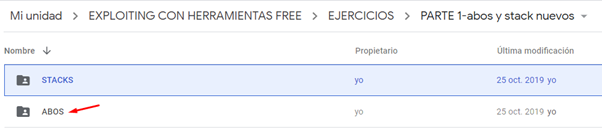
REVERSING & EXPLOITING USING FREE TOOLS (PART 5)

We will start with the analysis of stack 4, from this exercise we will do on with each tool to delve deeper into its use.

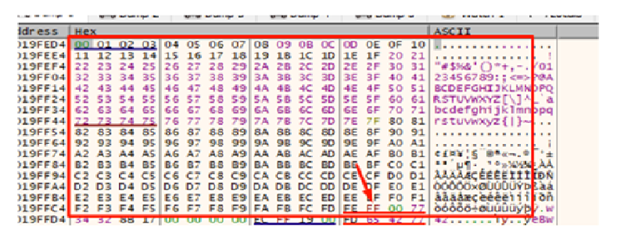
This exercise we will do it in IDA Free, the next exercises are the ABOS (Advanced Buffer Overflows) that are in the ABOS Folder, on these we will have to pop a calculator, a notepad or any application we want.



ABOS exercises are also 4, so we will do number 1 with Radare2, the number 2 in GHIDRA, number 3 in IDA Free and finally number 4 in Radare2 again, that will be the process we will use.

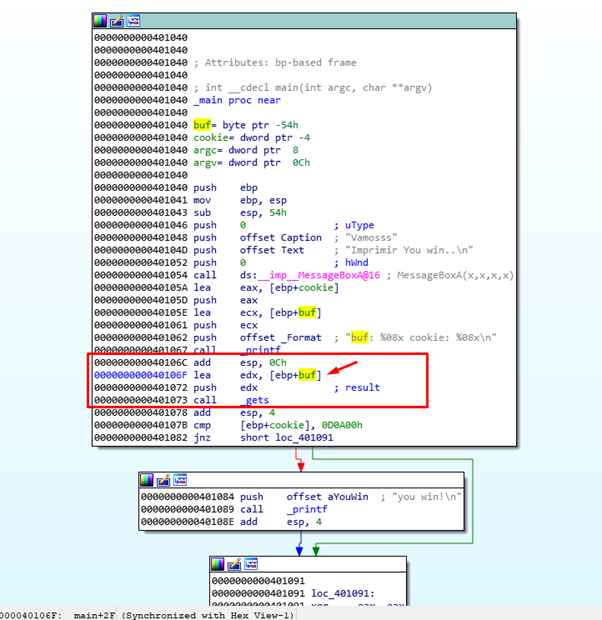
## STACK4 in IDA FREE

In part 4 we saw the topic of invalid characters, we’ve seen that in previous parts all of our stacks exercises we have two invalid characters **0x1a** and **0xa**

****

We can see that inside of the String where we try all the characters, the invalid characters for stacks exercises are as we said **0x1a** and **0xa**, the others can be sent with no problem.

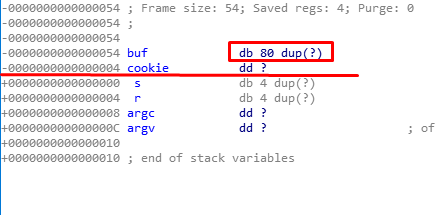
### Static Analysis of Stack4 with IDA Free



We can see that address of the buffer **buf**, is the parameter of the function gets, so there will store whatever we write until I press ENTER to finish typing.

Below we can see the comparison of the variable cookie with the constant value **0x0d0a00**, as the value cookie does not change ever the value inside of the function, is obvious that we couldn’t get to the **YOU WIN** part if the program works correctly, because cookie can never have that value. Of course, the only way to reach that point is exploiting the buffer overflow in the stack, as we saw in previous exercises.

We can see buffer **buf** length in the static representation of the stack, double clicking in any of the variables:

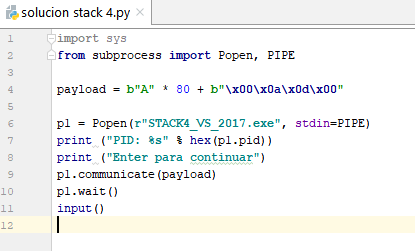


We can see that **buf** length is 80 bytes, so if we send to the program:

*payload = 80\* ’A’ + “BBBB”*

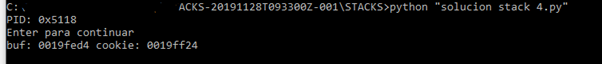
We would change the cookie with the string “BBBB” that it is 0x42424242, the problem in here is that to get the point of the **YOU WIN** we need cookie with the value **0x0d0a00** so we should give the payload:

*payload = 80\* ‘A’ + “\x00\x0a\x0d\x00”*

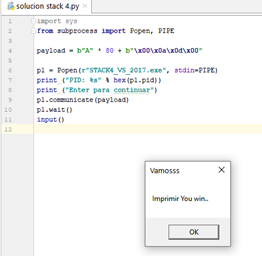
**

Let’s gonna run it from pycharm or from Windows command prompt and let’s see if we get the YOU WIN Point.

