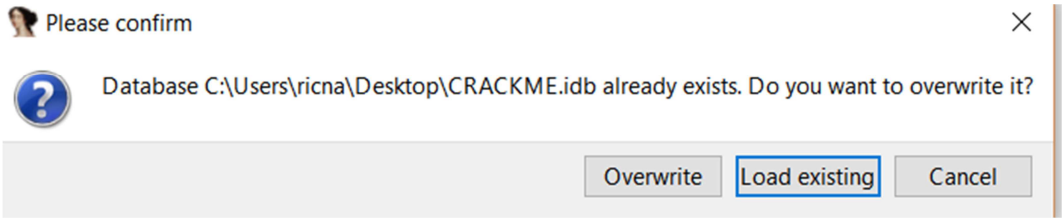
PART 10

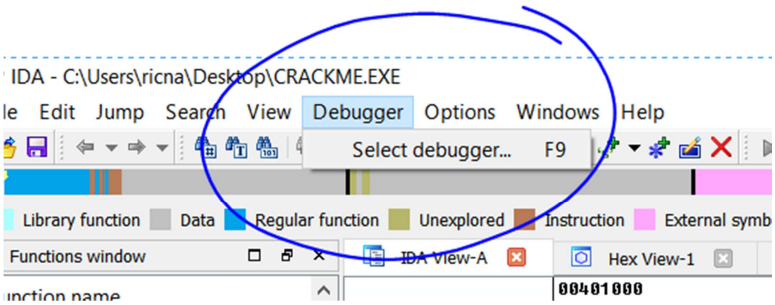
We have seen something about the LOADER and we will continue doing it, but we will see some DEBUGGER aspects to complement both.

IDA supports multiple DEBUGGERS to see that we load the original CRUEHEAD's crackme without any patch in IDA.

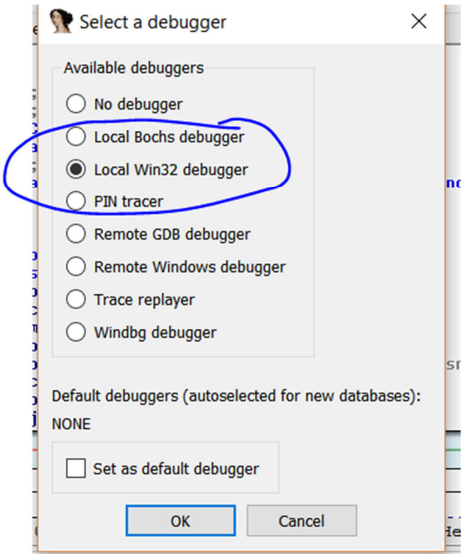
We select OVERWRITE if it asks to create a new database and overwrite the old one to do a new analysis if we already had a previous database or an idb file of the patched program.



Don't check MANUAL LOAD. So that, accept the windows that appear until it opens.



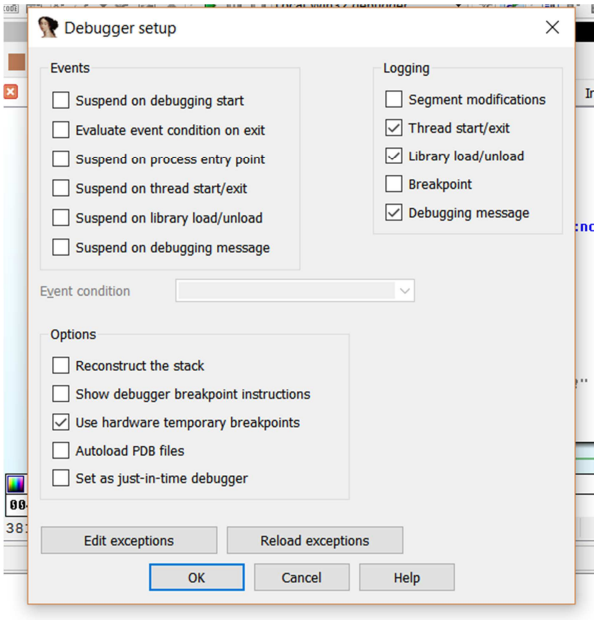
To start from the beginning among all the debugging possibilities, we select LOCAL WINDOWS DEBUGGER. We will see other ways later.



We'll see some DEBUGGER aspects in IDA that have their particularities and that we need to learn how to use it well to avoid complications.

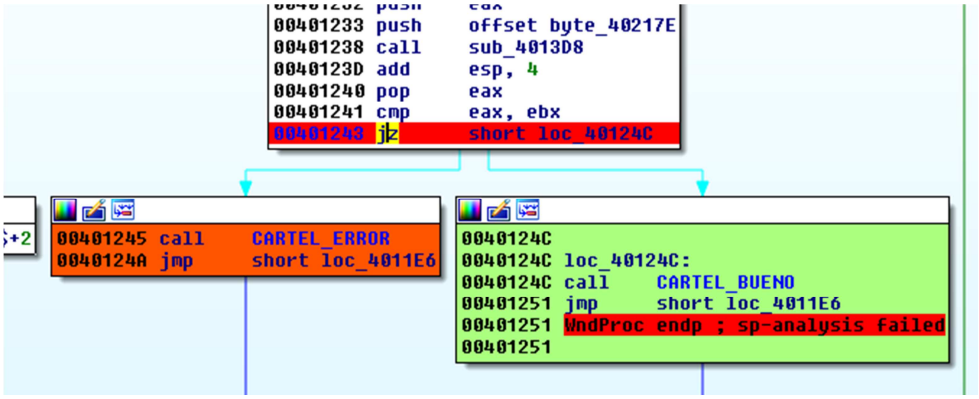
I think that the different arises because IDA started being just a LOADER. A very good static disassembler and with great interactive possibilities for reversing. Then, the DEBUGGER was added which brought some problems that were solved, but they sometimes make a difference with respect to the way of working with debuggers like Olly.

In DEBUGGERS – DEBUGGERS OPTIONS we have the options for the DEBUGGER.

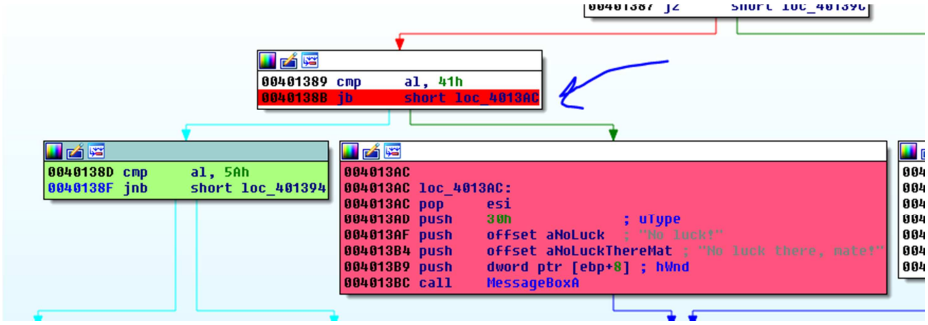


Let’s check SUSPEND ON PROCESS ENTRY POINT to stop at the entrypoint of it.

