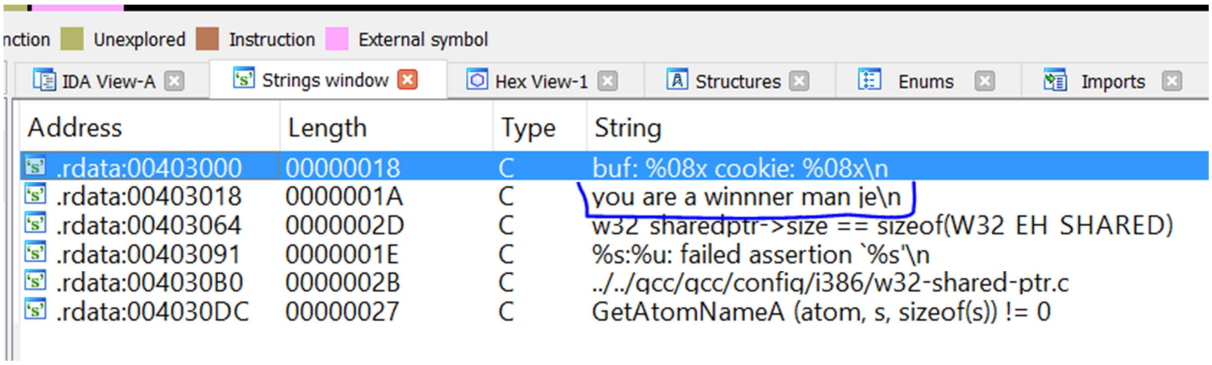
REVERSING WITH IDA PRO FROM SCRATCH
PART 23

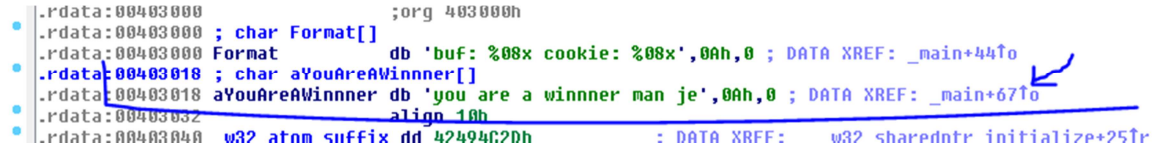
I had to solve the pending IDA1.exe exercise.

```
00401290
00401290 ; Attributes: bp-based frame
00401290
00401290 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401290 public _main
00401290 _main proc near
00401290
00401290 Format= dword ptr -0B8h
00401290 var_B4= dword ptr -0B4h
00401290 var_B0= dword ptr -0B0h
00401290 var_9C= dword ptr -9Ch
00401290 Buffer= byte ptr -98h
00401290 var_C= dword ptr -0Ch
00401290 argc= dword ptr  8
00401290 argv= dword ptr  0Ch
00401290 envp= dword ptr  10h
00401290
00401290 push    ebp
00401291 mov     ebp, esp
00401293 sub     esp, 0B8h
00401299 and     esp, 0FFFFFFF0h
0040129C mov     eax, 0
```

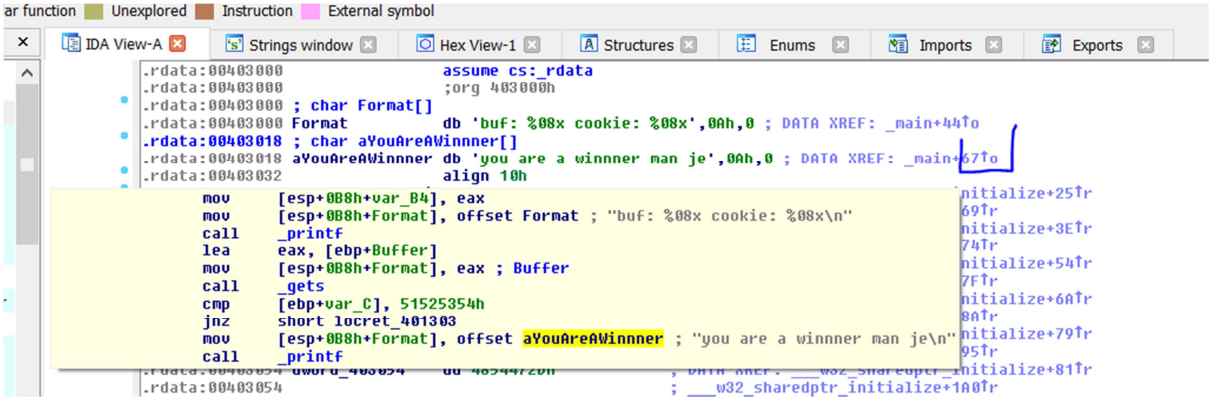
The exercise is based on EBP like the ones above. The difference is that this example is compiled with DEVC ++ which has a particular way of compiling the calls to APIs, different from what is customary and if you have never seen it, it can make you dizzy. Well, "Let's go part by part," said Jack. The first thing we see is the main function, if it does not appear, you can also get to the important function by looking at the strings.



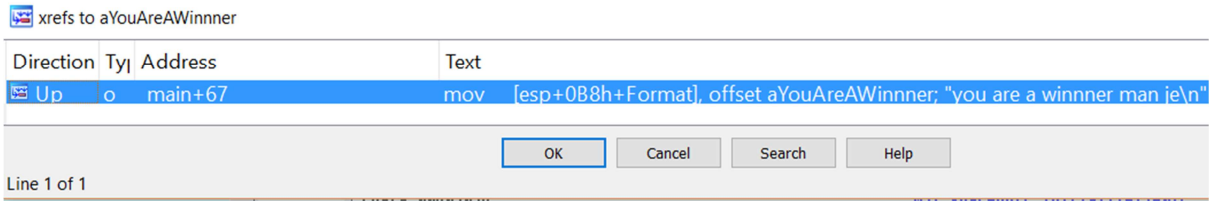
Double click on the string "You are a winner man je". We get to…



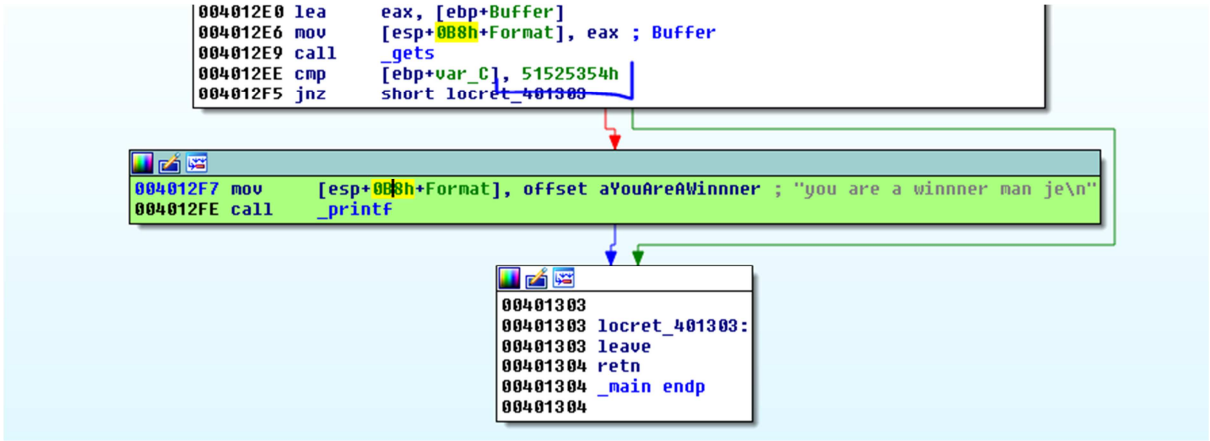
Passing the mouse over the arrow of the reference we can see where it is called.



