PART 7

FLOW CONTROL INSTRUCTIONS

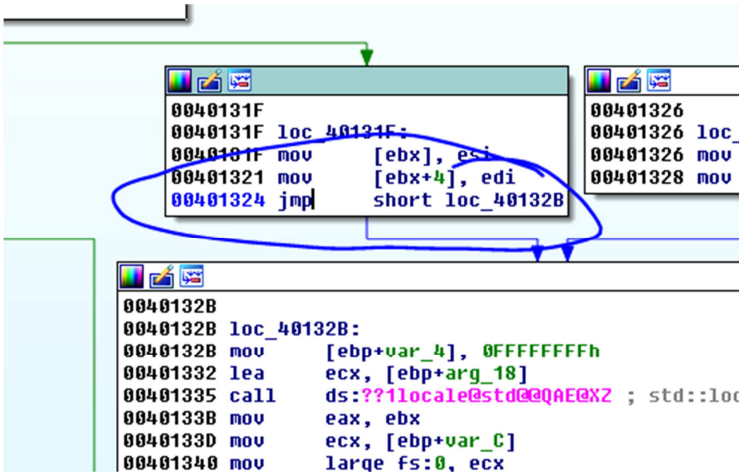
We are finishing with the instructions that are always the hardest part. Swallow that bitter pill because the best is coming soon.

The next instructions control the program flow. We know that EIP points to the next instruction that will be executed and when that happens, EIP will point to the next one.

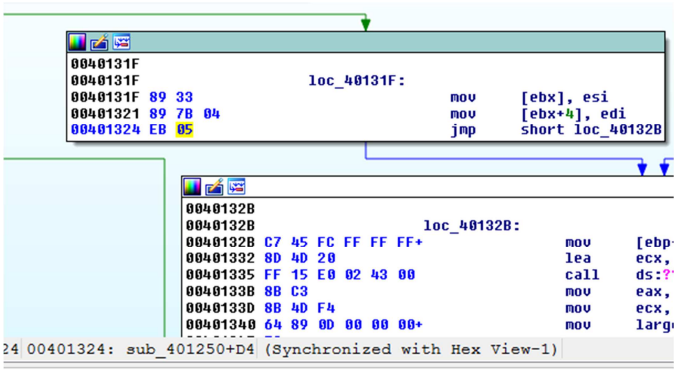
But the program itself has instructions to control its flow being able to detour the execution to a desirable instruction. We will see these cases.

JMP A

A will be a memory address where we want the program to jump unconditionally.



JMP SHORT is a short jump composed of two bytes and it has the possibility of jumping forward and backwards only. The direction is indicated by the second value because the first one is the OPCODE of the jump. We cannot jump so far.



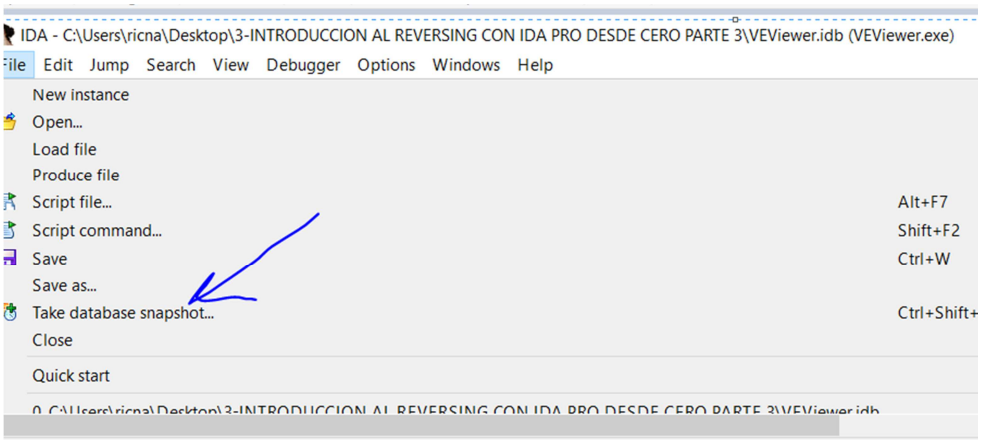
If we set IDA to show the instruction bytes, we can see the opcode EB that belongs to JMP and that it will jump 5 places forward from the end of the instruction. The destination address can be calculated like this:

```
Python>hex(0x401324 + 2 +5)
0x40132b
```

The instruction start address + 2 that is the number of bytes that compound the instruction and then I add it the 5 that the second byte shows.