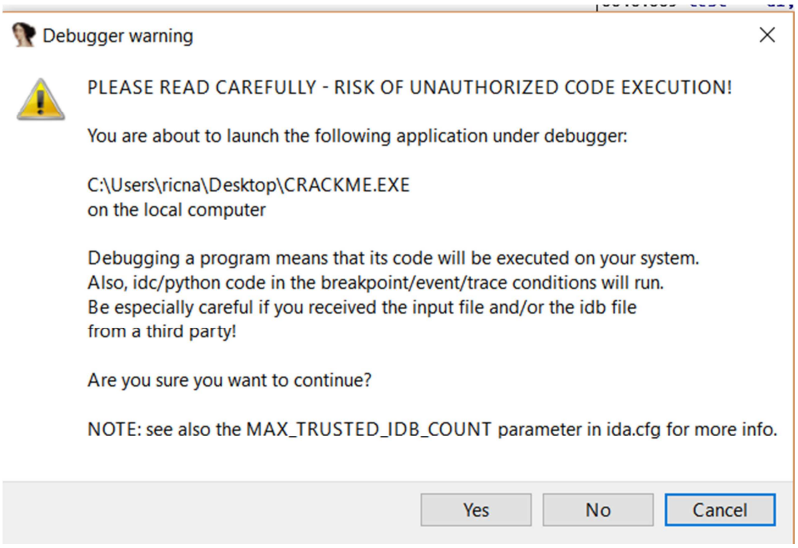
We will make the changes as before. We paint and rename the conditional jumps zone.



There, we set a BREAKPOINT on the jump that makes the decision in 0x401243 and we go to the other jump we had patched.



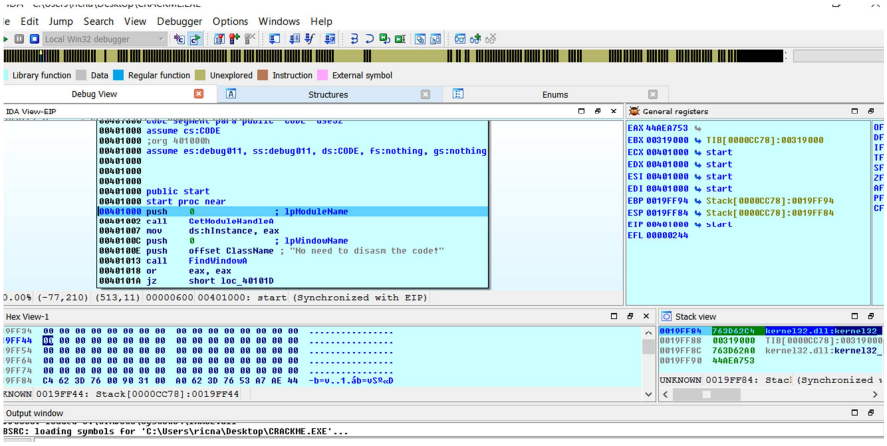
Now, we can run the debugger with DEBUGGER-START PROCESS.



This window will appear when we debug an executable in our local machine.

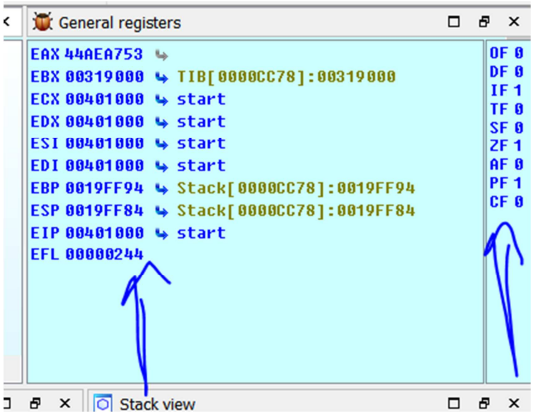
While we analyze it in the LOADER, it's never executed in our machine, but now it will and we have to be careful if it is a virus or something very dangerous, we need to use the REMOTE DEBUGGER and execute it in a virtual machine. We'll see that later.

As the CRUHEAD's crackme is better than Lassie medicated, we press YES.



As we set it to stop at the ENTRY POINT, it did it. It stopped at 0x401000. If I press the space bar, it will change into the graphical mode as in the LOADER.

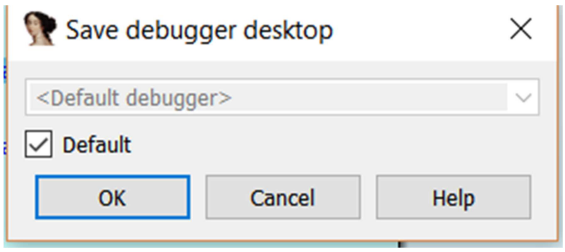
I rearrange the windows, dragging its margin down; I expand the stack a little to see the registers above and the flags on the right



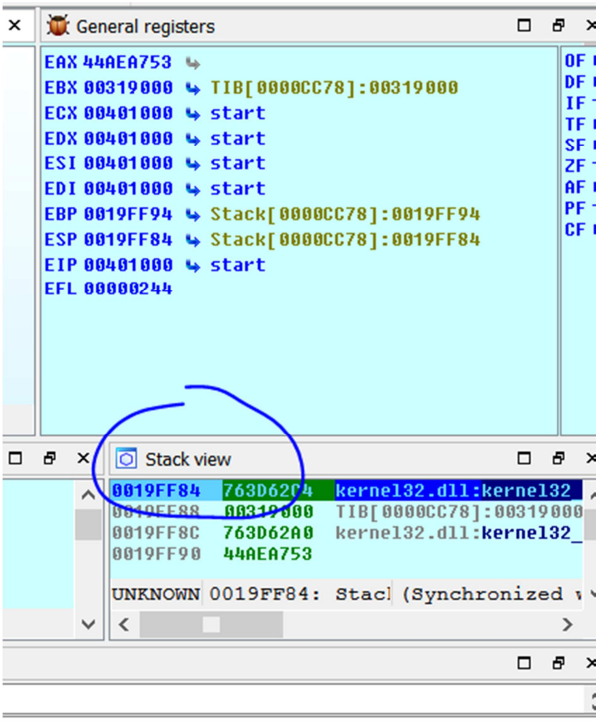
There, we see the registers and flags.

When I have a comfortable view, I will save it by default for the debugger in WINDOWS-SAVE DESKTOP checking the DEFAULT option.