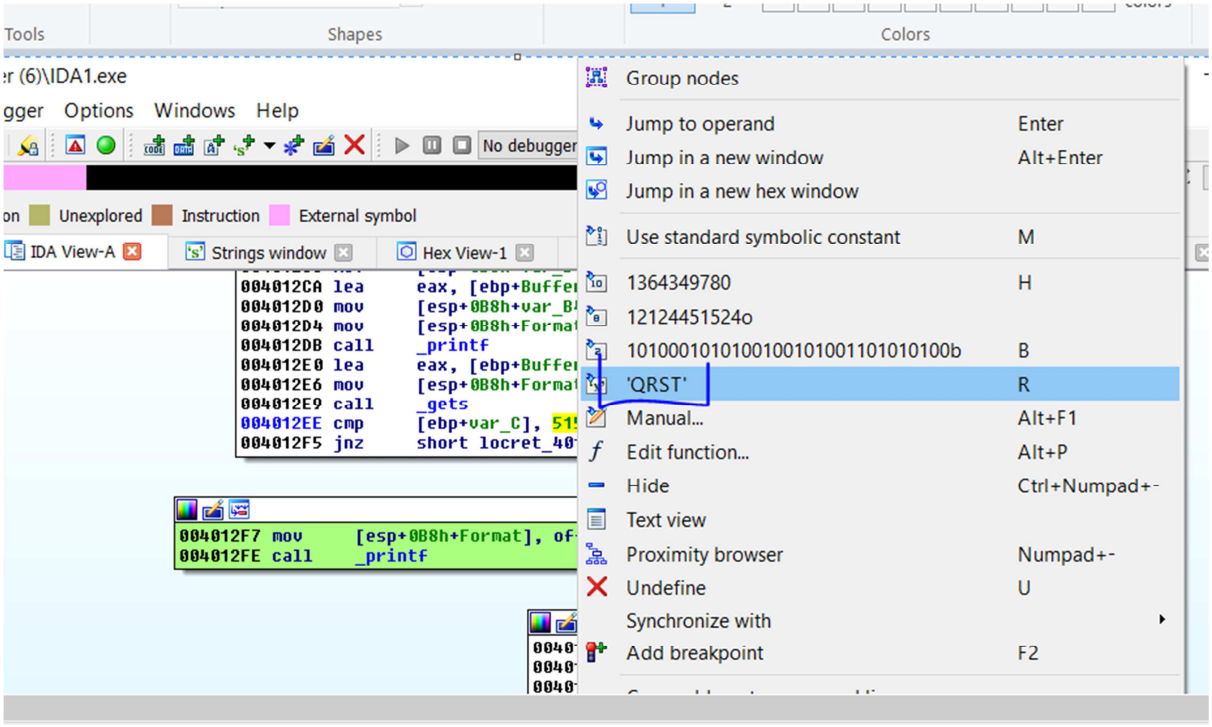
But better doing CTRL + X we go to where that code is.



Go there by clicking on the reference.

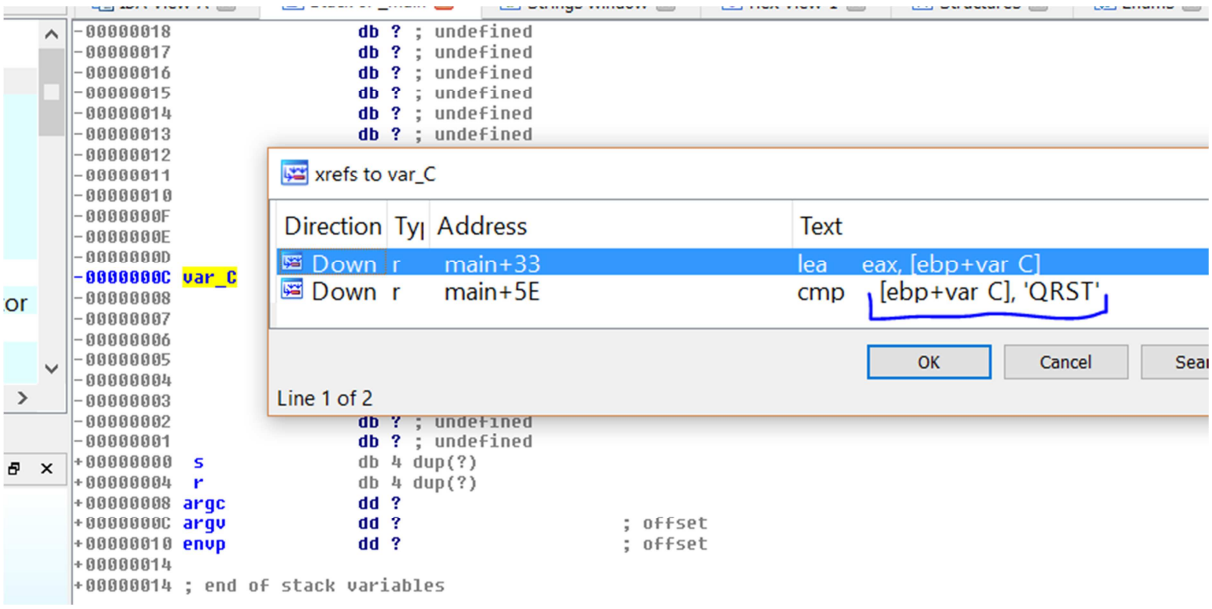


The part where I must arrive changed to green color that would be the GOOD BOY. We see that just before, there is a comparison of a variable var_C with a constant 0x51525354. If we right click on that value 0x51525354 the alternatives to represent it come out.