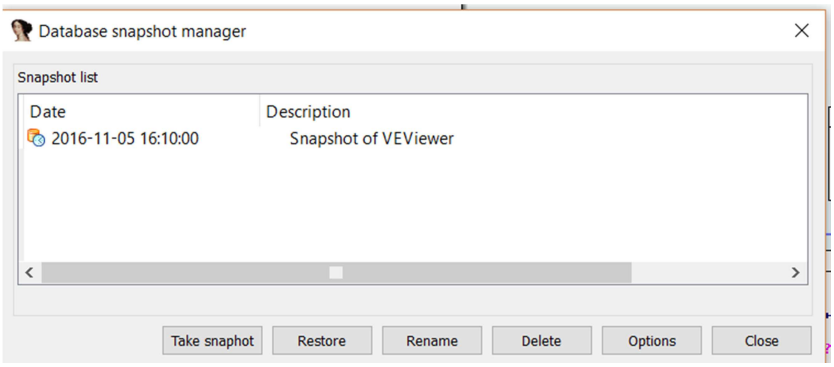
Obviously, jumping forward and backwards with just a byte is not a big range. The biggest positive jump forward will be 0x7F. We will see some example.

As we will do some modifications that will break the function down, it is recommended to do a database snapshot which lets us go back to the previous state. We do this when we think something can be broke down and we don't know how to revert it.



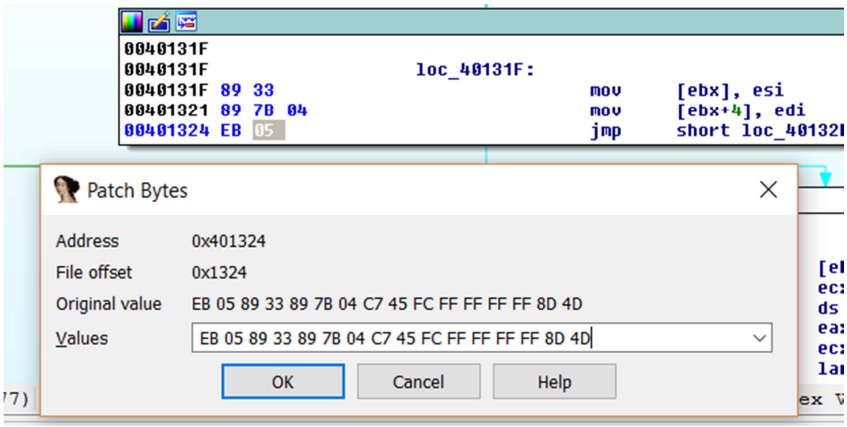
It will ask for a name and that's all.

IN VIEW-DATABASE SNAPSHOT MANAGER



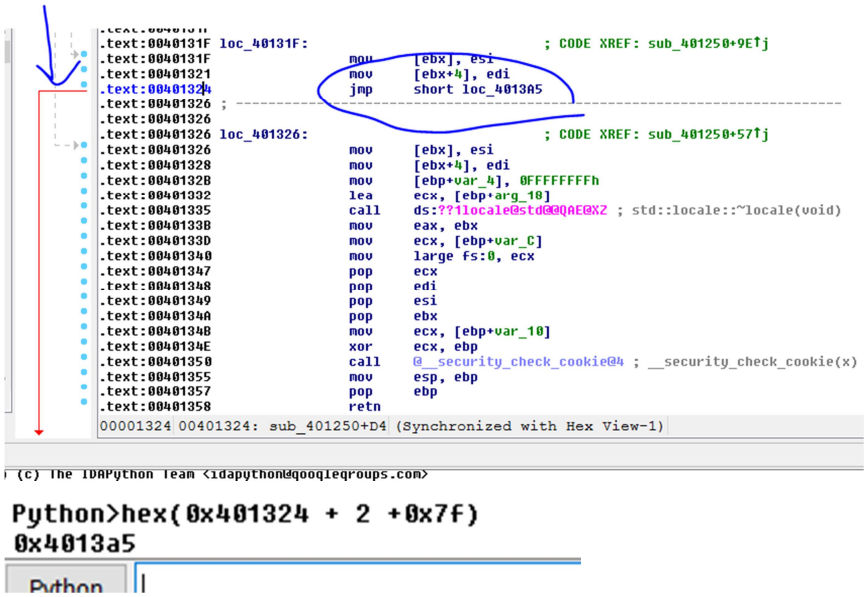
We can see a list of all snapshots and the date when they were done and with the RESTORE button we can go back to the saved states we want.