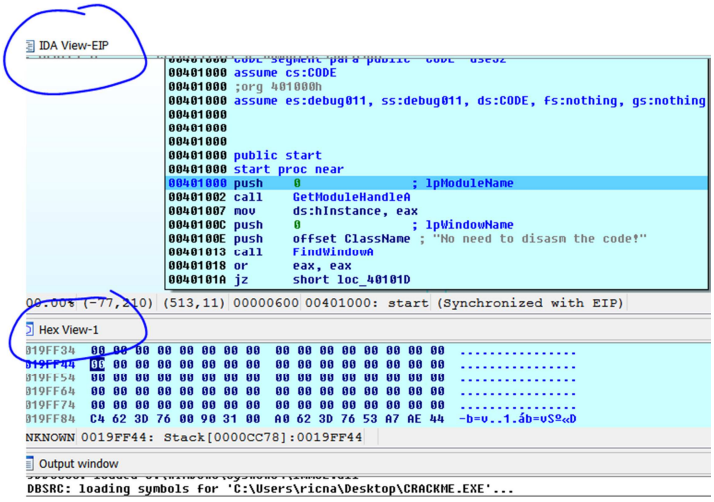
When we run the debugger, it will always run with our options. If we want to change it again, we can do it without problems.



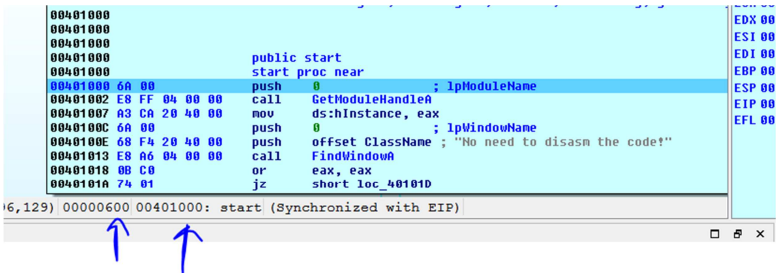
The stack view is under the registers.



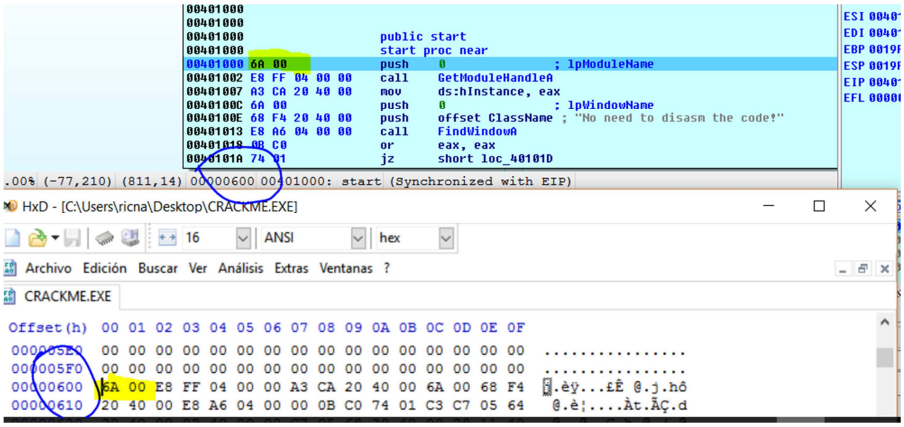
We also have the disassembly IDA-VIEW EIP and below is the HEW VIEW or HEX DUMP.



In the bottom part of the disassembly...

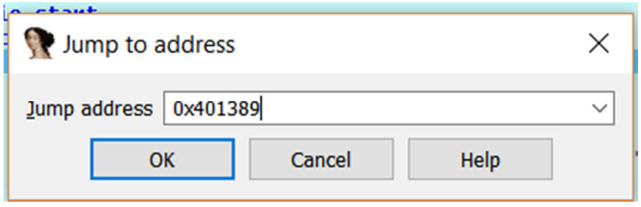


We'll always have the memory address and the FILE OFFSET that is the OFFSET in the executable file. If we open it in a hex editor, for example, HXD.

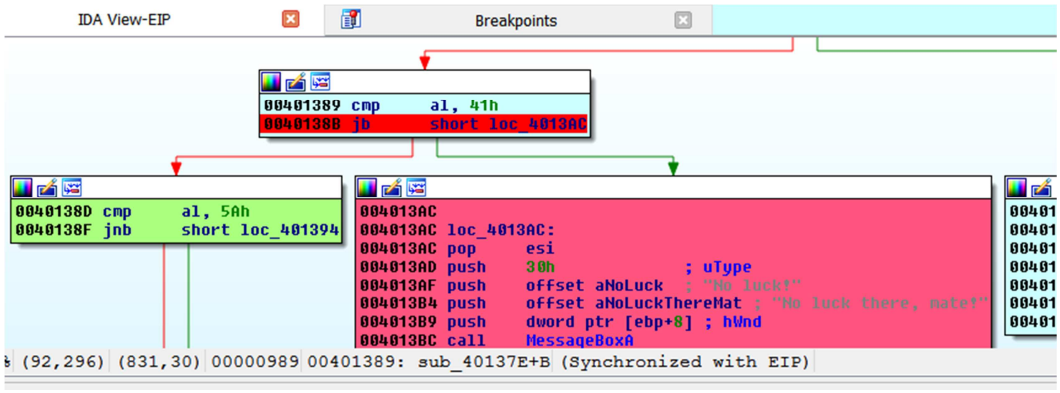


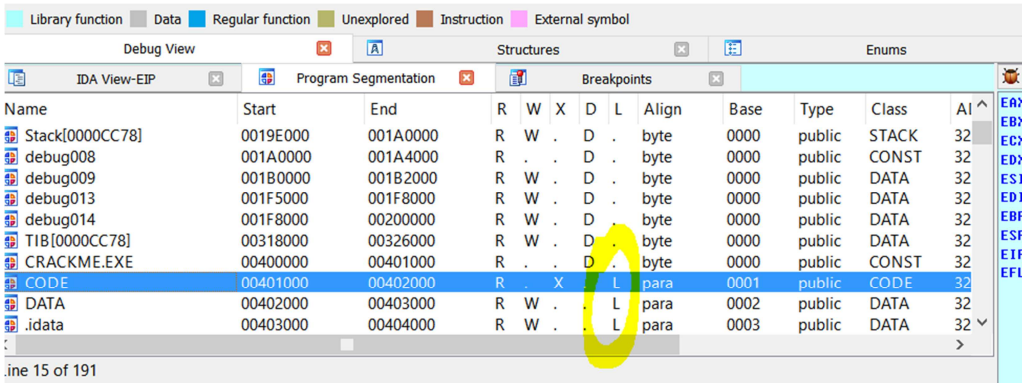
We see that in the 0x600 FILE OFFSET the bytes are there.

We know that, by default, the G shortcut is used to go to a memory address. If I press G and type 0x401389,



I'll go to the zone where a breakpoint was. We see the colors. In options, number of bytes, I enter 0 to have a better view and everything stays as in the LOADER. If we reverse, rename, etc, the changes are saved.