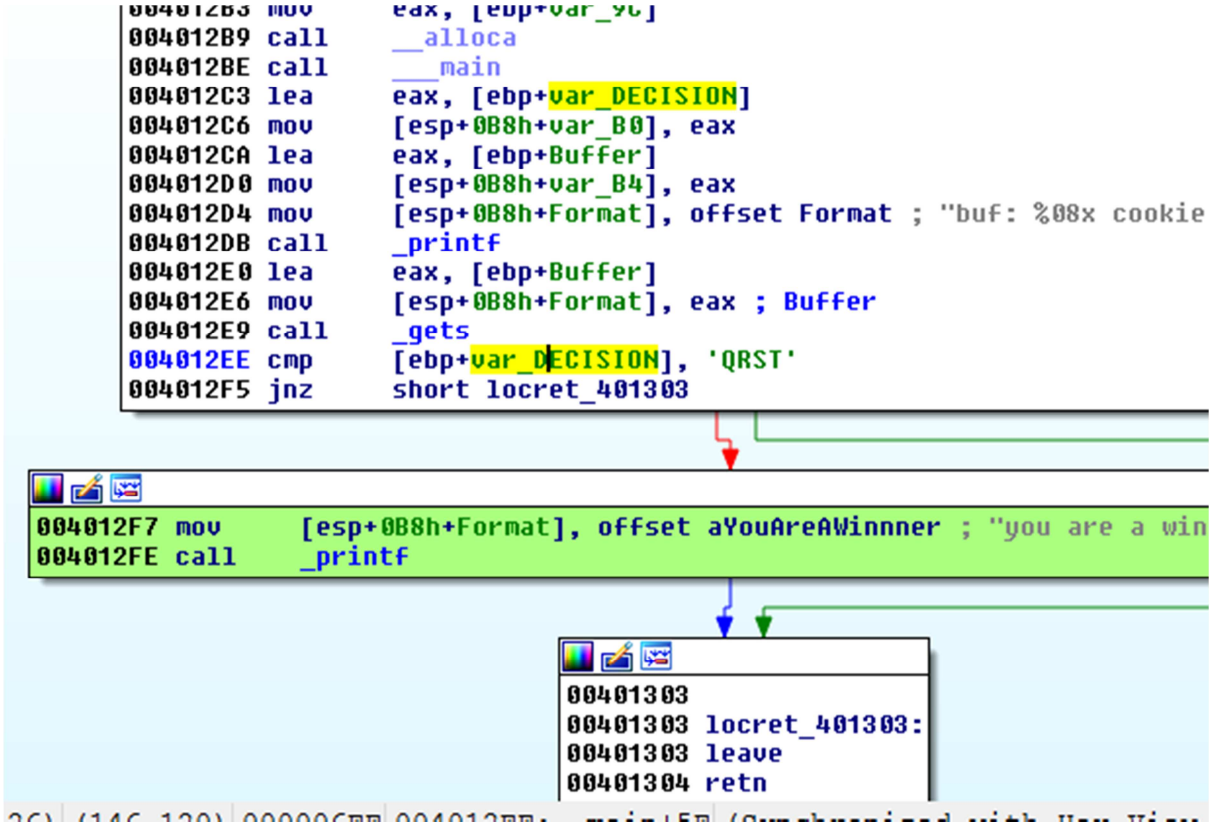
I can change it by the QRST letters which are the ASCII characters of that hexadecimal value.

Something that we can realize is that this compiler does not use the CANARY protection, because at the beginning of the function it should read the same from an address of the data section and be xored with EBP and stored just above the STORED EBP in the Stack and also reads it again just before finishing the function to call the CALL that checks it, none of that happens here, if we see static stack analysis, double clicking on any variable.



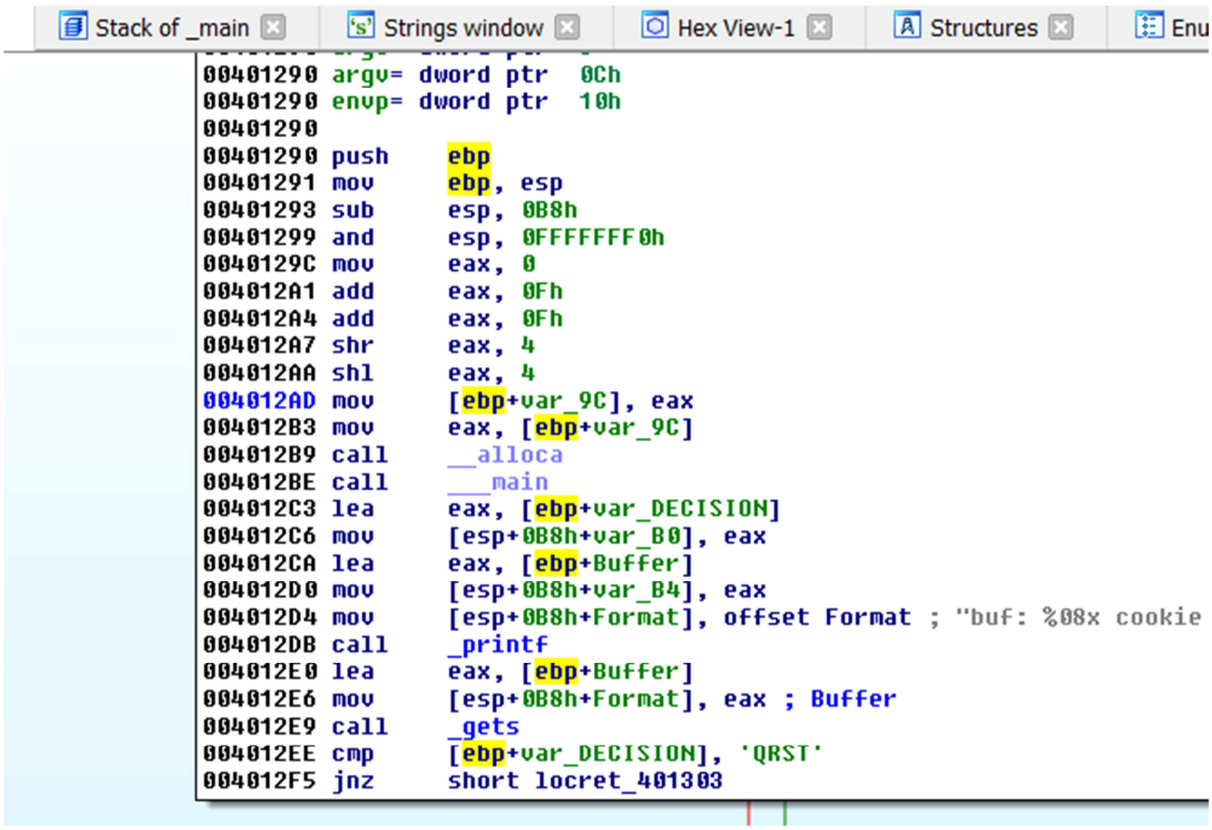
We see that the only thing in the stack, just above the STORED EBP, is the variable var_C that compares it with the QRST string to go to good boy, which is ruled out as the CANARY, since in this case it is a check which is the program's original code. A CANARY is not mixed with decisions of the original program code, it is something added by the compiler, external to the original code.