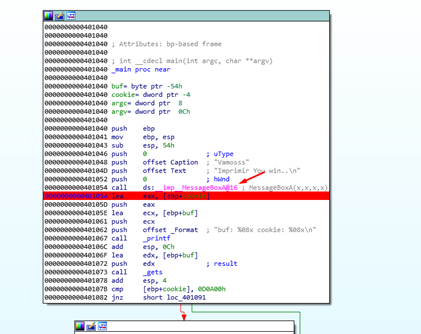
As we can see it does not print YOU WIN and program finishes:



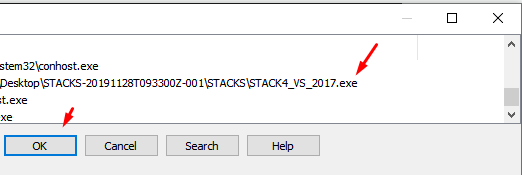
What we’ll do is to debug, this time we will do it with IDA Free, so we will run again the script from the Windows command prompt and we will get the MessageBoxA.



In the IDA Free we opened, first we have to set a breakpoint after the call to MessageBoxA:

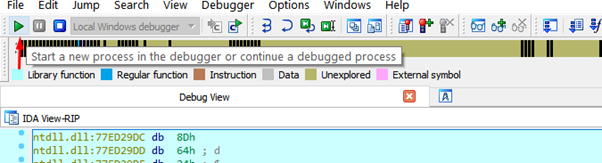


Now in IDA Free we go to Debugger - Attach to process

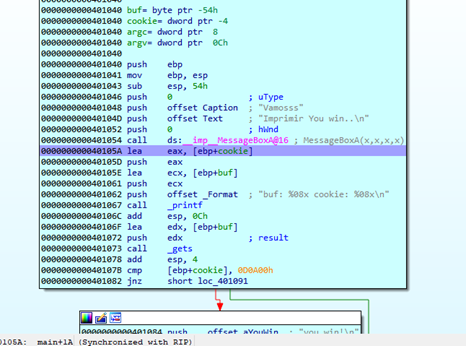


In the process list we search for STACK4 and accept.

When it stops we press in “play” to run or we can use “f9”

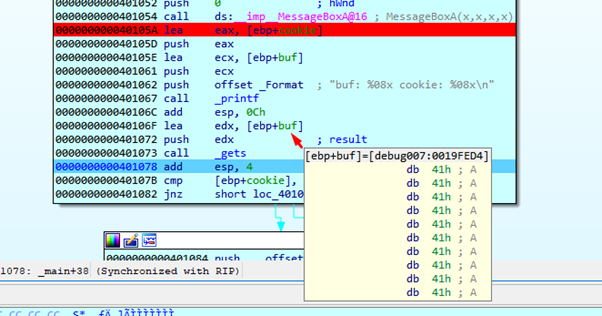


Now it will be running, we accept the MessageBox.

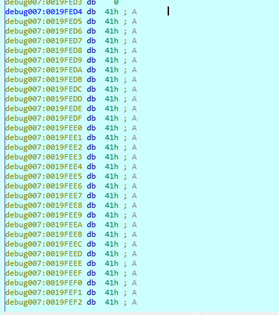


As we can see, there it stops at the breakpoint, we trace with F8 (Step Over) and we go over the function “gets”, so we will see how it manages our payload.

Hovering the mouse over **buf** without clicking, we’ll see our payload.



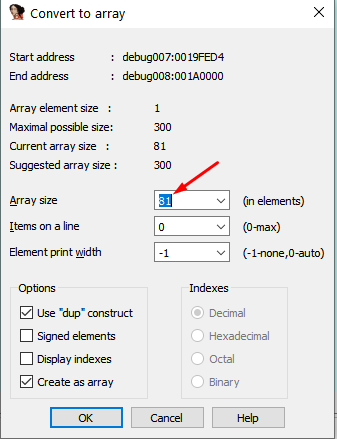
If I double click in **buf**.



If we want to see the length of an string without counting one by one, I press key “A” in the first 41h to transform all the buffer in a string.



We can see that the last 0 it’s part of the string (because in C programming, an string is a serie of characters ended with a NULL byte), now if I do right click - ARRAY there it will show us the length of 81 including the last 0.

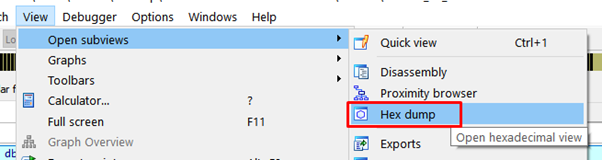


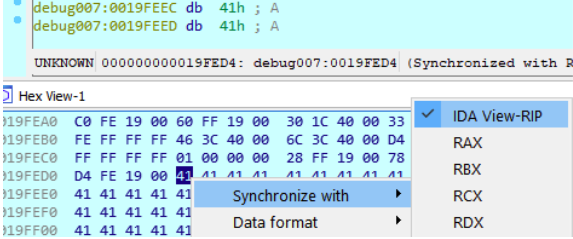
We just press Cancel as we did it to see the length, we see that our 80 ‘A’s entered, and the next zero what it continues is different:



We can see that the buffer stopped at the byte 0x0a.