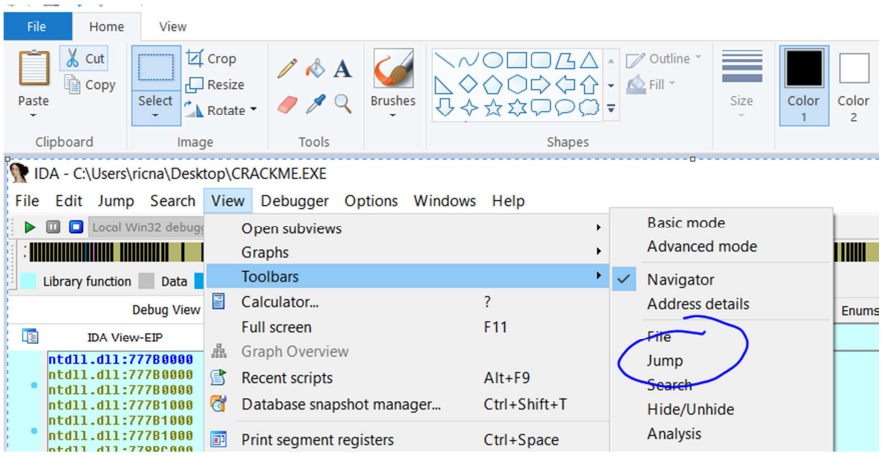




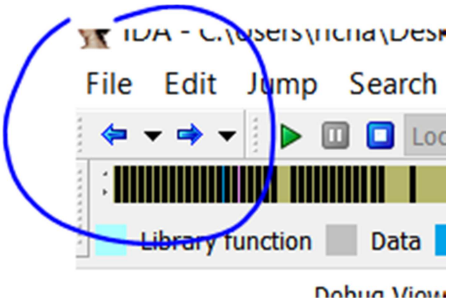
In VIEW-OPEN SUBVIEW-SEGMENTS, we see now the three segments that the LOADER loads, the CODE that loads 0x401000, the DATA and IDATA. Any modification done here will be saved because they are loaded in the LOADER. But the modifications outside them will be lost because they are modules loaded by the debugger and won't be saved in the database.

The modules we reverse and want to debug must be in the LOADER or L. When they have the L means that they are in both and I'll be able to reverse statically in the LOADER and debug in them without losing info and my static reversing work.

One of the little bars I consider useful is:

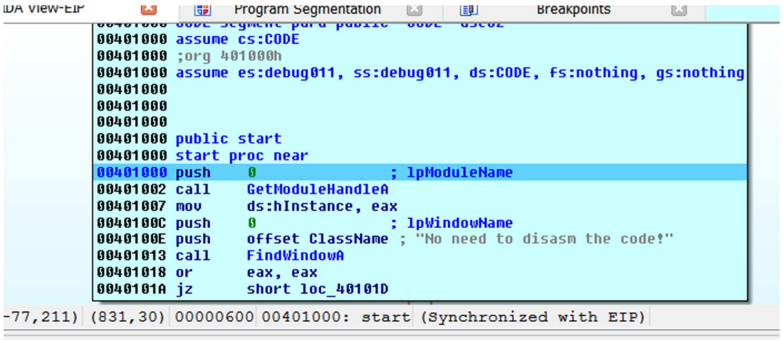


I add it and save it again with SAVE DESKTOP to have it as default.

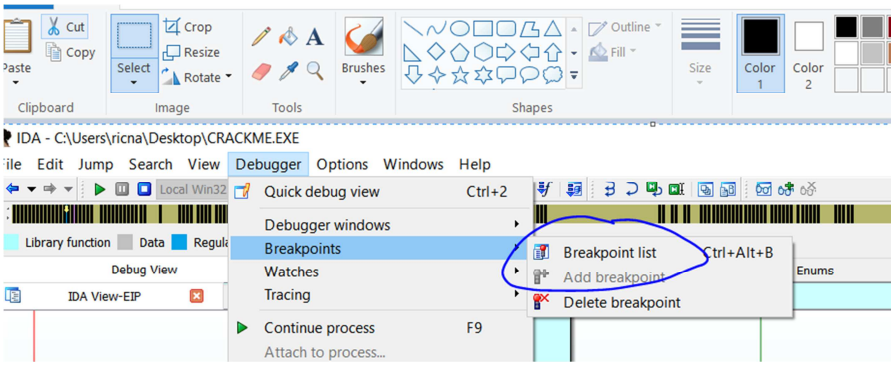


The possibility of going backwards and forward is very useful and comfortable.

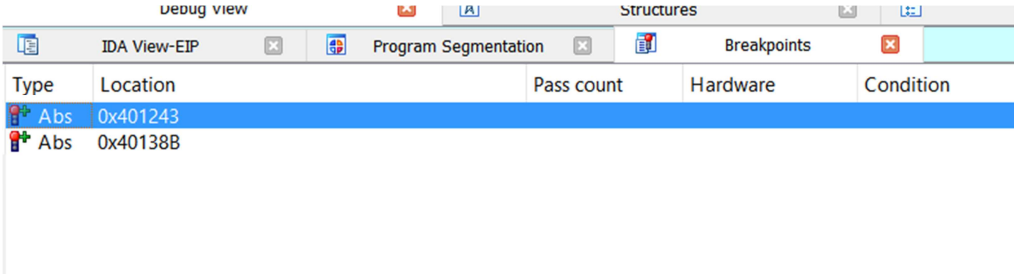
Pressing the arrow backwards I go back to the entry point I was before.

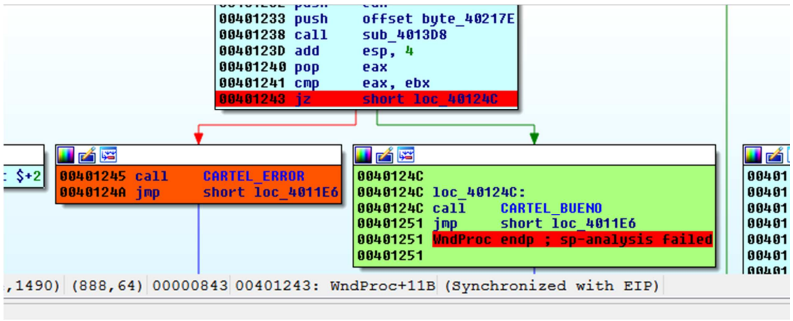


Also in the DEBUGGER menu.