We can rename this variable to not confuse it with the CANARY as var_DECISION.



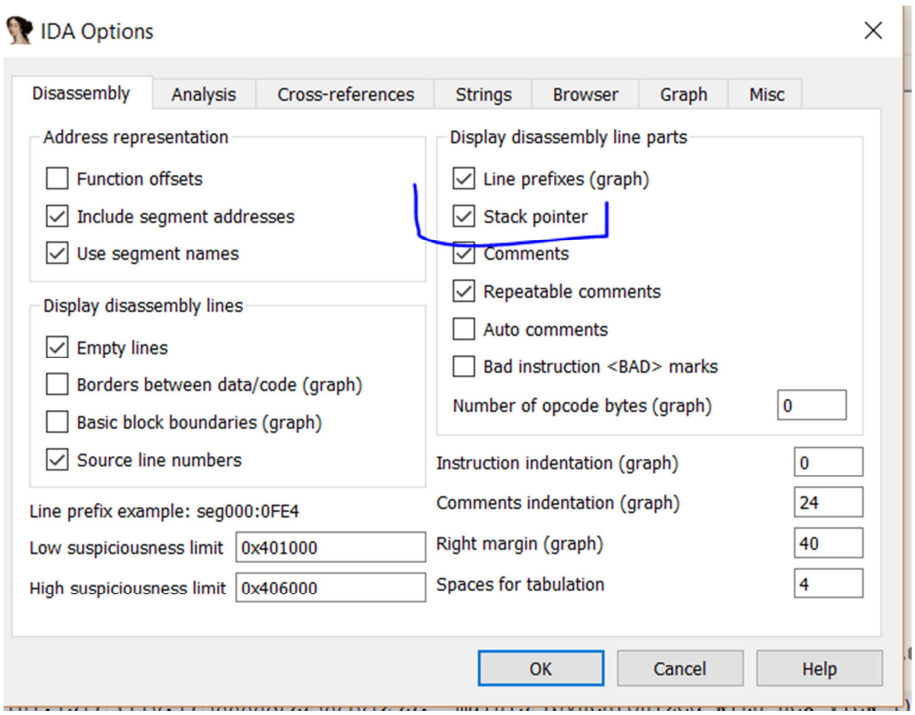
Which is used twice in the function, but can we really change the value of that variable to make the program go to Good Boy?

Those who look at this code the first thing that catches them is that there are variables and arguments related to EBP and there are others that are taken with ESP as a reference.



All that initial code is added by the compiler to set ESP just above the local variables and buffers, after the SUB ESP, 0B8, it fits with this, although it never varies much more than to align and round it, we could do the accounts, but if We do not want to complicate the life, we can debug it to see to what value ESP is left just to finish all this and to begin with the original code of the function.

The one who wants to debug can do it but we have another help from IDA that is very useful that is the variation of ESP from the beginning of the function that is taken as 0.

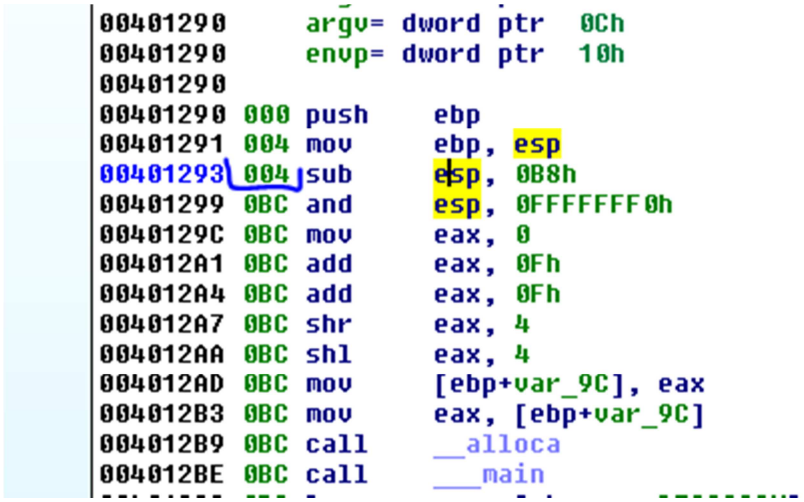


If we see the code now.

```
00401290     var_DECISION= dword ptr -0Ch
00401290     argc= dword ptr  8
00401290     argv= dword ptr  0Ch
00401290     envp= dword ptr  10h
00401290
00401290 000 push    ebp
00401291 004 mov     ebp, esp
00401293 004 sub     esp, 0B8h
00401299 0BC and     esp, 0FFFFFFF0h
0040129C 0BC mov     eax, 0
004012A1 0BC add     eax, 0Fh
004012A4 0BC add     eax, 0Fh
004012A7 0BC shr     eax, 4
004012AA 0BC shl     eax, 4
004012AD 0BC mov     [ebp+var_9C], eax
004012B3 0BC mov     eax, [ebp+var_9C]
004012B9 0BC call    __alloca
004012BE 0BC call    ___main
004012C3 0BC lea     eax, [ebp+var_DECISION]
004012C6 0BC mov     [esp+0B8h+var_B0], eax
004012CA 0BC lea     eax, [ebp+Buffer]
004012D0 0BC mov     [esp+0B8h+var_B4], eax
004012D4 0BC mov     [esp+0B8h+Format], offset Format ; "buf: %08x co
004012DB 0BC call    _printf
```

We see the influence of each instruction on the stack from scratch which is the start, after executing the PUSH EBP, the stack decreases by 4, so the second line has 004 on the right.

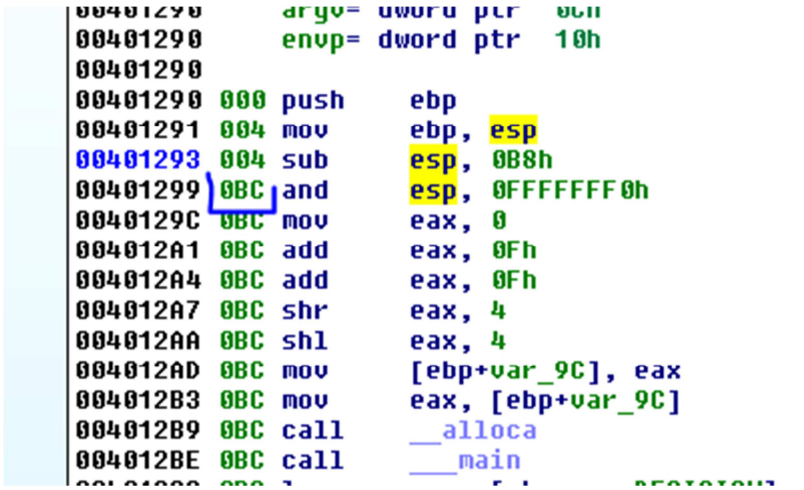
The second instruction is MOV EBP, ESP which does not change the stack because it is only a mov and therefore ESP remains the same, which changes is EBP.