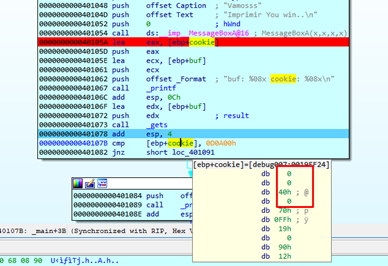
To see the part of payload that was inserted, I have the HEX-VIEW window too.



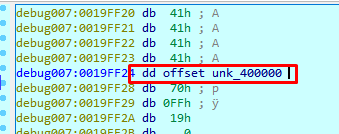


We can choose to synchronize the content with the memory pointed by some registers, in this case that synchronize with IDA VIEW-RIP that is the upper window, once is chosen, I will see my data in a similar way to OLLYDBG DUMP view, as a column with memory addresses at left and the content at right.

So as we can see, not all the payload we sent entered, also if we see the cookie value, we know that are the 4 bytes after the ‘A’s, but if I hover the mouse over cookie variable:



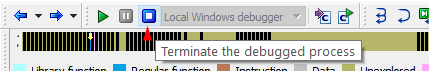
To change the view of non-defined bytes as right now, to show variable as a DWORD, a double click on it and I press “D” until the dword, it will change from **db** to **dw** and finally to **dd** (BYTE, WORD and DWORD).



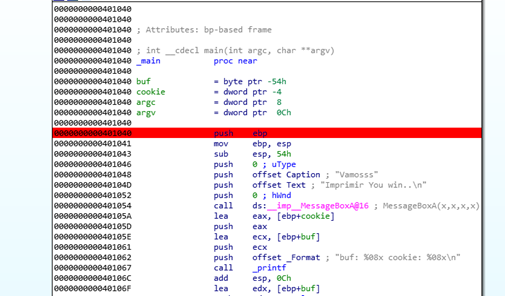
So the value of the variable **cookie** is **0x400000** and as it’s not equals to **0x0d0a00** it will not go to **YOU WIN**.

As we saw the problem is that the character **0x0a** is invalid, and it can’t be copied to buffer because it cuts our payload and it’s completely necessary to compare correctly and jump to the **YOU WIN**, now that we see it’s not possible to write the payload, which is the other way to reach the **YOU WIN** point?

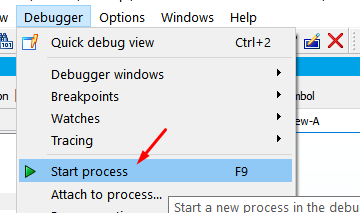
I stop the debugger.



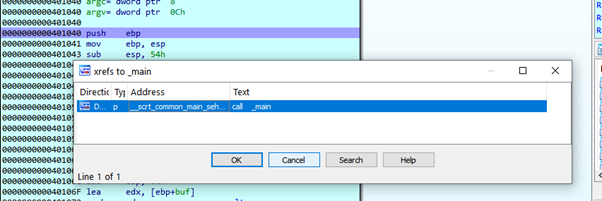
I put a breakpoint in the first instruction of main function.

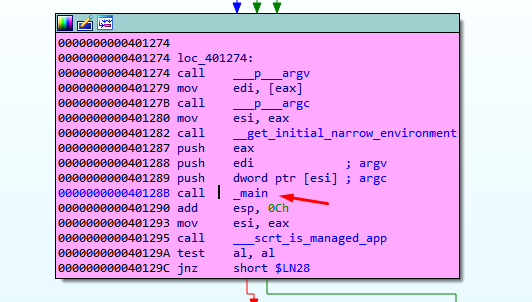


And this time I’m going to run it without script, directly from the beginning and I’m going to type the data in the prompt manually.



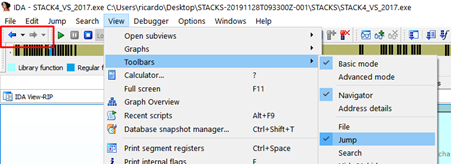
Once that stops at breakpoint we press X, so we can see where the function main is call from.

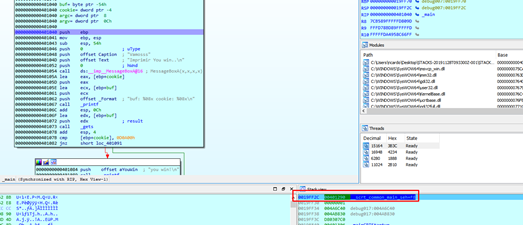




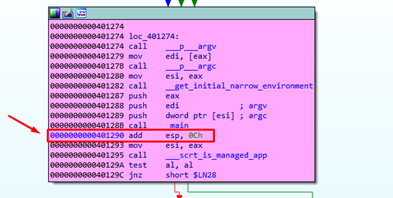
The thing is that when that call is called and entered in main after have being pushed the arguments to the stack, RETURN ADDRESS is stored in the first place of the stack. When main finishes and gets RETN instruction, the function will obtain that **RETURN ADDRESS = 0x401290** where it will return, just after the call instruction we see above.