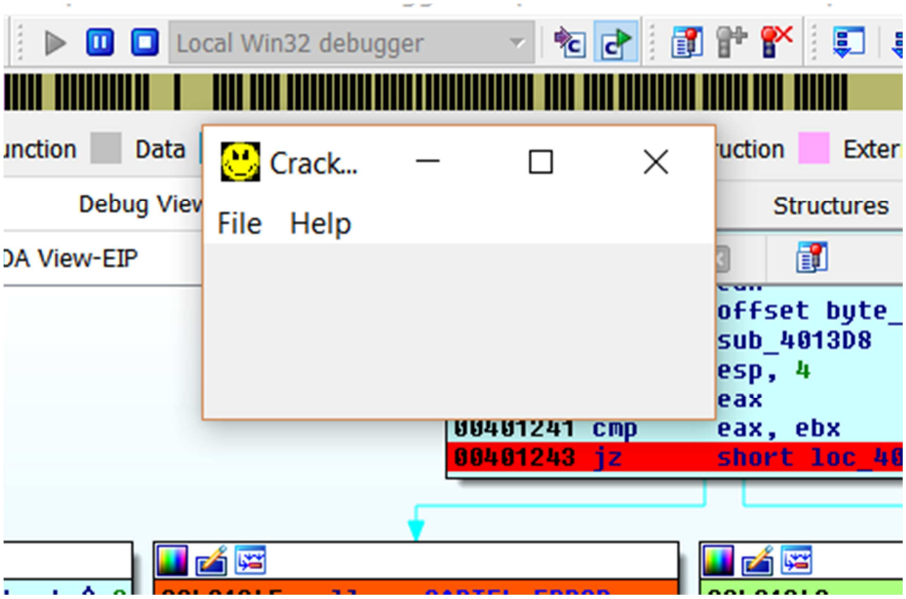


I can see the Breakpoint list and go wherever I want clicking on it.

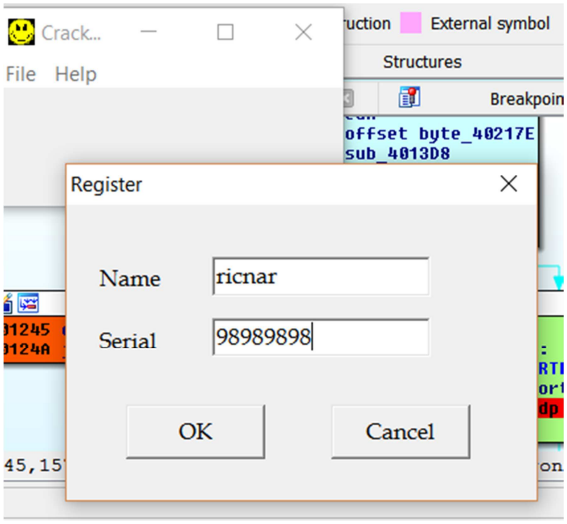




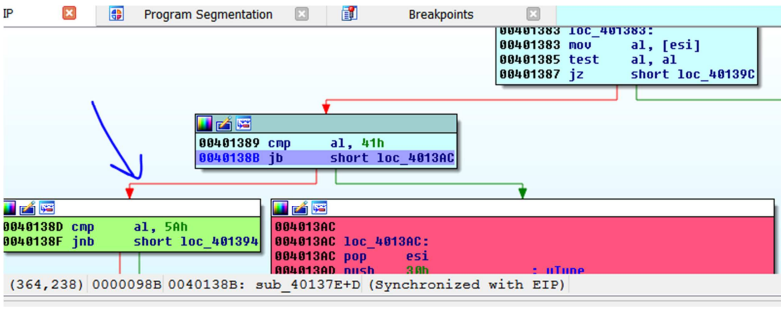
I’m at the Entry Point and I have two breakpoints set. I could press F9 to run it.



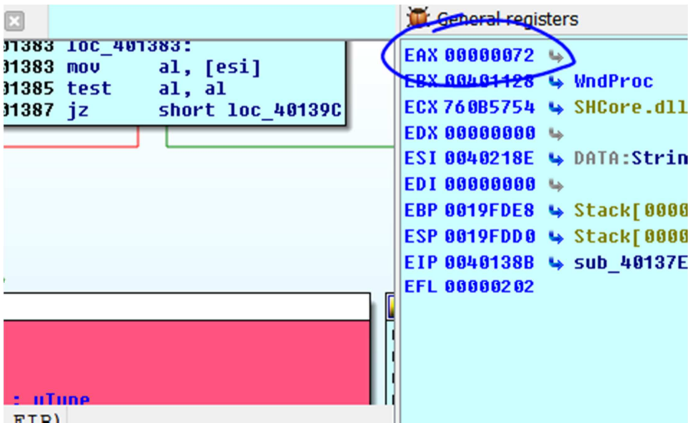
Go to HELP-REGISTER and enter a key.



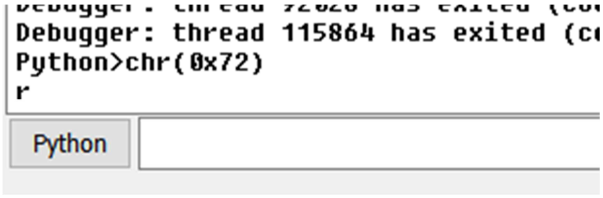
Click OK.



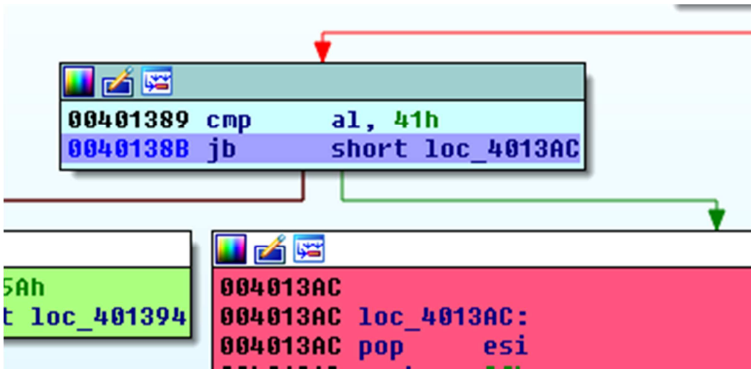
The left arrow is highlighted that is where it will continue. EAX = 0x72.



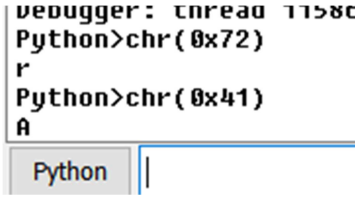
If I type **chr(0x72)** on the Python bar, I see that is the letter **r** of **r**icnar.



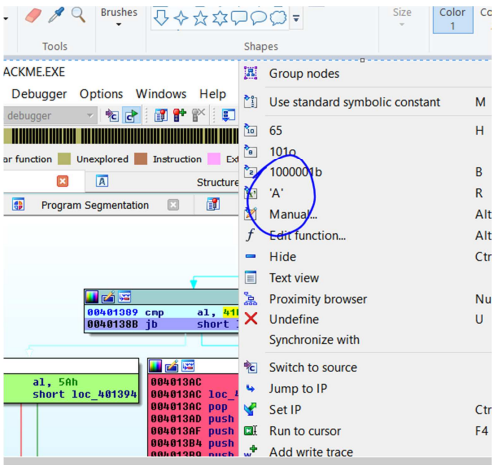
And it will compare to **0x41** to see if it is below.

