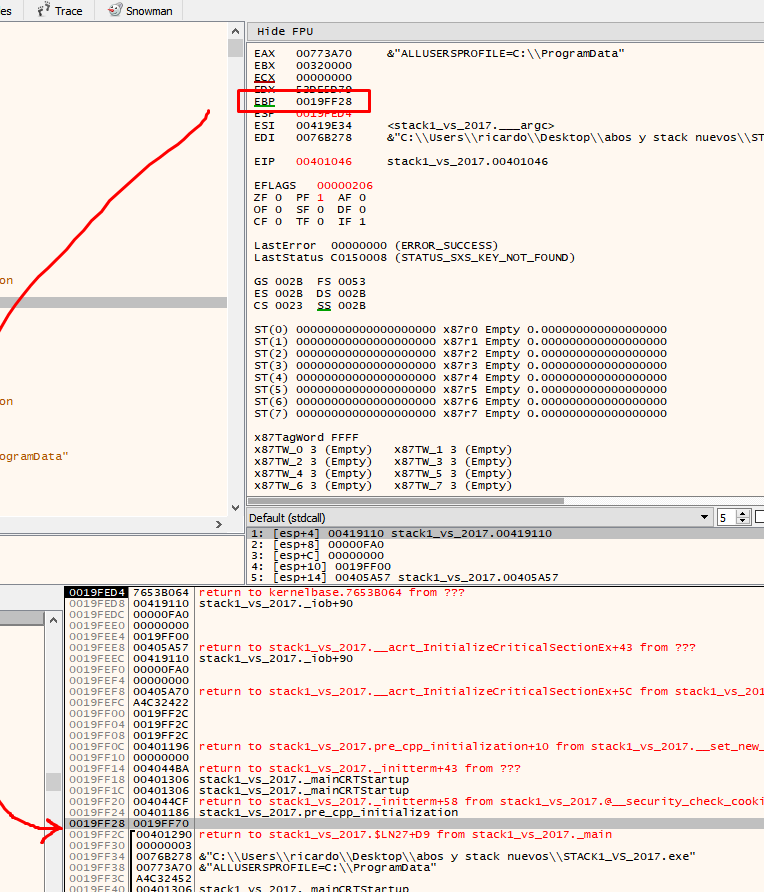
We see that ESP will be located above EBP that will remain as a reference, and that 0x54 passed to decimal is 84 which is the sum of the length of buf and cookie (remember that they were 80 and 4 respectively).



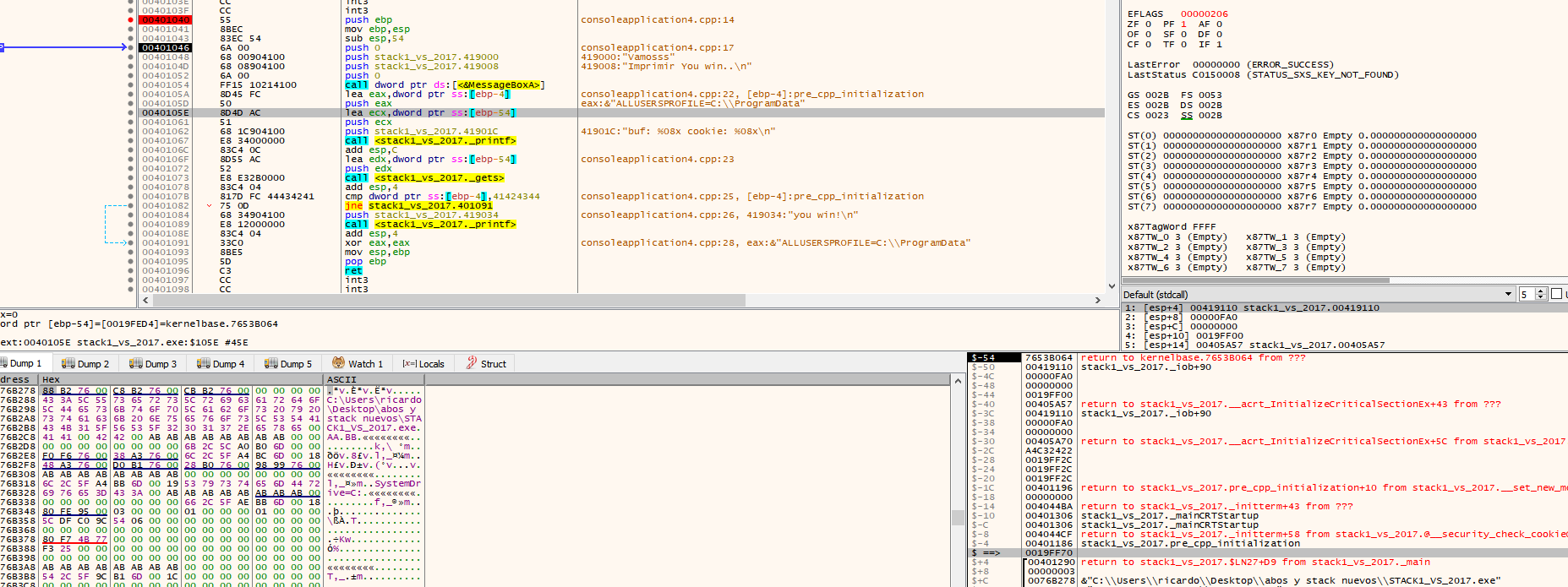




When executing this instruction, a space is created for the buf and cookie variables of 84 bytes, you can click on the first column on the stack, look at the EBP value and search it in the stack, obviously now it will be displayed lower in the stack from top of the stack.