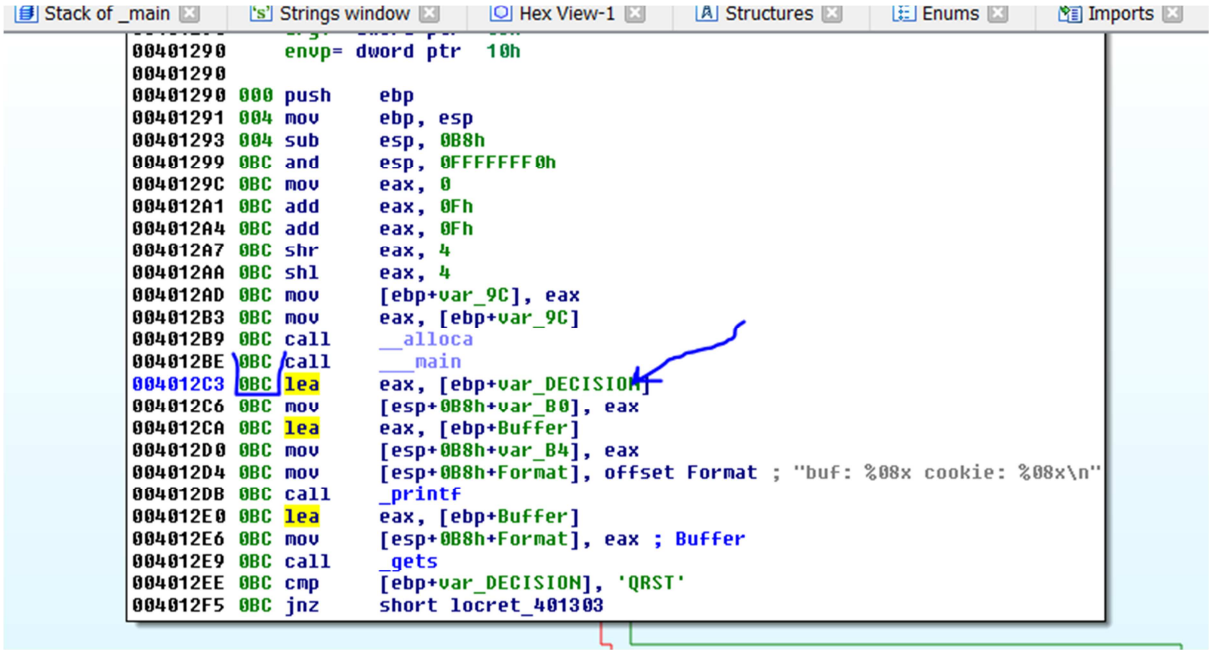
That's why the third line has the same 004 because it did not change ESP.

Recall that ESP and EBP are the same and EBP will be from here the reference remaining both 4 of the ESP of the beginning.

The next line subtracts 0xB8 from ESP so it is 0xBC from the start and as EBP was at 4 the difference between EBP and ESP will be just 0xB8.

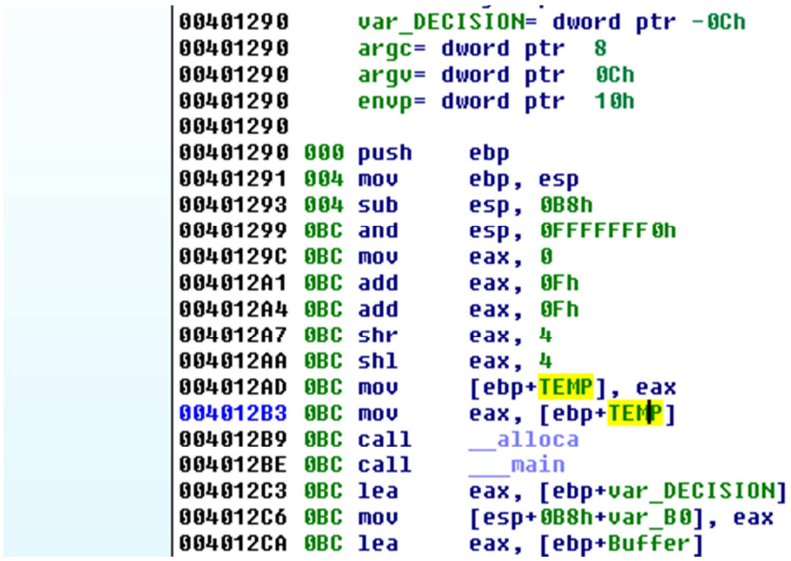


We see that then the distance does not change even when passing the ALLOCA and CALL_MAIN CALLs that are added by the processor, so we can conclude that it does not affect all that, and that the distance between EBP and ESP is 0xb8 which is the space for local variables and buffers.