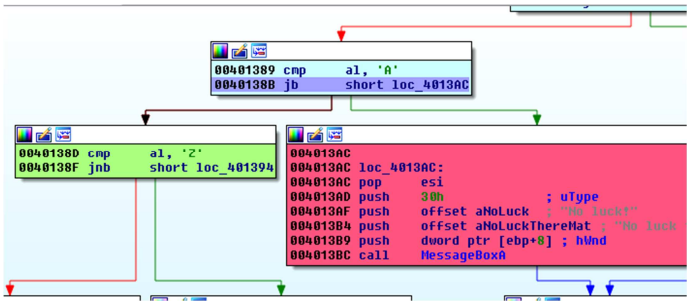


chr(0x41) in Python bar shows us the letter **A**.



We can also right click on the 41h of the disassembly and among the options; I see the same letter **A**.

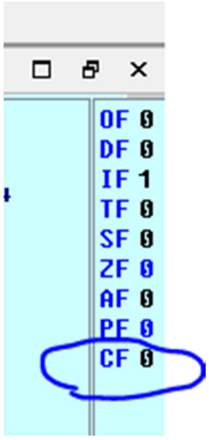




It compares 0x72 to A and Z. By now we won't solve this crackme completely, but we see that 0x72 is greater that 0x41 so, it won't go to the error msg red block. It would jump to it if it were below, but it can be evaluated seeing the flags.

ASM	CONDITION	OPERATION
JA	Z=0 and C=0	Jump if above
JAE	C=0	Jump if above or equal
JB	C=1	Jump if below
JBE	Z=1 or C=1	Jump if below or equal
JC	C=1	Jump if C flag is activated
JE or JZ	Z=1	Jump if equal or zero
JG	Z=0 and S=0	Jump if greater
JGE	S=0	Jump if greater or equal
JL	S<>0	Jump if less
JLE	Z=1 or S<>0	Jump if less or equal
JNC	C=0	Jump if C flag is not activated
JNE or JNZ	Z=0	Jump if not equal or zero
JNO	O=0	Jump if not O flag is activated (overflow)
JNS	S=0	Jump if not S flag is activated (sign)
JNP or JPO	P=0	Jump if not P flag is activated (parity)
JO	O=0	Jump if O flag is activated (overflow)
JP or JPE	P=1	Jump if P flag is activated (parity)
JS	S=1	Jump if S flag is activated (sign)
JCXZ	CX=0	Jump if CX = 0
JECXZ	ECX=0	Jump if ECX = 0

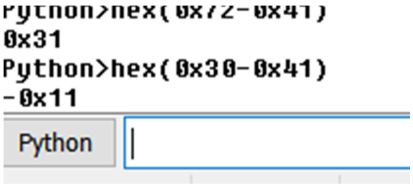
There, we see the JB jump which jumps or follow the green arrow in IDA if the first number is below, but the C flag (also called CF or C) is activated too when comparing that. It says that it jumps if C=1. If we see the IDA flags.



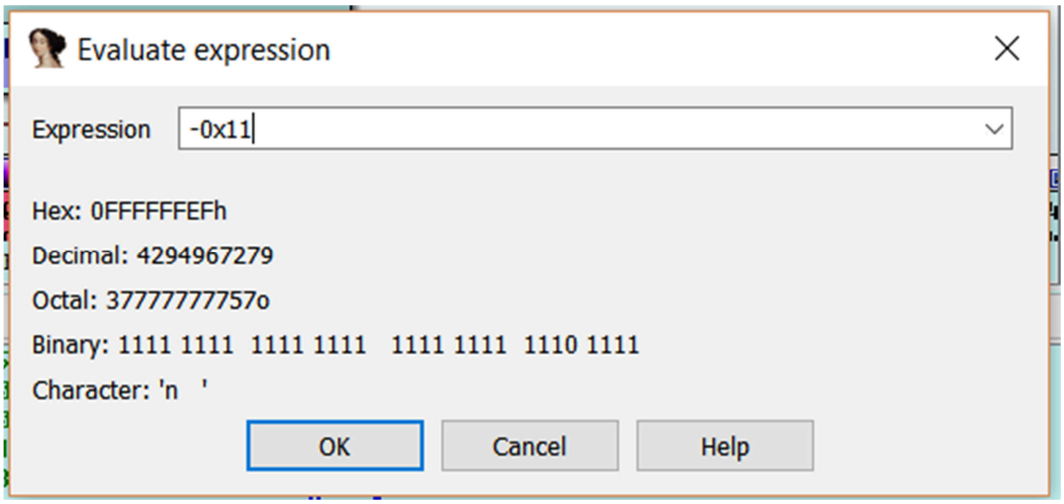
The C flag = 0, then the jump won't be taken and it will go to the red arrow.

What is the math condition in which the CARRY FLAG is activated?

The C flag gives us info about some wrong operation between unsigned integers. If I do the subtraction as **CMP** is a subtraction without saving the result: **0x72-0x41 = 0x31** this is positive and there is no problem, but if my value were **0x30** and I do **0x30 - 0x41 = -0x11**.



Which is a negative value and it is not accepted as a result of a positive number operation, then it continues working with it as hex.



It will be 0xFFFFFFFEF and that as a positive number 4294967279 is too big and in no way subtracting 0x30 – 0x41 = 0xFFFFFFEF positive.

How do we know if the sign is considered or not in an operation?

That depends on the jump, in this case, JB is a jump used after an unsigned comparison. For operations with signed numbers, we use JL.

If I compare 0xFFFFFFFF to 0x40 in an unsigned jump, it is obviously greater, but if it is a signed jump, it will be -1 and it is less than 0x40.

To evaluate if it is a signed comparison or not we must see the next conditional jump.

UNSIGNED JUMPS

SYMBOL	DESCRIPTION	FLAGS
JE/JZ	Jumps if equal or zero	ZF
JNE/JNZ	Jumps if not equal or zero	ZF
JA/JNBE	Jumps if above or not below or equal	CF, ZF
JAE/JNB	Jumps if above or not below	CF
JB/JNAE	Jumps if below or not above or equal	CF
JBE/JNA	Jumps if below or equal or not above	CF, AF

If the jump is any of these the sign is not evaluated, while each one has its counterpart like JB and JL in the signed jump table.