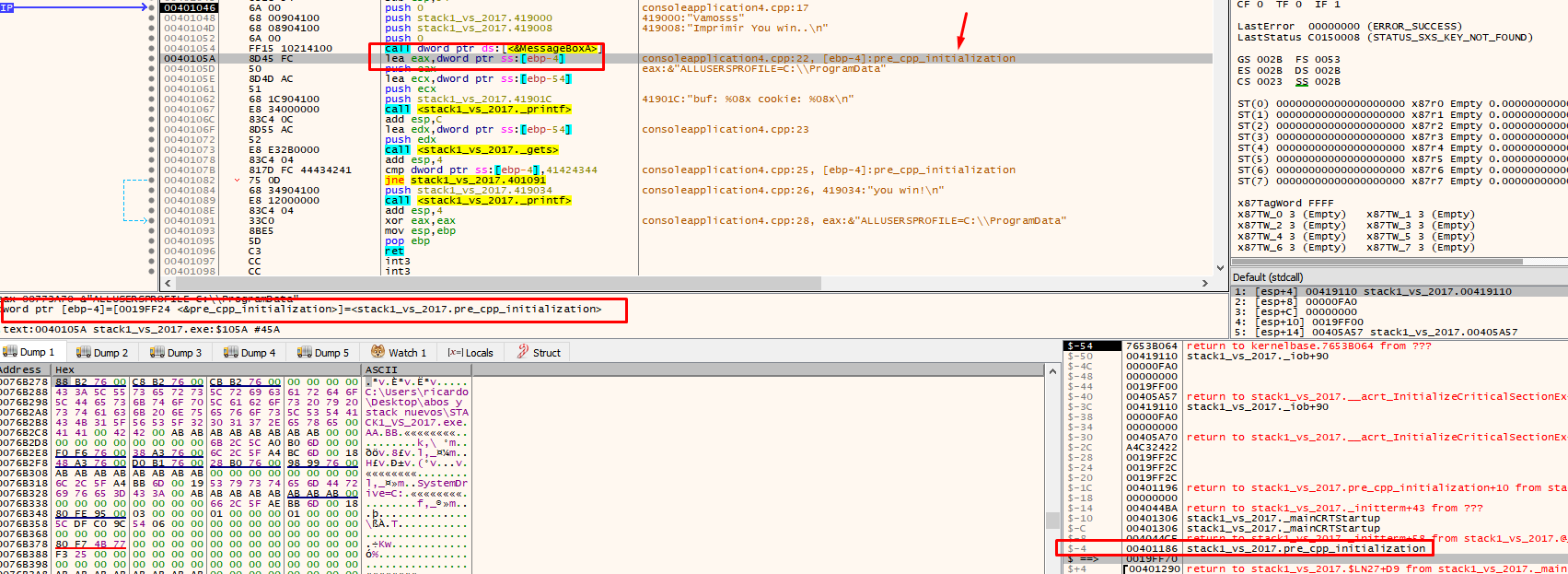


I double click there.



In this way we will have the values ​​with respect to EBP also in the stack shown in the first column

For example, ebp-4 coincides in the list, in the -4 displayed in the first column of the stack and in the clarification also, it appears as ebp-4.