One of the most comfortable toolbars in IDA is **JUMP**, it can be added and with it we can go back and forward, to return to main function for example where we were:

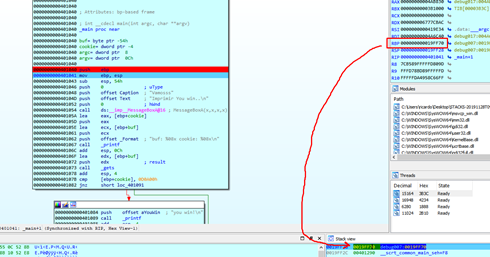




There we see that RETURN ADDRESS is equal to **0x401290**, so when main function finishes it return to here:

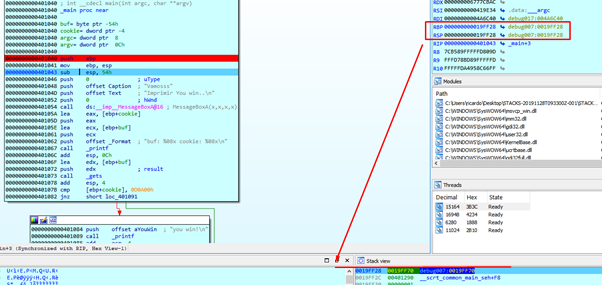


Once we enter in main function, the PUSH EBP stores on the stack the EBP of the caller function (previous function) that called main so we call it STORED EBP that is the EBP of the parent function.



If I press F8 I see how it stored on the stack, then ESP value is copied to EBP.

It’s created here what we called the HORIZON, I press F8 to see it:



Now that EBP is set to the HORIZON value that will have in the function, ESP must go up to allocate space for local variables, **cookie** and **buf**, this is done with the instruction **SUB ESP, 0x54**.

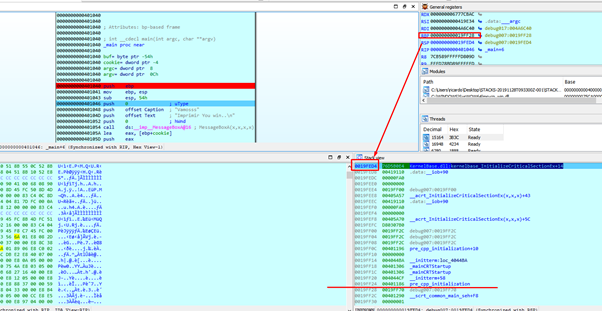
Just to clarify and avoid misunderstood, we say “ESP must go up” but this is visually in the debugger or disassembler, because ESP will be above of EBP. But memory is shown in opposite the low memory values are shown up in the stack view so increment the stack will go lower in memory, so really ESP value will be lower than EBP.

So when we talk about going up on the stack is just something about visual representation, in the same way with INCREMENTING.

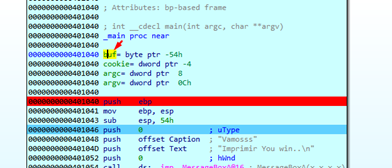
EBP saves the HORIZON address and ESP will be above 0x54 bytes of space for local variables:

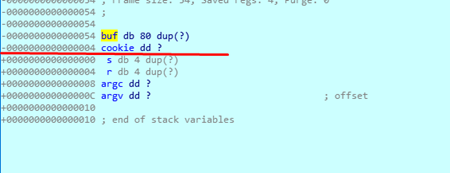


So ESP will be 84 bytes above, from these 84 bytes, 80 will be **buf** and 4 for **cookie**.



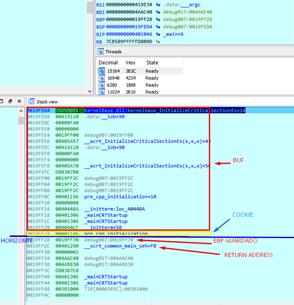
In the static analysis of the stack of IDA we can see the same, to go in the debugger we have to do double click in the definition of variables, in any of them:





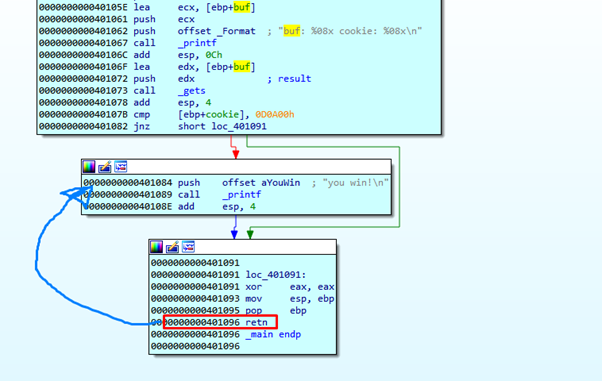
So IDA show us statically the same stack representation. The HORIZON is in ZERO and I get the same that I’ve seen debugging

|  |  |  |
| --- | --- | --- |
| **buf** | 80 allocated bytes |  |
| **cookie** | 4 allocated bytes |  |
| **-** | HORIZON | 0x19FF28 (in my system) |
| **s** | STORED EBP | 0x19FF70 (in my system) |
| **r** | Return Address | 0x401290 |
| **argc** |  |  |
| **argv** |  |  |



**EBP GUARDADO = STORED EBP**

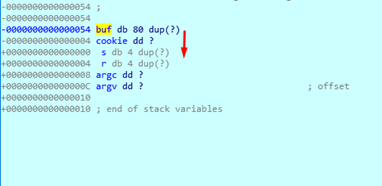
Well nuts and bolts here is that RETURN ADDRESS (the place where program will return after finishing main function) is right down under **buf**, so with little bit of precision we could modify this value where to return, in this case we will return to the address that will print **YOU WIN**, we will force to return where we want.