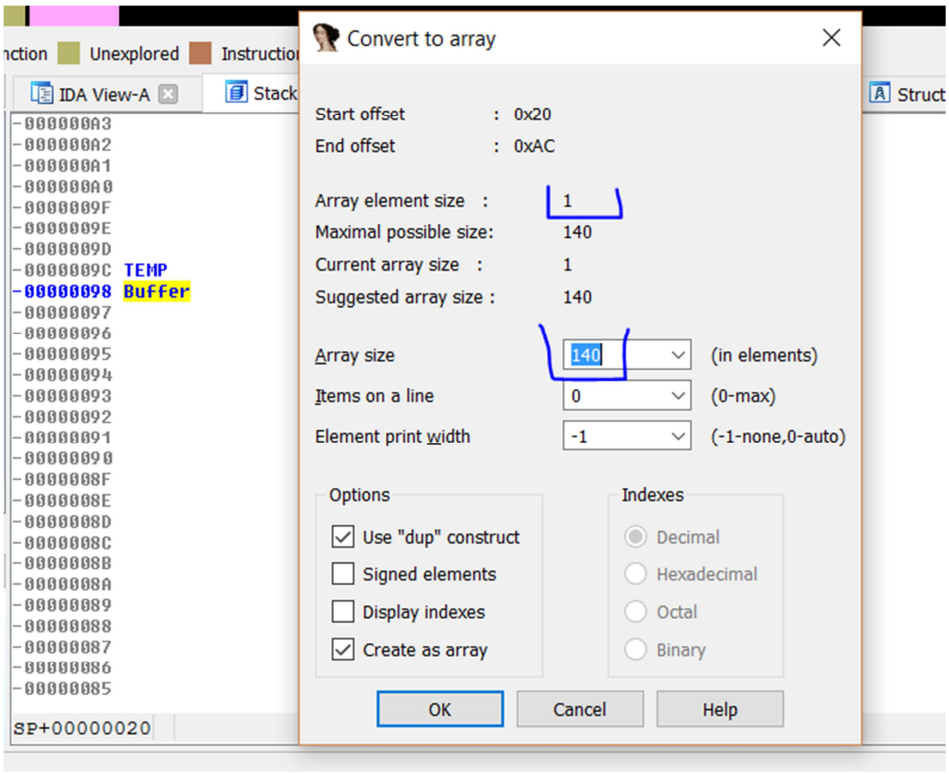


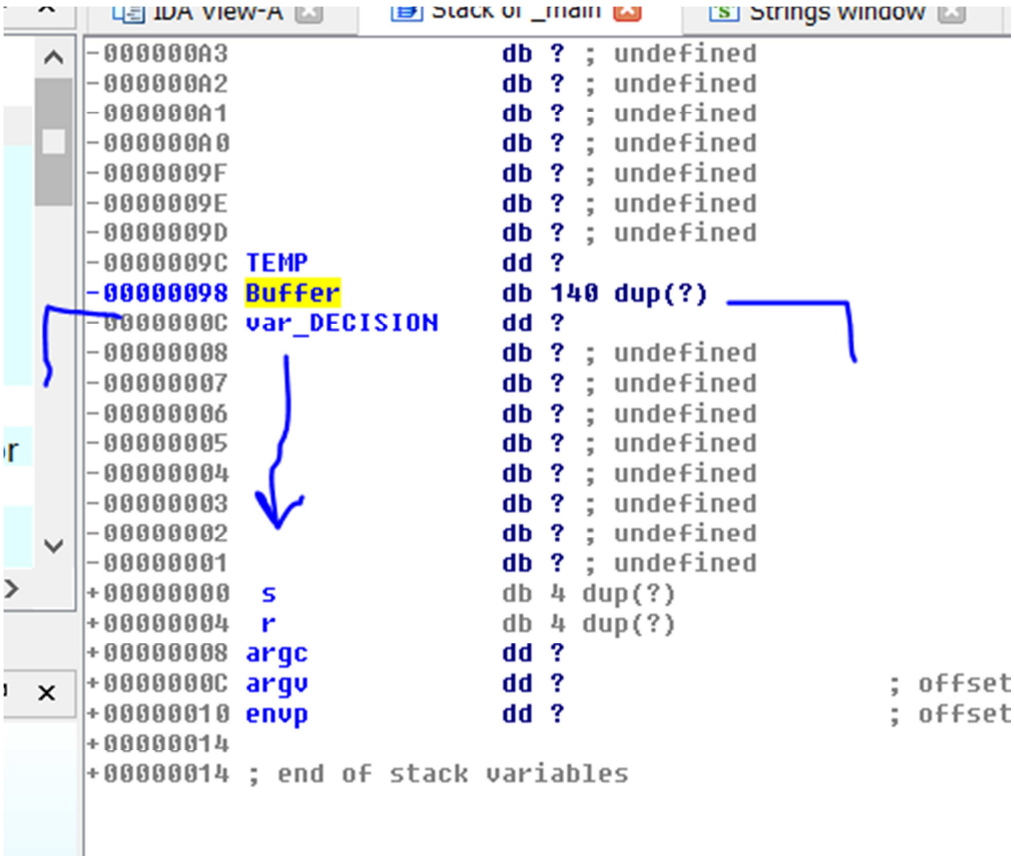
There, the code of the function itself begins, and the distance from ESP to the start was 0xBC and 0xB8 is the zone reserved for variables and buffers.

As the var_9c variable is a temporary variable created by the compiler and not used more we will rename it as TEMP.



Under TEMP we have the Buffer, right click - ARRAY we see that the length is 140 decimal by the long 1 byte of each element, so the buffer length is 140 decimal.

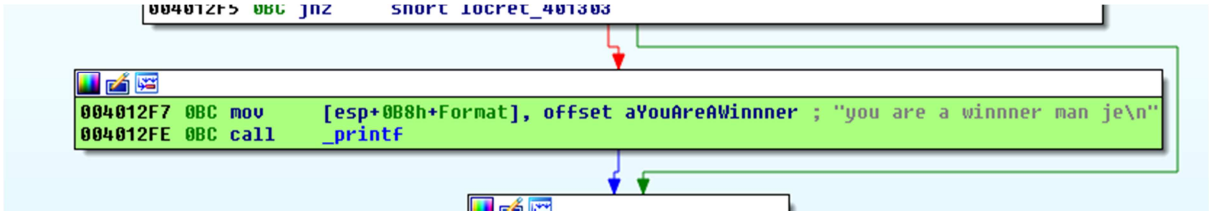




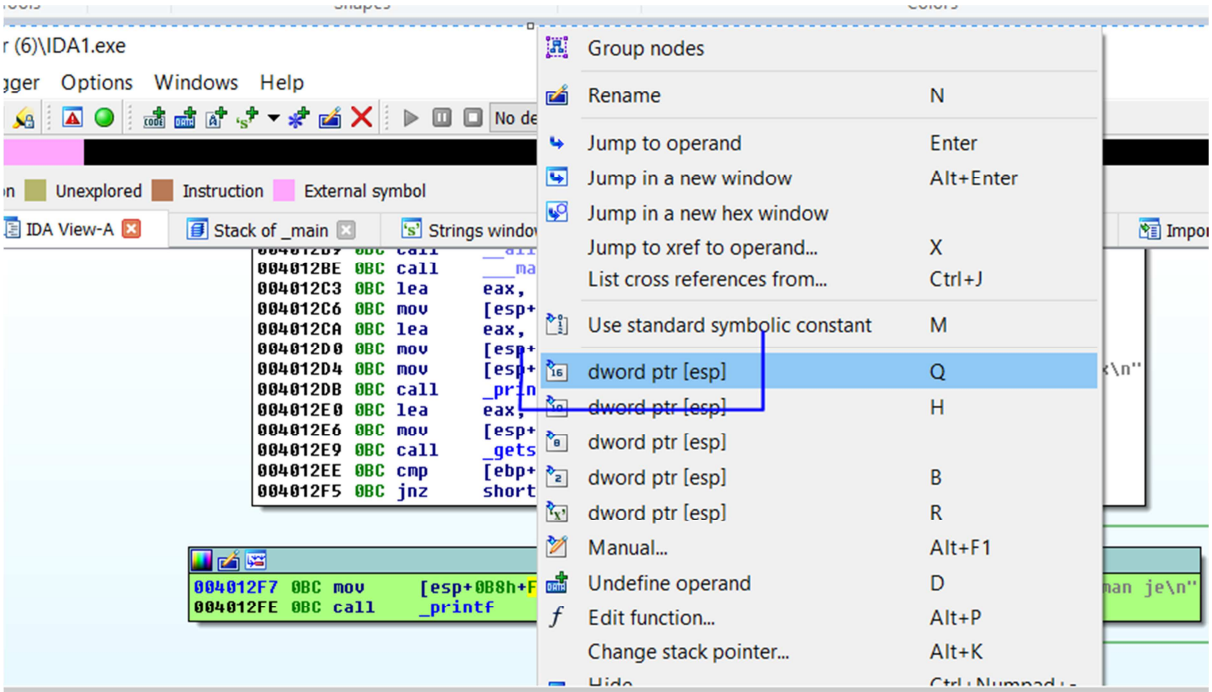
If we were able to overflow the Buffer by copying over 140 bytes, it would have a Buffer Overflow vulnerability, exploiting which one could step on the var_DECISION variable and if we could write further down, we would get to STORED EBP and RETURN ADDRESS depending on how much we can copy.