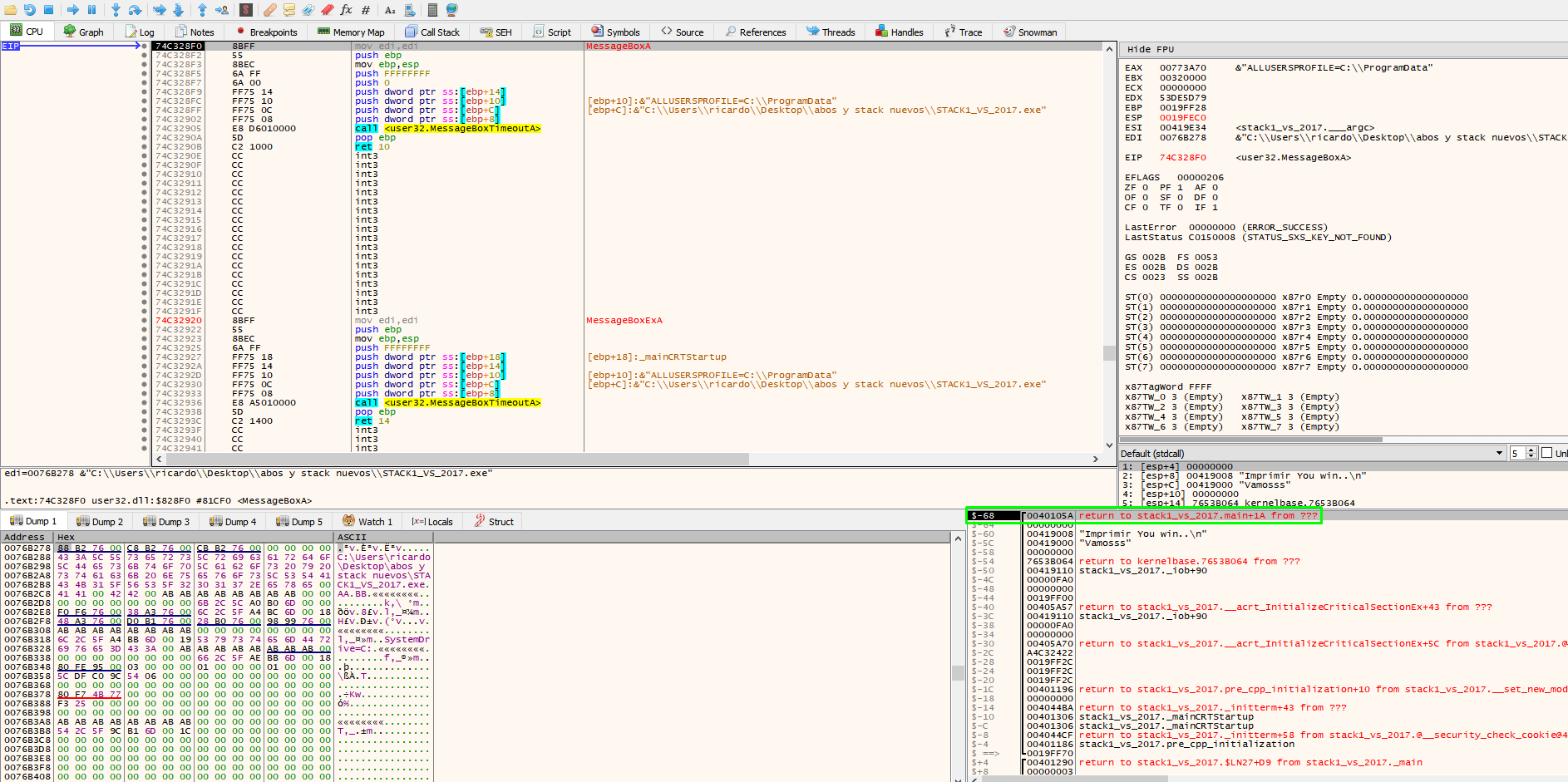


The position where ESP is located to reserve variables will always move up because it must respect the space allocated for the variables. For instance, when making the four pushes for MessageBoxA it places them above the reserved space in the ESP.

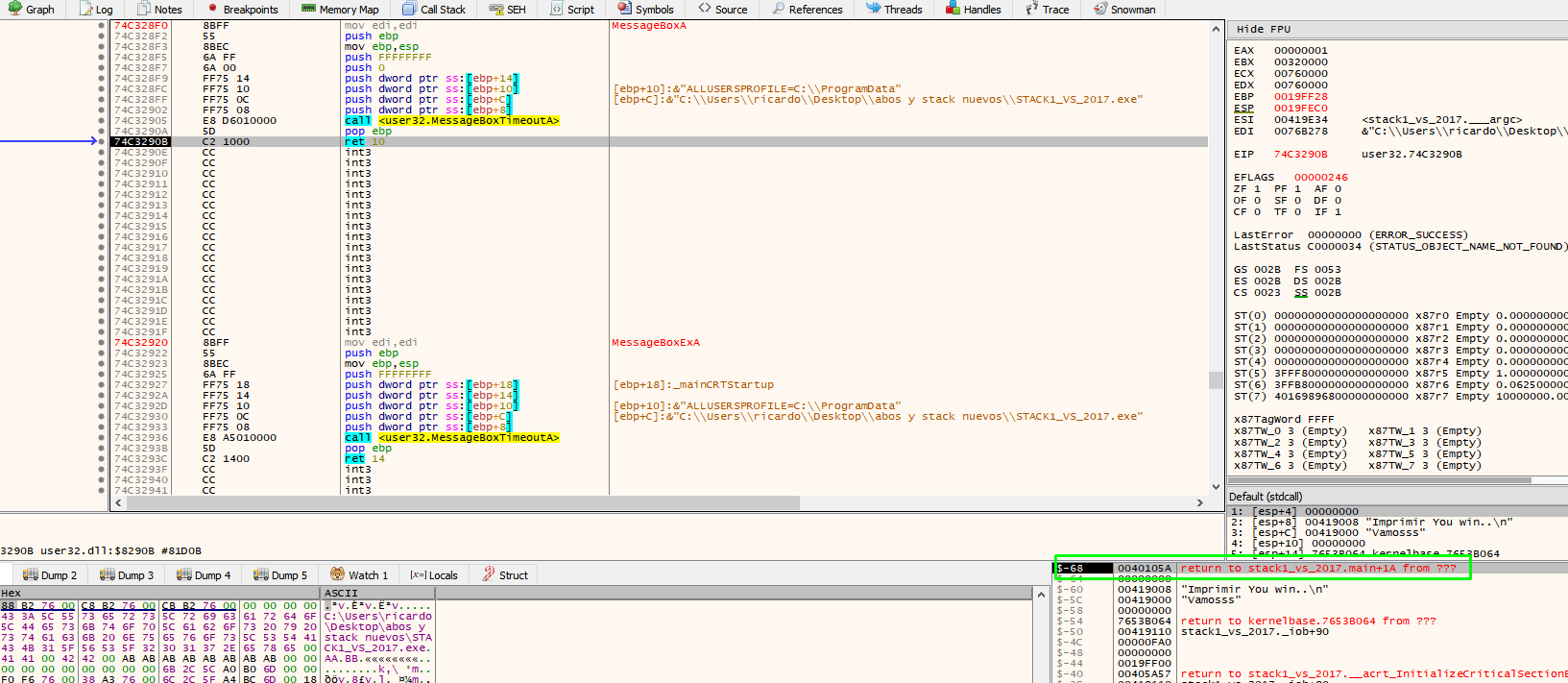
If I look at the stack below, I see in green the 4 arguments that I added above the reserved space that is marked in red.