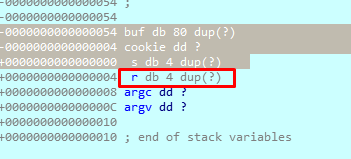
There in the **RETN** function will finish and should return to **0x401290** as we saw previously, this is the value RETURN ADDRESS that was stored when we entered the function, but we will modify that value overflowing **buf**, we will set the value **0x401084** and when executing **RETN** we will return to the block where it prints **YOU WIN**.

Now we will calculate the distance to modify the RETURN ADDRESS, let’s gonna see the static stack representation again.

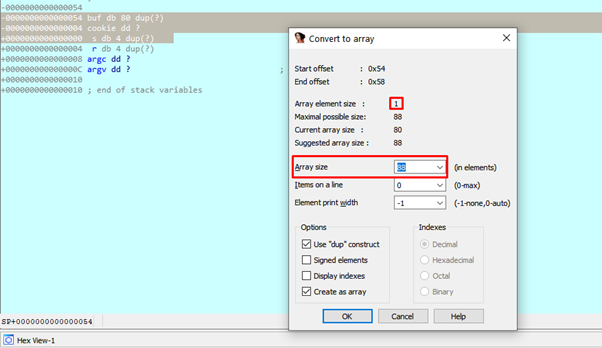
We know that it will start copying from **buf** down:



We will mark from **buf** until RETURN ADDRESS (r)



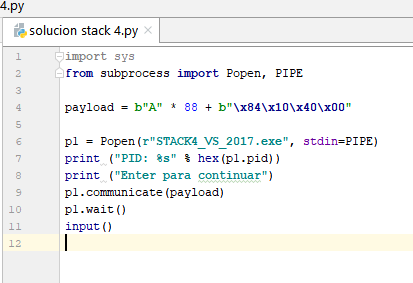
Now in the marked zone we just do right click - ARRAY, just to see the number of bytes to copy until that zone.



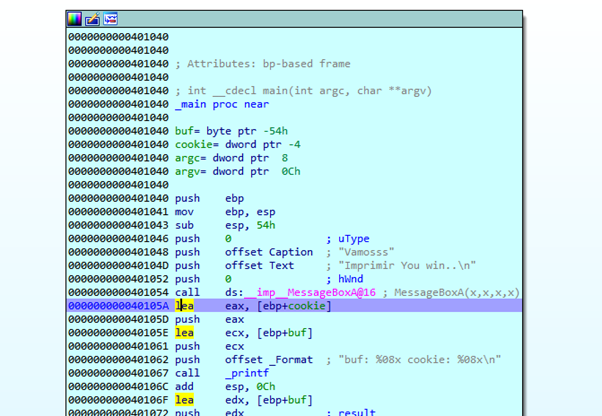
It would be 88 bytes, and then it would be the 4 bytes to modify RETURN ADDRESS, so our payload should be something like:

payload = **b"A"** \* 88 + **b"\x84\x10\x40\x00"**

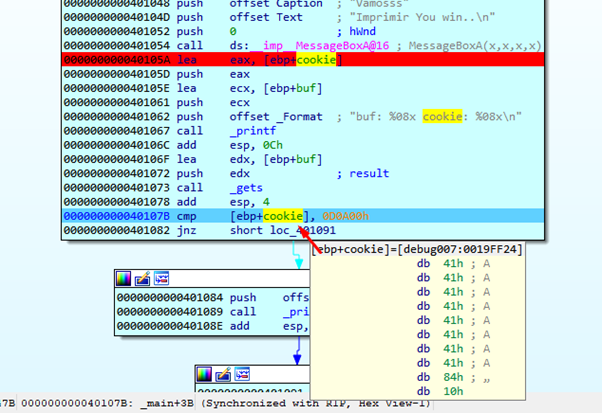
Being **"\x84\x10\x40\x00**" the address where I want to jump **0x401084.**



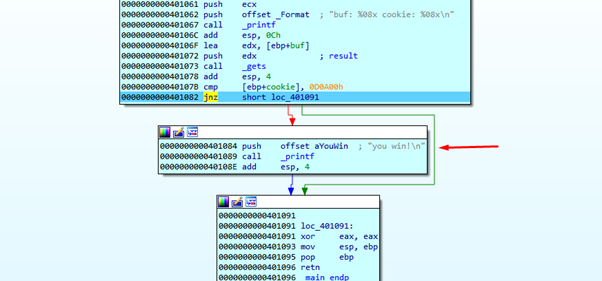
Stop the debugger, and run the script attaching IDA with the breakpoint set after the MessageBoxA, once we are attached press F9, accept the MessageBoxA and debugger will stop at breakpoint:



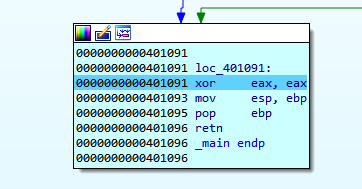
I go tracing the program with F8, and I step over gets and I go to the comparison with the cookie.