SIGNED JUMPS

SYMBOL	DESCRIPTION	FLAGS
JE/JZ	Jumps if equal or zero	ZF
JNE/JNZ	Jumps if not equal or zero	ZF
JG/JNLE	Jumps if greater or not less or equal	ZF, SF,OF
JGE/JNL	Jumps if greater or equal or not less	SF,OF
JL/JNGE	Jumps if less or not greater or equal	SF,OF
JLE/JNG	Jumps if less or equal or not greater	ZF,SF,OF

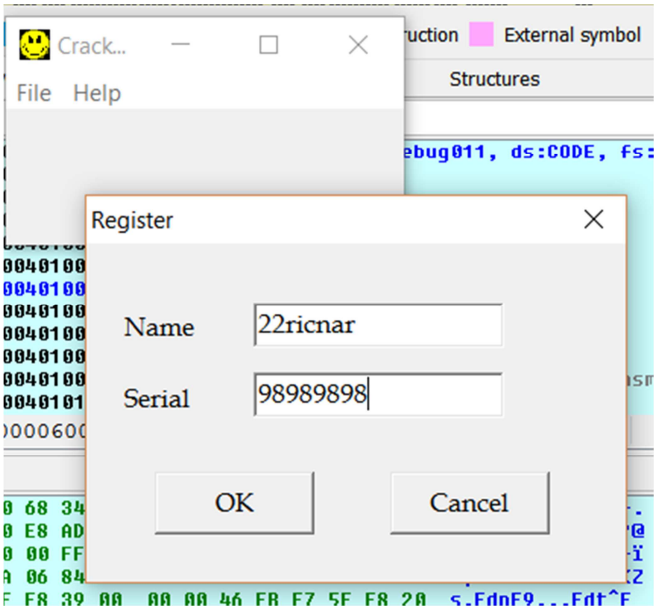
We see that JE evaluates if they are equal in both tables because, in that case, the sign is not important. If they are equal, the ZF = 1. It means that flag is activated.

JG = jumps if greater. In the signed table, it has its counterpart with the **JA** that jumps if above in the unsigned table.

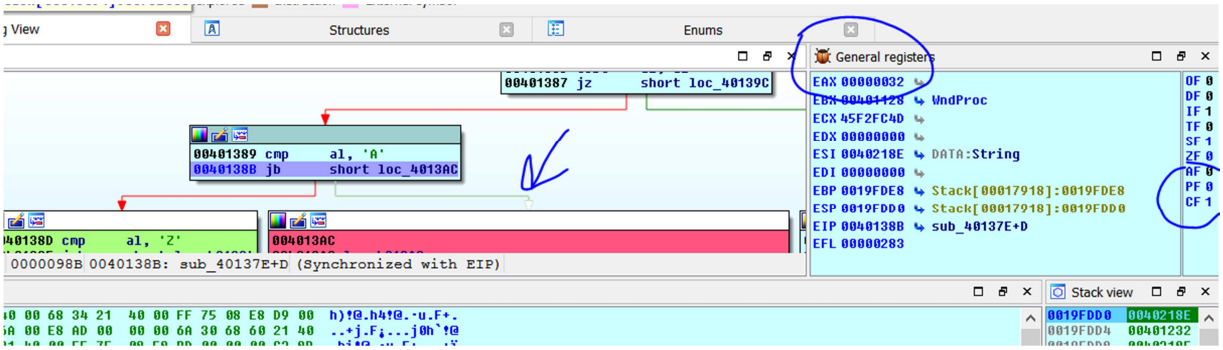
In the daily analysis we see the flags too much and see a JB jump and we know that it is a comparison between positive or unsigned numbers and that if the first one is below, it will jump, but it’s good to see what’s below.

If I continue stopping at all breakpoints, I we’ll see that I’m in a loop that reads each character of my name at a time and compare them to 0x41. If there is any below, it will show the bad boy. As I enter only letters (ricnar) it won’t be the case.

But let's restart the process with TERMINATE PROCESS and START PROCESS again and let's enter as name: **22ricnar** and the key: **98989898**.



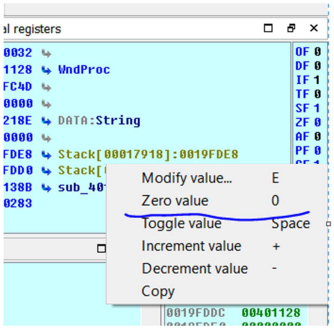
Click OK and it will stop at the breakpoint.



Now, my first character is 0x32 that belongs to the first **2** of **2**2ricnar.

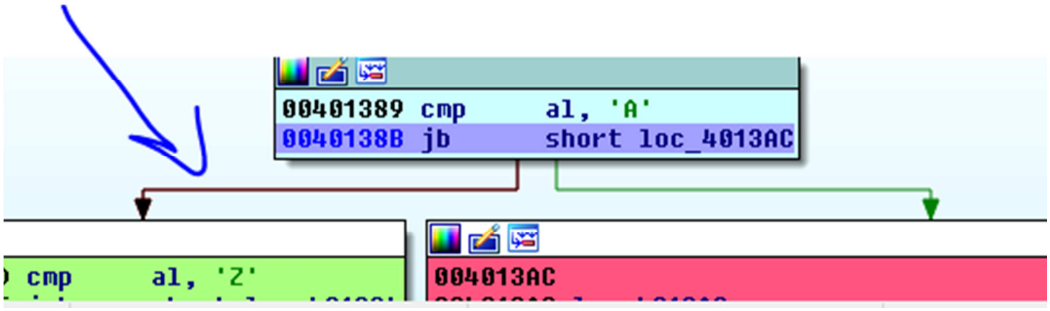
As 0x22 is less than 0x41, the green arrow activates meaning that the jump will be taken and we note that the C flag is activated because **0x32 - 0x41** is an unsigned subtraction it results negative and that is an error that activates the C flag.

If I right click on the C flag…



We can set it to zero.

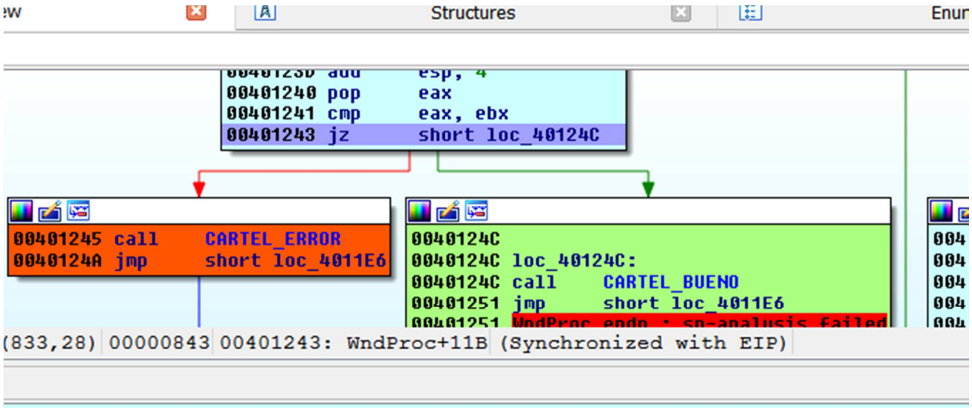
At the moment we change it, the red arrow starts blinking because we inverted it.