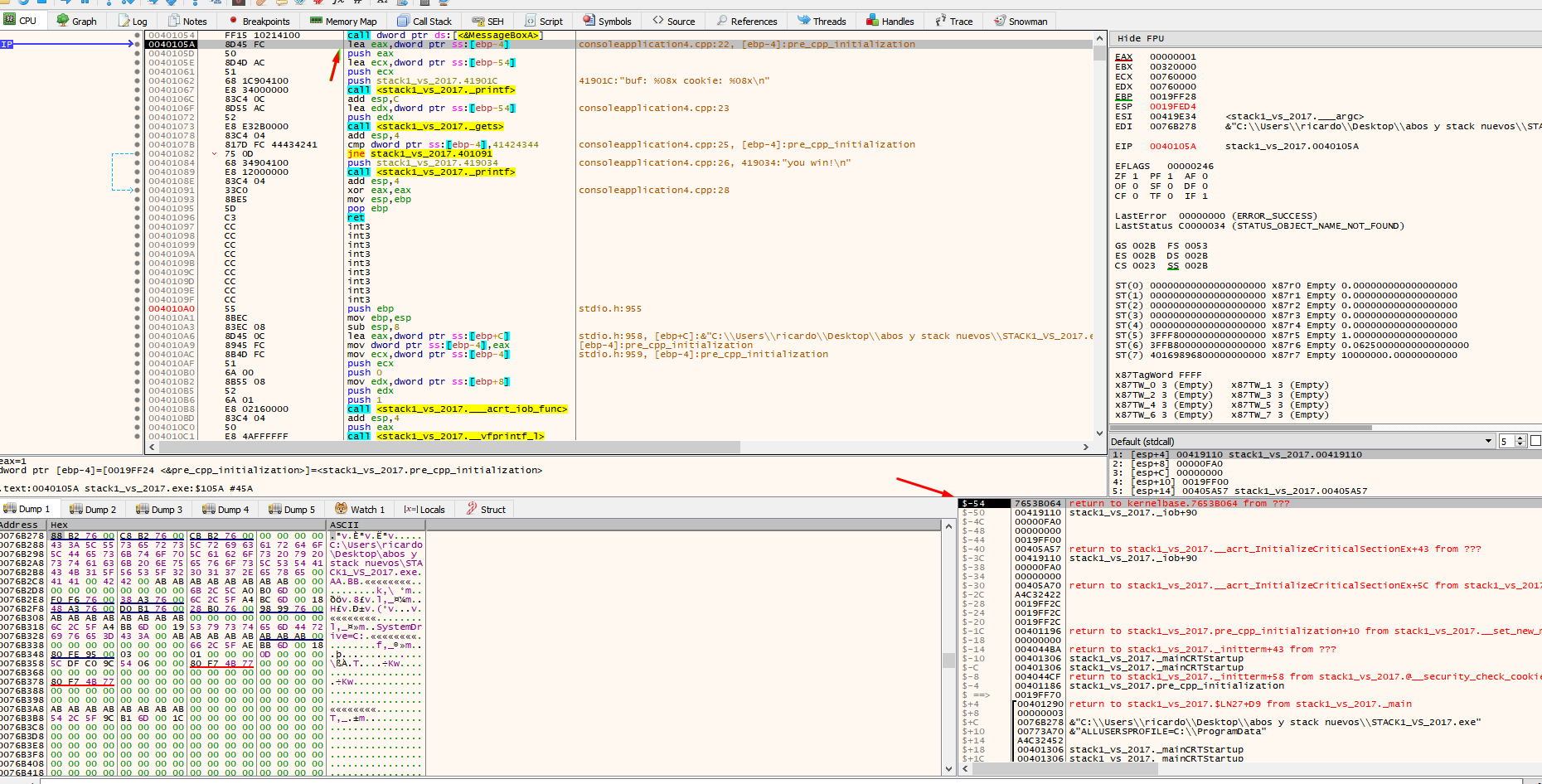
When entering MessageBoxA, the RETURN ADDRESS of that function is stored on the stack.



There, is the return address of MessageBoxA, when I get to the RET of that function by tracing with F8 and I accept the MessageBox.



We see that you will return just below the call to MessageBoxA.