We can see that the cookie will not be **0x0d0a00**, in this case it will be **0x41414141** (I can go with double click and press D until change it to **dd** to see it as a DWORD), and as those values are not the same, it will not go to YOU WIN.

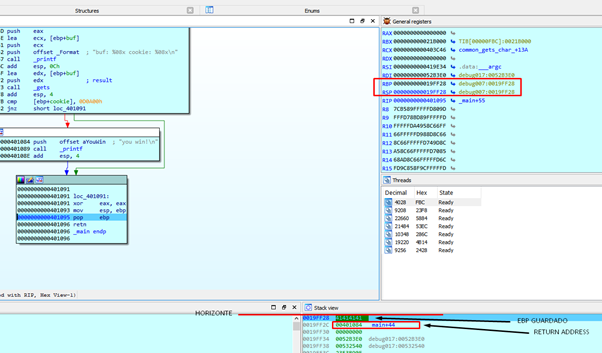


As instruction JNZ jumps if are not equal, and this is TRUE as values are not equal, it will jump to 0x401091 following the green arrow of the result TRUE.



We get the EIPLOGUE of the function, EAX will return with value zero, that would be the return value of the function if it has one (all the function returns something in EAX, also void functions).

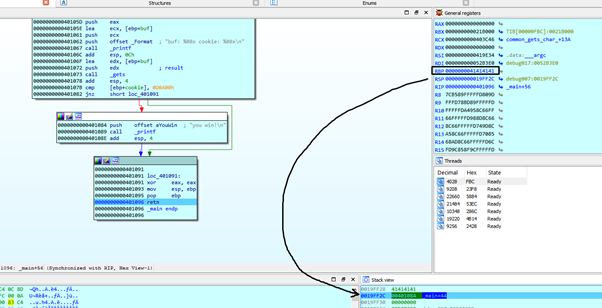
Then ESP return to the value of HORIZON freeing the zone of variables **cookie** and **buf** but without deleting the content, it just return to the value of HORIZON again and there is the RETURN ADDRESS right down after EBP STORED, and it will return where that value points.



So we have the ESP and EBP values just in the HORIZON, down is **s** the EBP STORED and **r** the **RETURN ADDRESS.**

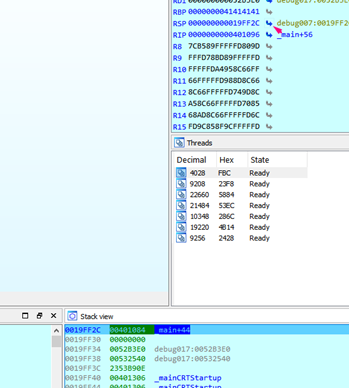
We see that our payload modified **EBP STORED** with 0x41414141 and the **RETURN ADDRESS** that it was **0x401290** now it’s **0x401084**.

If I press F7 (STEP INTO) POP EBP will copy into EBP the EBP STORED of the parent function, problem is that it was modified with 0x41414141 when we overflowed the stack, a different value from PROLOGUE stored the real value with PUSH EBP.

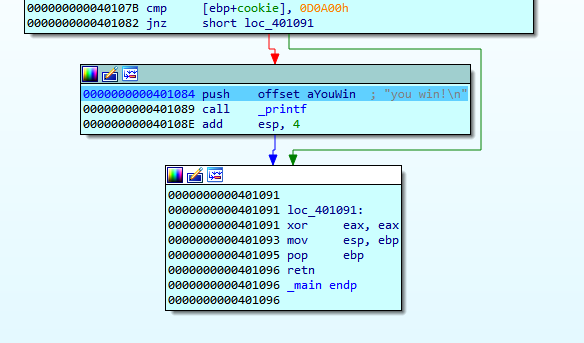


So EBP will be 0x41414141 and the stack now will be aligned to do the RETN and return to the address **0x401084.**

If you want the stack to show the upper value, you have to click on stack window and click in the arrow right of RSP:



Now we see that the first value of the stack is 0x401084, we press F7 to return.



We can see that get the part of YOU WIN without a correct cookie, forcing the program to return to that address modifying the RETURN ADDRESS.

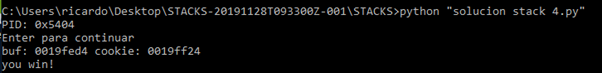
This is correct and is a valid solution, the problem is that EBP will have 0x41414141 so if we continue tracing, it will crash the program.

Continue tracing with F8 to see it.