Let's continue with the reversing, the highest point of the compilation is the way you pass the arguments to the functions, instead of using PUSH to save the arguments in the stack, save them directly with MOV, we will see this.

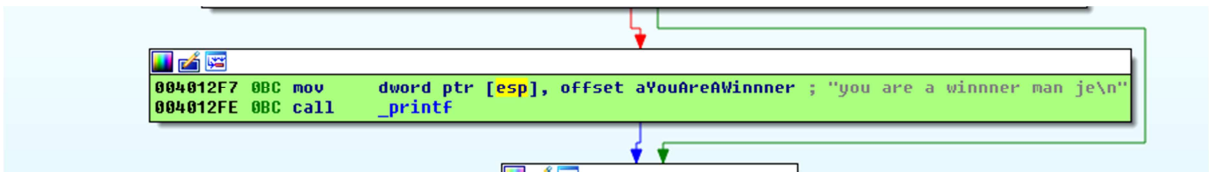
There, it is clear, in this case, **printf** has a single argument that is the address to the string "You are a winner je." And no other argument.



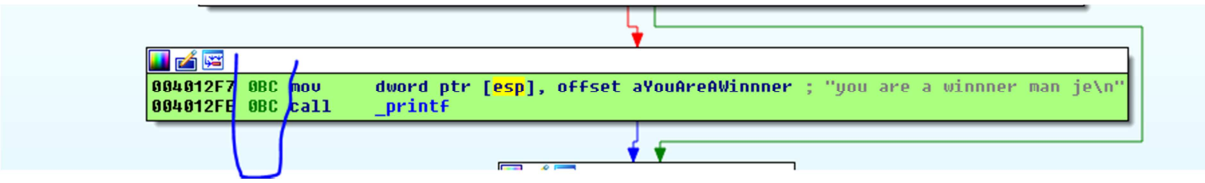
We see that it saves that address because it uses the word OFFSET in front, and saves it in the stack, but where? The notation does not help us much but if we right click, we will see some alternative notation.



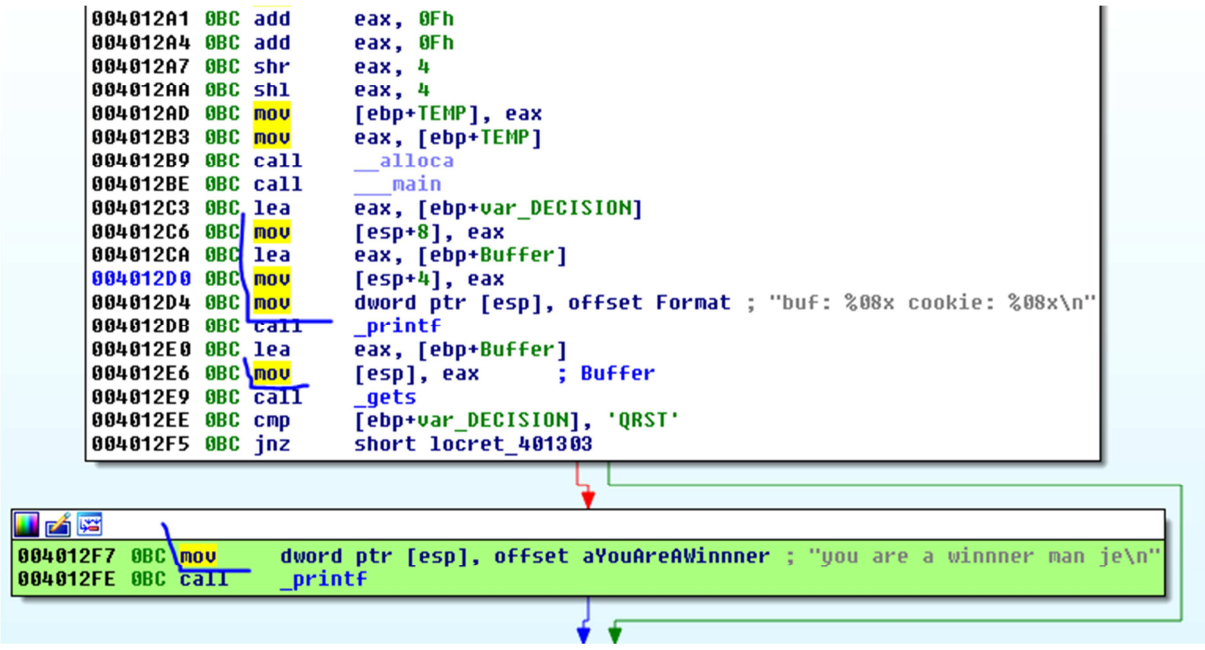
If we change it for this.



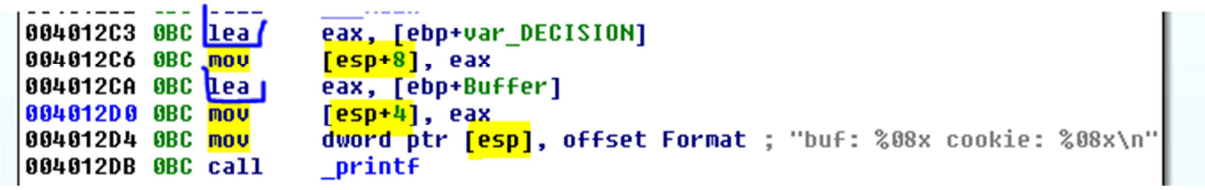
We see that what is really doing is to put in the content of ESP (which is the top position of the stack) the address of the string to use as argument, i.e. this program instead of making PUSH to the top position of the stack, Moves the arguments to the stack positions with MOV, and logically the PUSH would change the value of ESP, whereas a MOV does not. It can be seen in the BC value just before the function.



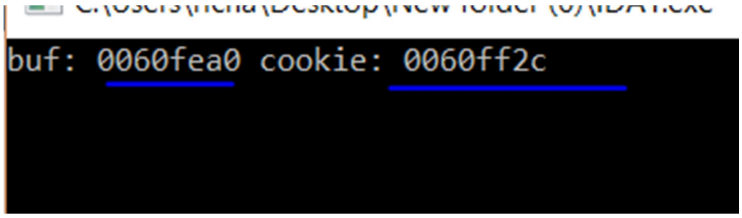
The same goes for all other APIs. If we apply the same criterion only in those that can be arguments of APIs, right clicking and changing by alternative notation, we are much more understandable code.