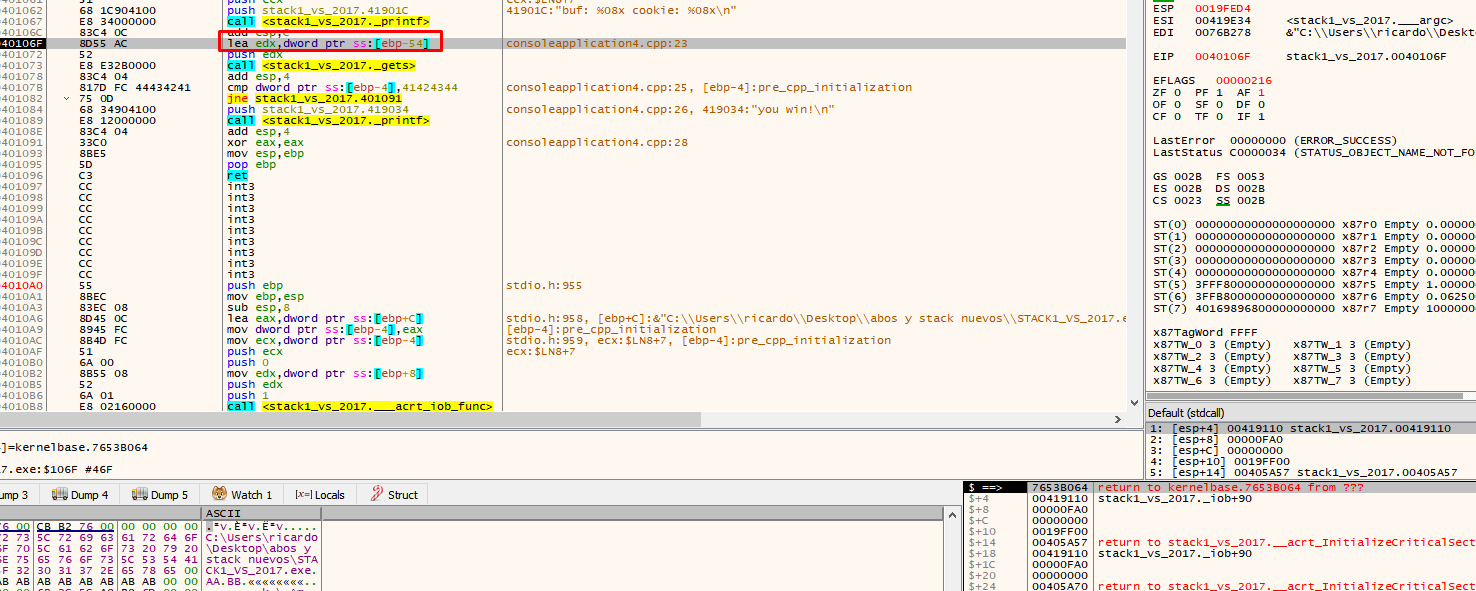


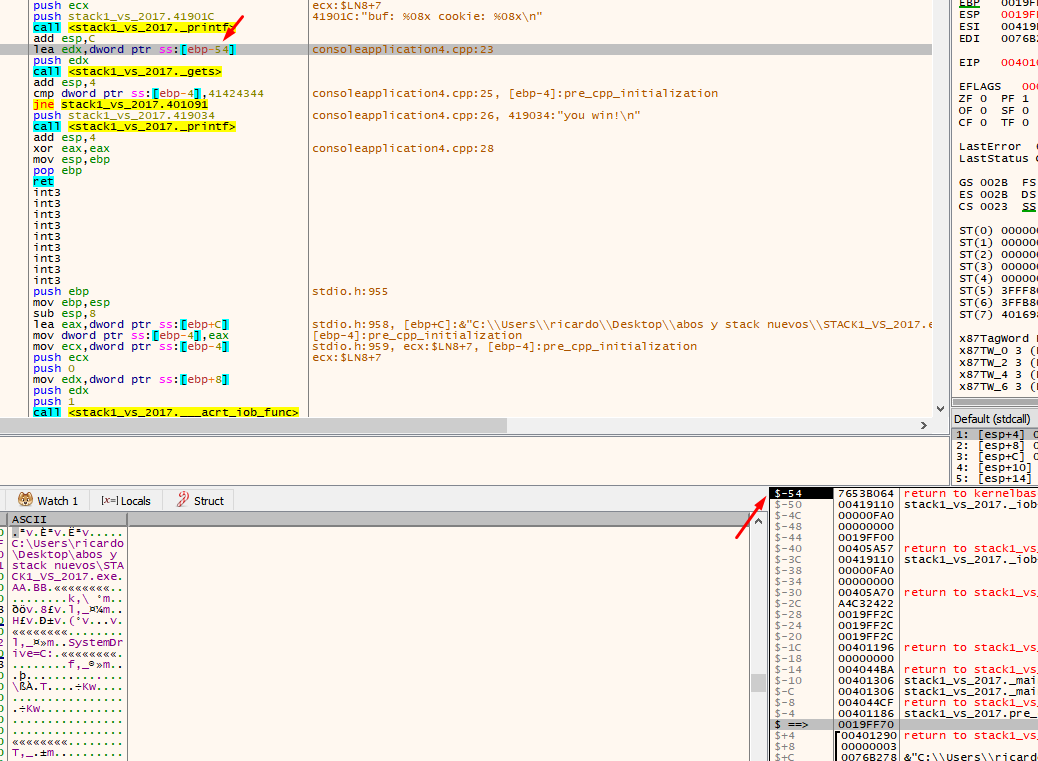
And those “PUSH”ed values that you had saved for MessageBoxA and the RETURN ADDRESS of that function were already used and ESP is again just above the reserved area as before calling any function, the same will happen with the call to printf will save the function’s parameters with PUSH and the RETURN ADDRESS of the same and ESP will rise but when leaving the same it will lower again just above the reserved area

Once you pass over the printf, the buf and cookie addresses are printed.