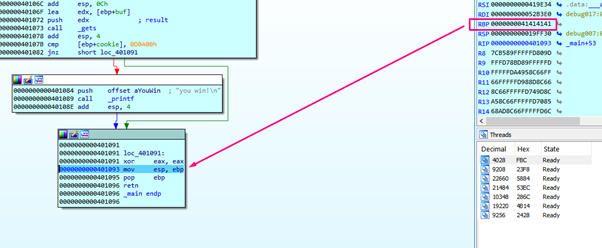
We see that if we execute the program from PYCHARM, once it goes through printf it does not show the **YOU WIN** in the python console.



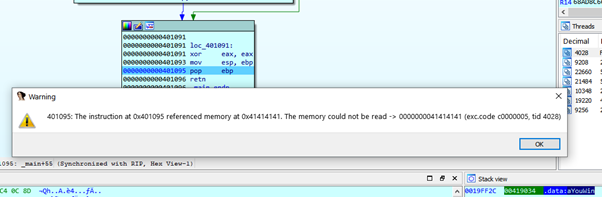
This is because the way that PYCHARM uses stdin and stdout, but if I run it from a windows command prompt even when program crash, it will show the **YOU WIN**.



We will continue tracing to see where it breaks:



We see that when the program execute the EPILOGUE again but with EBP equals to 0x41414141 and copies that value to ESP it will crash trying to to access to the stack value with POP EBP where an address is not allocated:



For people just starting it is a perfect valid solution, it doesn’t matter that program crashes, the solution that I will show without executing code in the stack and without hardcoding in any address, it requires more knowledge and is harder, but for advanced levels.



I’m going to trace a bit, but I will not explain how I found this as it would take longer:

**import sys**

**from subprocess import Popen, PIPE**

**payload = b"A" \* 84 + b"\x58\x97\x41\x00"+ b"\xd5\x12\x40\x00" + b"GGGG" + b"\xc0\x18\x40\x00" + 8 \* b"A" + b"\x84\x10\x40\x00" + (b"\xb3\x46\x40\x00")\*10**

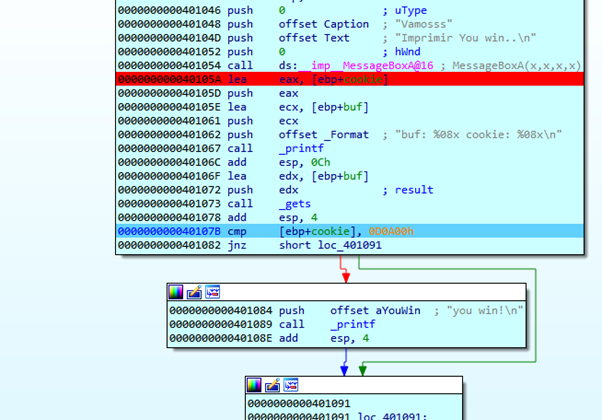
**p1 = Popen(r"STACK4\_VS\_2017.exe", stdin=PIPE)**

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

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

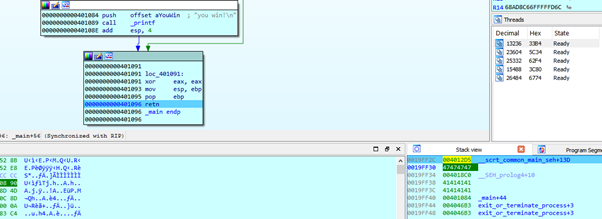
**p1.communicate(payload)**

I run it from a windows console and I attach it when MessageBoxA appears:



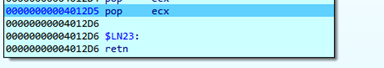
As cookie is not the same, it will jump to **0x401091**

And I get the instruction RETN.



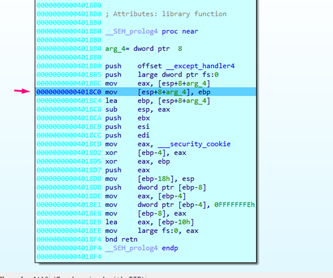
I jump to **0x4012d5** that is code from the executable so I don’t have any problem with ASLR, DEP or anything.

This is going to execute a POP ECX-RET, this will accommodate the stack.





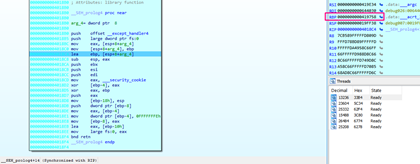
We see that my payload it contains a “GGGG” that it is 0x47474747 that it’s not an important value that is moved to ECX, and then the next RET makes the program to jump to the next value of my payload that is **0x4018c0**.



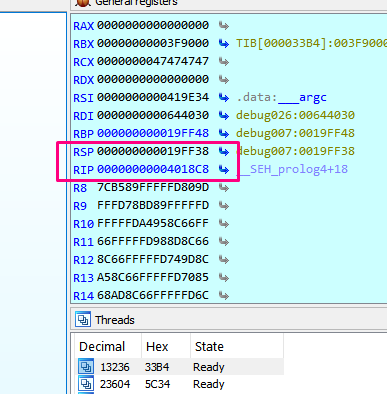
This is the function where it jumped SEH\_Prolog, it makes the same work than the PROLOGUE of the function we’ve seen, among other things it matches the values of ESP and EBP to move later ESP up the value given as argument.

What I care about here is the **LEA EBP, [ESP + XXX]** that will store again EBP to a value on the stack, fixing the problem that could make crash the program, if we do it correctly.