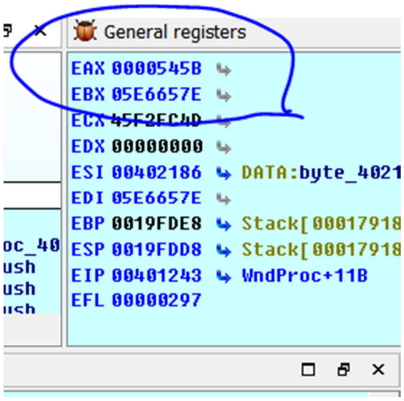
If we press RUN, it will stop again when it evaluates the next **2** of 2**2**ricnar and the green arrow will blink again. We invert CF again setting it in 0.

The next times it stops at this jump will correspond to **ricnar** which are greater than **0x41** and they don't activate the **CF** and they continue for the red arrow.

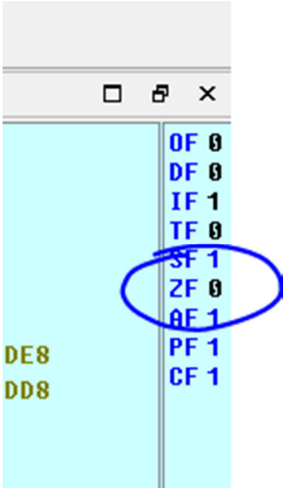
After fooling the check of each character of my name, we go to the final jump.



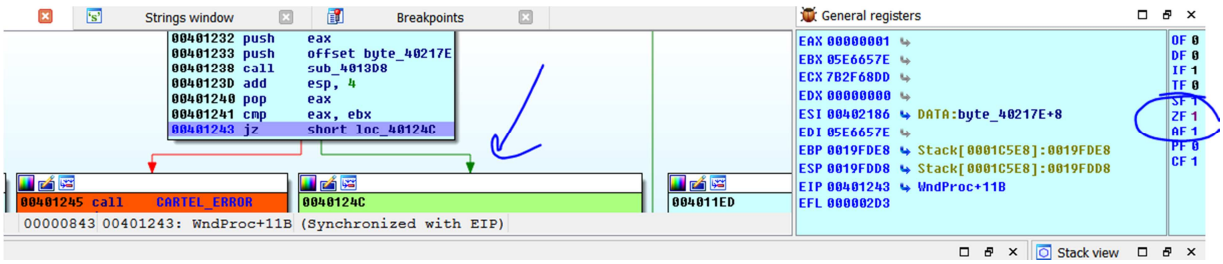
There, it compares EAX to EBX to see if they are equal. It doesn't matter the sign. It turned on the red arrow because they are different and it will kick me out.

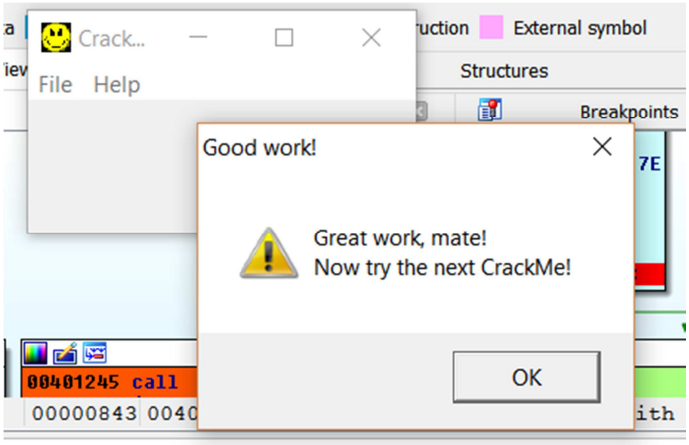


There, I see they are different and the Z flag is not activated.



If we activate it, it will change the jump and it will follow the green arrow to the Good Boy. Right click on the ZF and select INCREMENT VALUE.





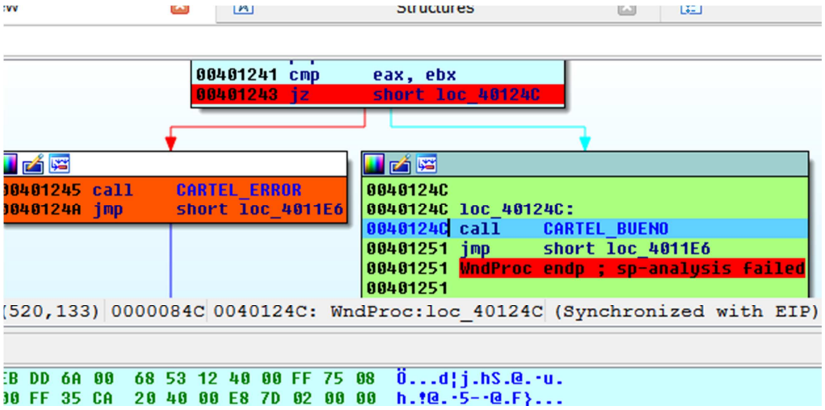
We did the same as when we patched it, but without any modification, just changing the flags in the DEBUGGER.

Many times when we don't want to invert jumps directly we go to the good block where we want it to continue and place the cursor there and right click on it, although IDA 6.8 has a bug that was solved in the 6.9 version that sometimes when right clicking it crashed, if that happens to you look for the shortcut you need in the following link.

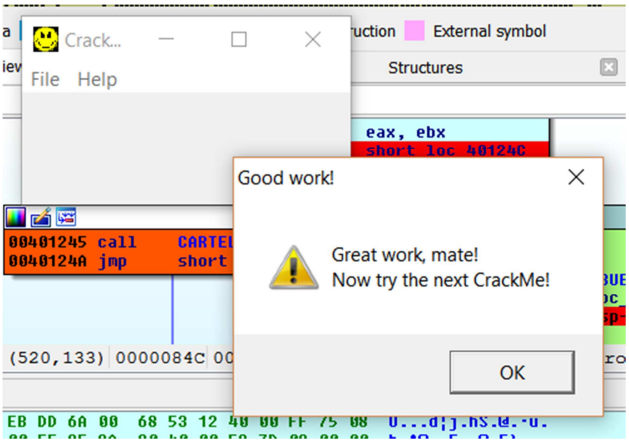
https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf

Also, in IDA going to OPTIONS-SHORTCUTS.

If we have problems when we right click to set EIP, place the cursor where you want to go, for example, 0x40124C and press CTRL + N that is SET EIP.



And the program will continue from 0x40124C that is the same that inverting the flag.



Ricardo Narvaja

Translated by: @IvinsonCLS