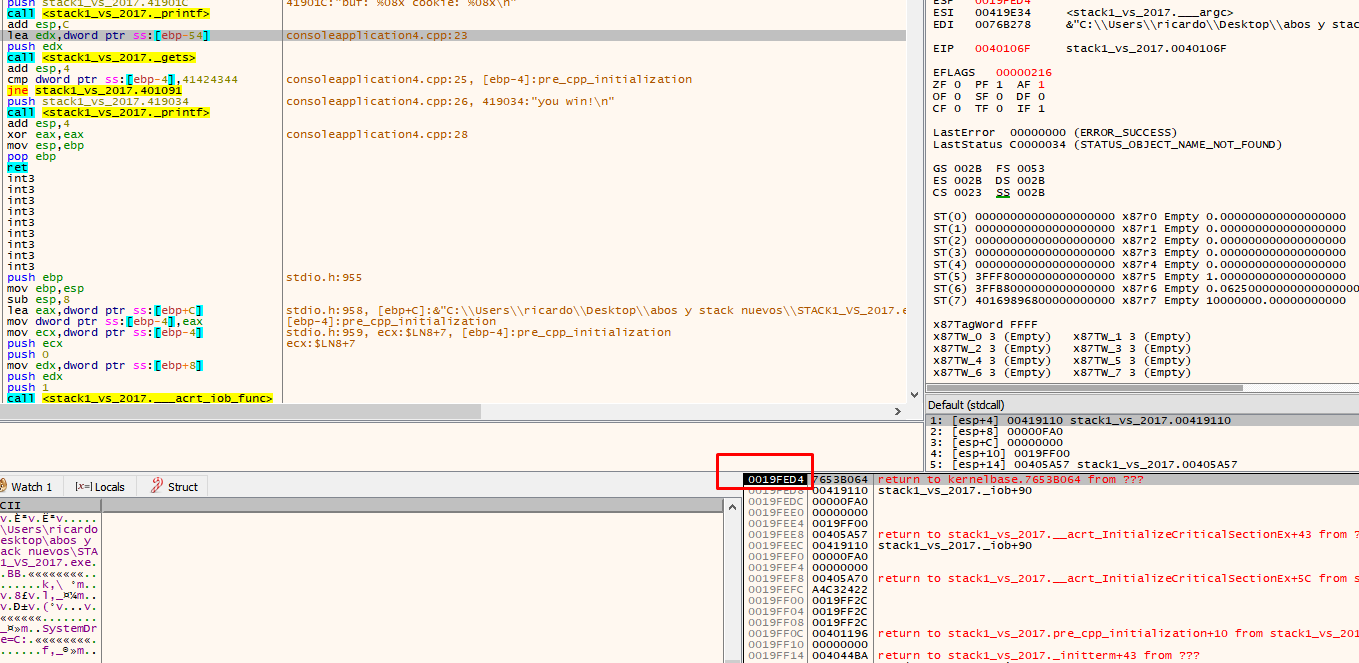


The address of buf in my machine was 19fed4 and the address of cookie was 19ff24. In yours could completely different.

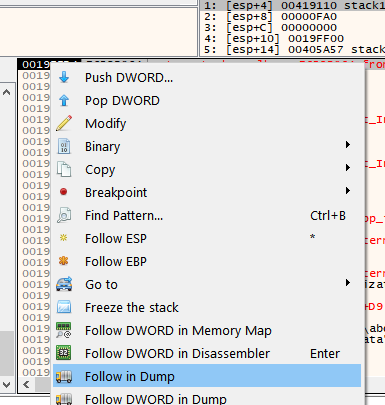


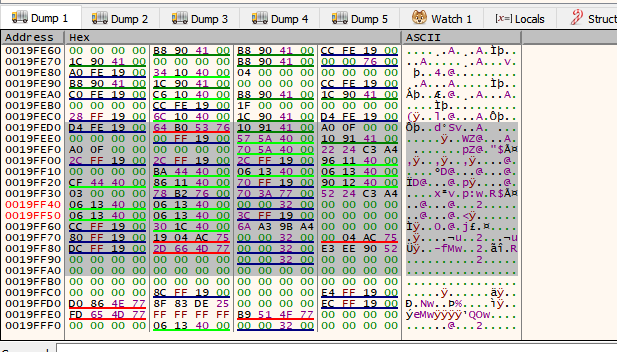


There we see that it uses EBP-54, this is the address of our buf (19fed4 in my machine), if I double click on the stack, the debugger shows the address as $-54.