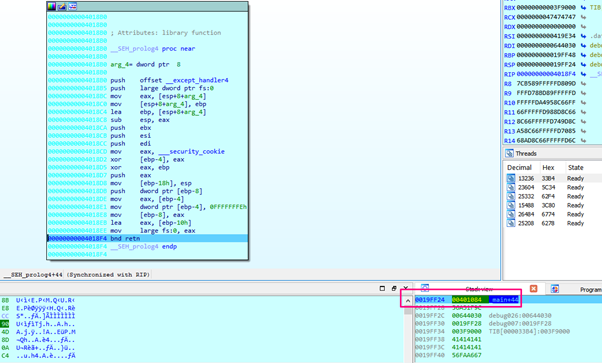
If I trace I see that EBP before of LEA is **0x419758**.



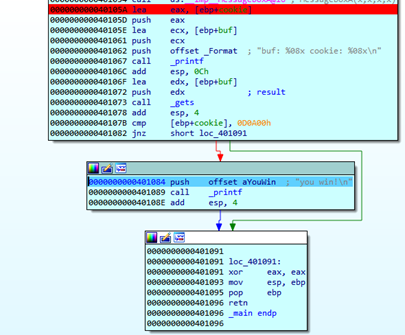
And after the LEA, in my system it is 0x19ff38.



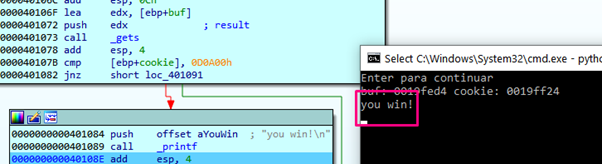
So EBP returns to a stack value, the one we want and as it depends on ESP value, the previous POP ECX - RET just moved right where I need, we will see, just continue tracing:

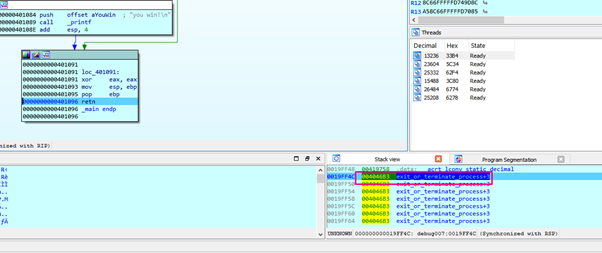


Once we get the RET instruction it will jump to **0x401084**, that is the block of YOU WIN, but now with correct EBP, let’s continue tracing:

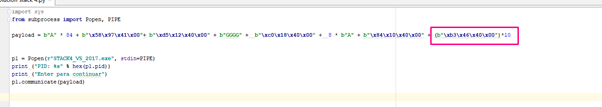


There it is, I step over with F8, and we see the YOU WIN, now we just need that finishes without crashing.

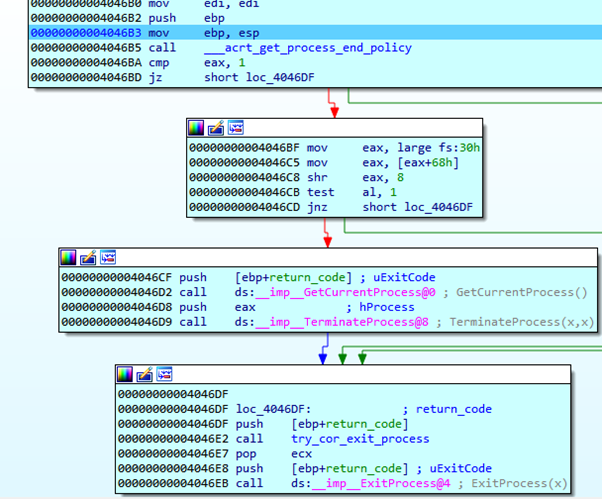




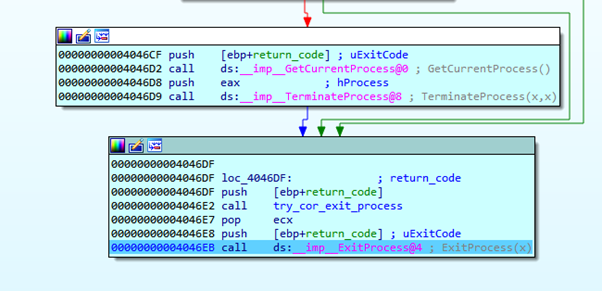
I see that when gets the final RETN I make the program jumps to exit, that part of the script is this one:



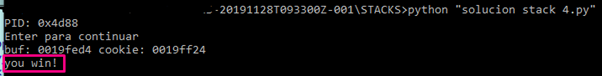
I see that EBP was set again to a bad value, but I decide not to jump to the first instruction of the function but here:



Where it modifies EBP with the correct value of ESP, and then continue to EXITPROCESS.



I get the ExitProcess where the program finishes without error.



In the next part we will start with ABO1 using RADARE, in that one we have to execute the calc or the notepad. I hope someone tries.

See you in part 6

Ricardo Narvaja

Translated by @Fare9

15/11/2019