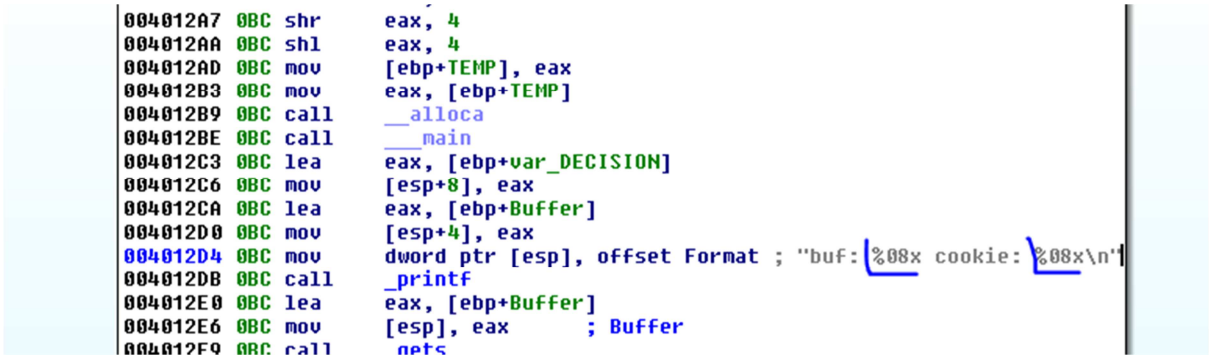


We see the first **printf** which has three arguments because it makes format string replacing in the string "buf:% 08x cookie:% 08x \ n".], For the two upper arguments.

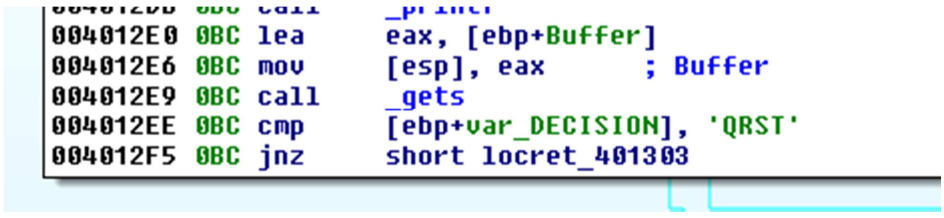


The first argument is the address of the var_DECISION variable that is obtained with the LEA. It moves it to EAX and saves it in the contents of ESP+8. The second argument is the address of the Buffer variable that gets it with the LEA. It moves it to EAX and saves it to the contents of ESP+4 as the next argument and the third one saves it in the ESP content and it will be the address of the string, in this case, with the format. If you run it outside IDA, we see that it is the console print of the addresses of both variables, since it replaces in the original string format string, by the hexadecimal value of both addresses because it uses %x which is the conversion to print the hex value.





Then it continues to call the gets function which enters characters by keyboard without any limit, so you can write more than 140 characters without problem.



Here too, it passes as an argument to the content of ESP, the address of the Buffer that is where it will write.

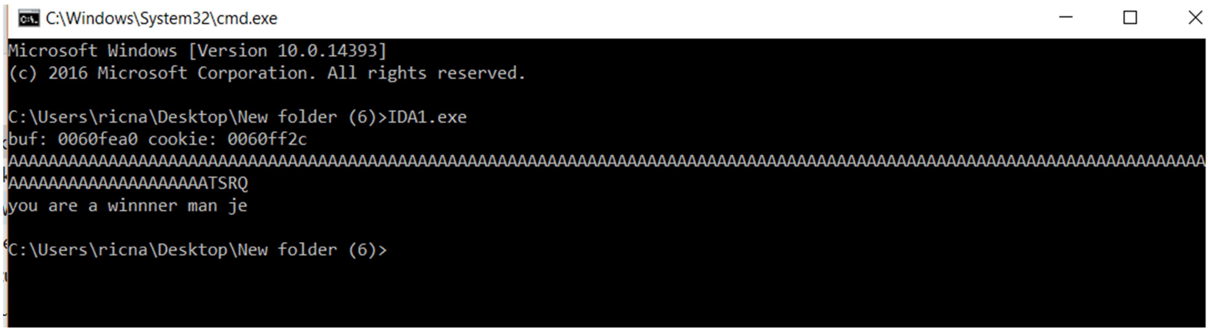
So we know that if I write for example 140 Aes and then TSQR since it must be upside down by the little endian, the program should jump to good boy

since we will step on the var_DECISION variable that is just below the Buffer, with the QRST value. Let's try first by hand, before doing the script.

Print the string in IDA and copy it to the clipboard, paste it here.

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAATSRQ

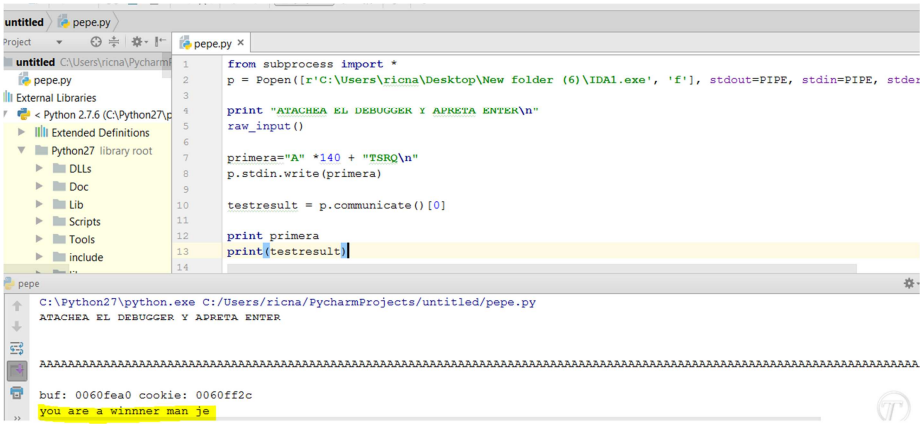
I run it in a console but it will close and I will not see the string if it leaves, when it is waiting I hit the string and ...



The script is very simple. It is also the same as the previous one. In this case, it does not have two inputs per stdin, but only one.

```
from subprocess import *
p = Popen([r'C:\Users\ricna\Desktop\New folder (6)\IDA1.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="A" *140 + "TSRQ\n"
p.stdin.write(primera)

testresult = p.communicate()[0]

print primera
```

print(testresult)



There you see the result, many will not have been able to do it by the way to pass the arguments, but now it will be easier to do exercise 2 since it is very similar and compiled similarly, but you already know the trick.

The next exercise is called IDA2.exe

Ricardo Narvaja
Translated by: @IvinsonCLS

.