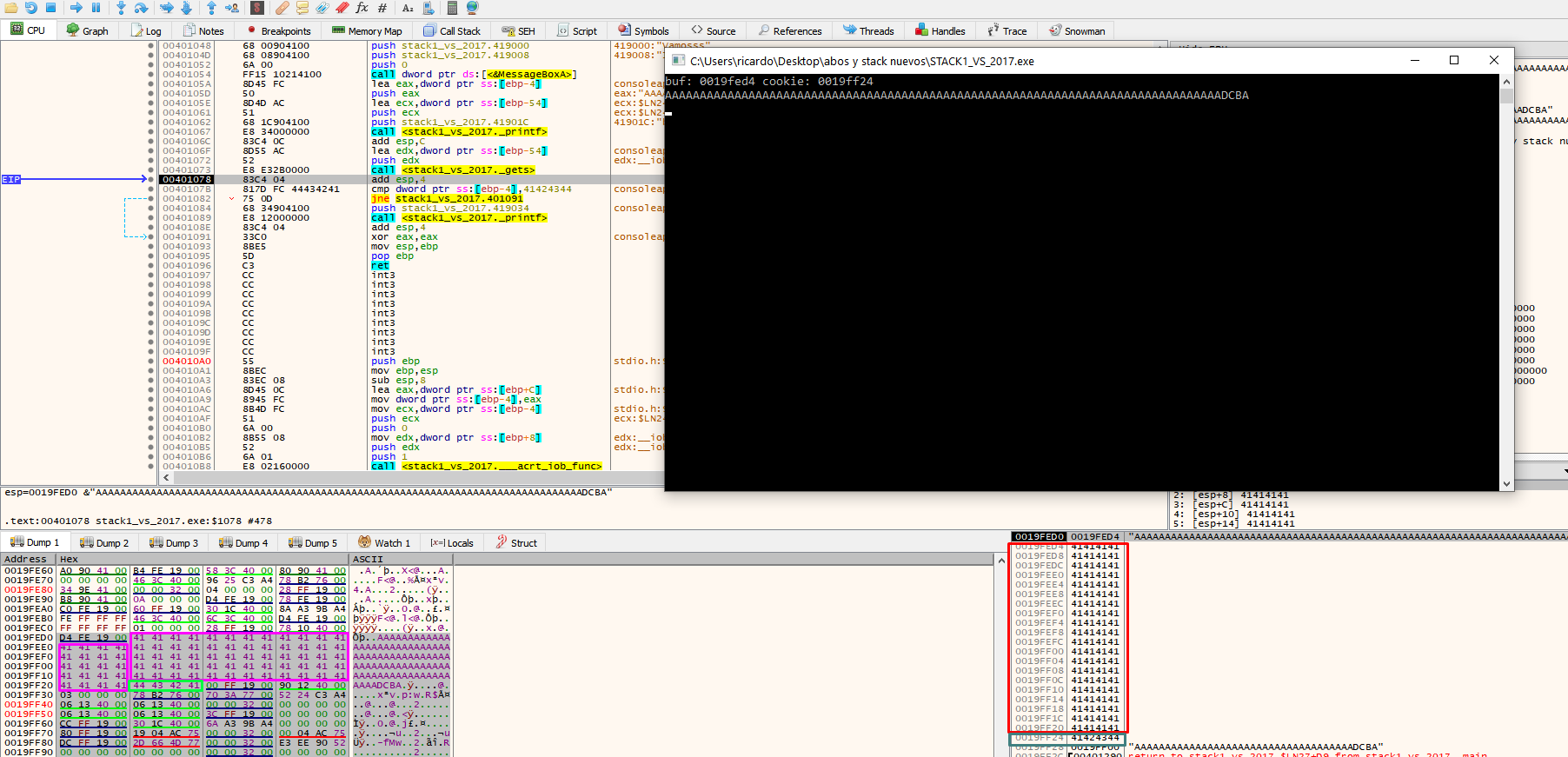
Now as that address is where I will save the data that I type, I can put it in the dump to see how it saves the bytes there too.





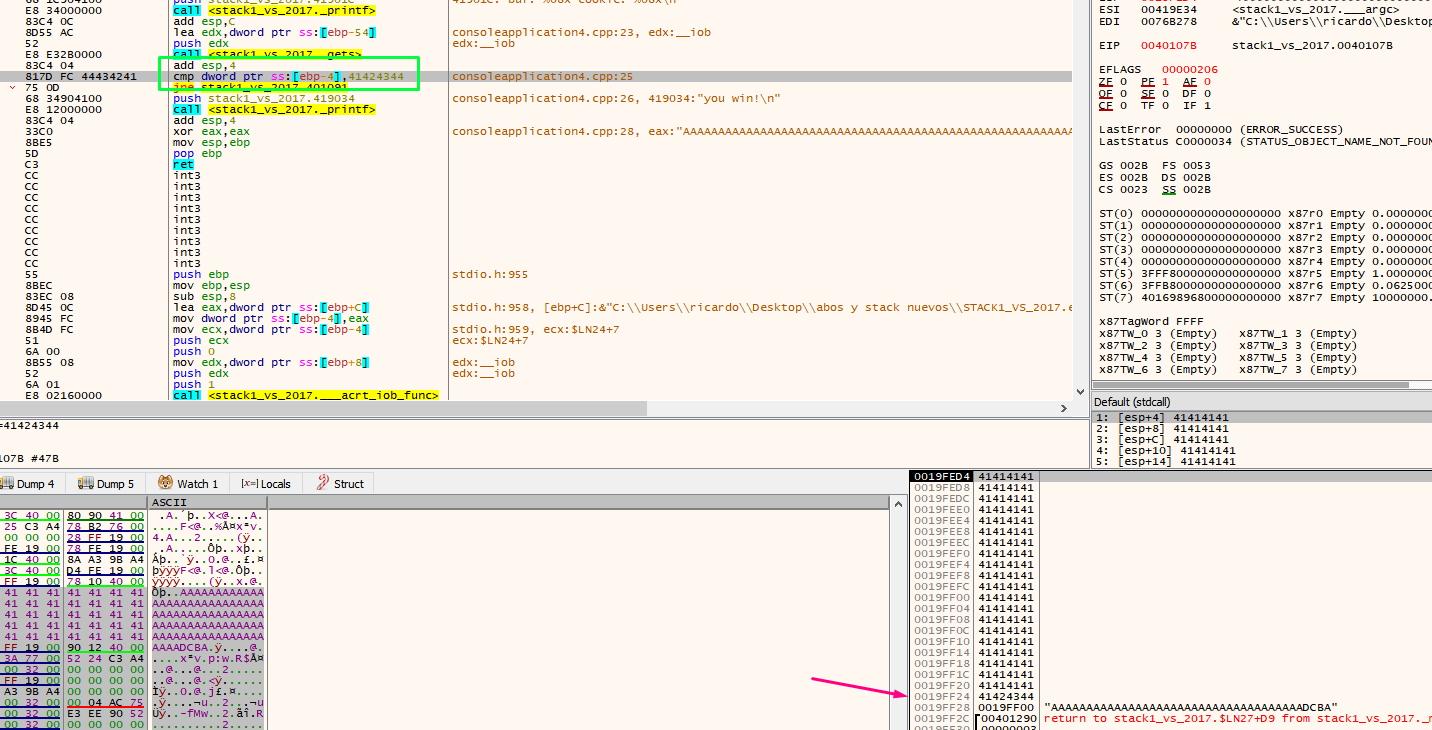
There it is, more up it’s not displayed because there is no more data below.

When I set over with F8 I go to the console, type the data that I want to send and once I press ENTER to finish, the buffer “buf” will be filled and also the cookie value too.



We see that cookie was at address 19ff24 on my machine.

The next instruction compares cookie with 0x41424344.



We see that EBP-4 says that it is a cookie, in addition to the address if we reset the HORIZON to the value of EBP as before.

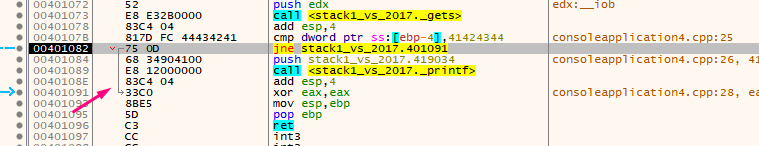


I make double click there.

We see that EBP-4 is cookie since it is in the -4 of the stack by zeroing the HORIZON.



We see that the program will not jump and will go to YOU ​​WIN.





We achieved the objective manually, that the program prints you WIN.

We have dynamically analyzed stack1, using X64dbg which is a debugger and does not allow us to analyze it without running the program, for this we must use other tools such as IDA PRO, GHIDRA or RADARE.

I can make a script model to exploit the exercise from Python.

**import** sys

**from** subprocess **import** Popen, PIPE

payload = **b"A"** \* 80 + **b"\x44\x43\x42\x41"**

p1 = Popen(**r"C:\Users\<user>\Desktop\abos y stack nuevos\STACK1\_VS\_2017.exe"**, stdin=PIPE)

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

print (**"Enter para continuar"**)

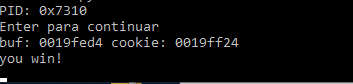
p1.communicate(payload)

p1.wait()

input()

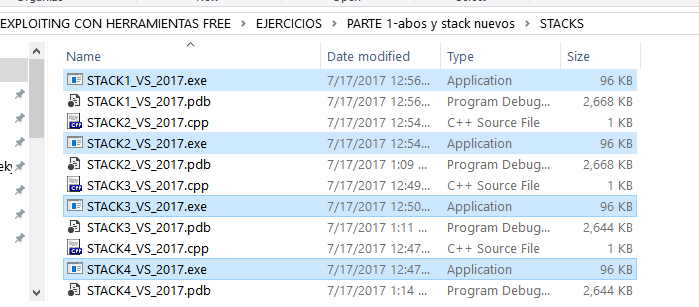
As it should be done in Python 3 I must place the parentheses in the prints and be careful when adding strings that must be bytes (put b in front of what were strings in Python 2).

I notice that the path is correct and when running it.



Well, we already have a Python 3 script model to exploit stack1, in the next part we will continue with the static analysis in IDA, RADARE and GHIDRA of this same exercise.

While you can try to trace and understand with X64dbg for now the other 3 stacks exercises that are very similar to this.



See that in addition to stack1 there are 2, 3 and 4, you can try to solve them are very simple and similar to stack1, so do not get bored hehe.

The next part we will see IDA FREE, RADARE y GHIDRA.

See you in part 3.

Ricardo Narvaja

Translated by @arrizen

25/10/2019

1. “We see that the instruction will take those integers (int) and insert them in the output string with base 16 or in hexadecimal, 08 refers to the fact that if the number has less than 8 digits it will fill it with spaces.” [When carrying 0 in front this character is used to fill] [↑](#footnote-ref-0)
2. <https://www.arumeinformatica.es/blog/los-formatos-big-endian-y-little-endian/> [↑](#footnote-ref-1)