Let's see what happens if I change the 05 by 7F.



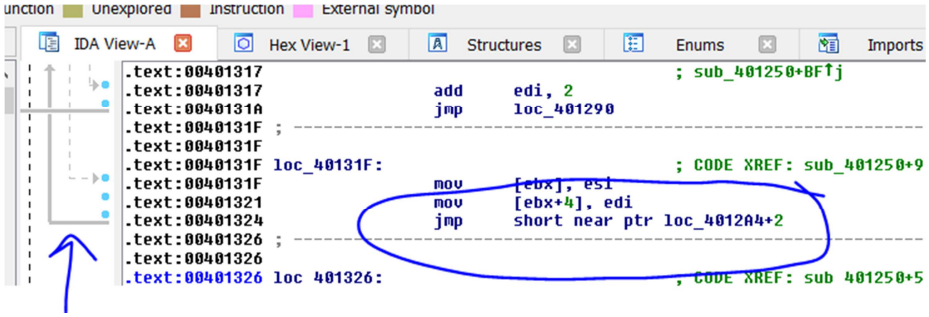
Using IDA PATCH BYTE, I change 05 by 7F.

The jump is a little longer and it looks out of the function. If I press the space bar to quit the graphic mode.

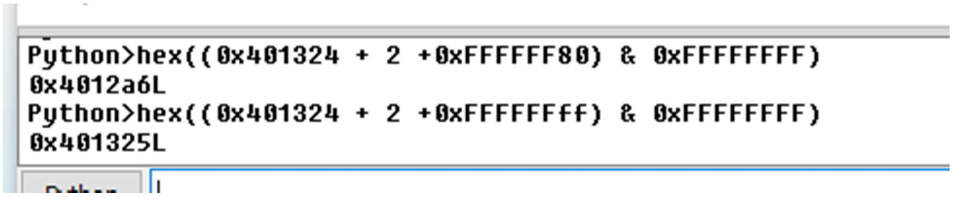


We note that it is OK and it jumps forward to 0x4013A5. Let's see what happens if we change the 0x7F by 0x80.

We go back to the graphic mode with the space bar and change it by 0x80.



Now, we do the biggest jump upward. Here, exceeding the 0x7F that is the biggest jump forward, we changed to 0x80 that is the biggest jump backwards.



In this case, as we go backwards, to fit everything with the formula and just for math purposes because Python doesn't know that jumps can go forward or backwards from a value, we must write -0x80 to its hexa value in dword that is 0xFFFFFF80 and then, as we saw when doing AND 0xFFFFFFFF of the result we clean all bits bigger than the necessary for a 32-bit number and we have the same address 0x4012A6.

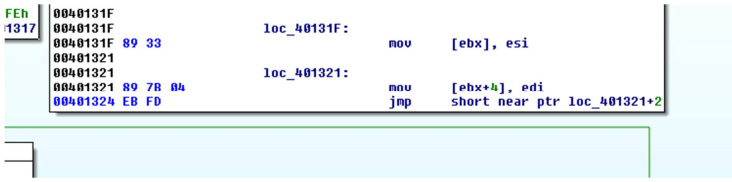
If I use 0xFF as a second byte, it would be a minimum jump that it is -1 in hex. I add it 0xFFFFFFFF to fit the count. Remember that we always add the size of the instruction, in this case, two bytes. This will jump backwards because the 2 we added makes it take it as the calculation start. It will go to 0x401325.

If we continue backwards one more position, the second byte would be FE that means jumping -2 backwards from the instruction end. In the formula, it would be adding 0xFFFFFFFE.

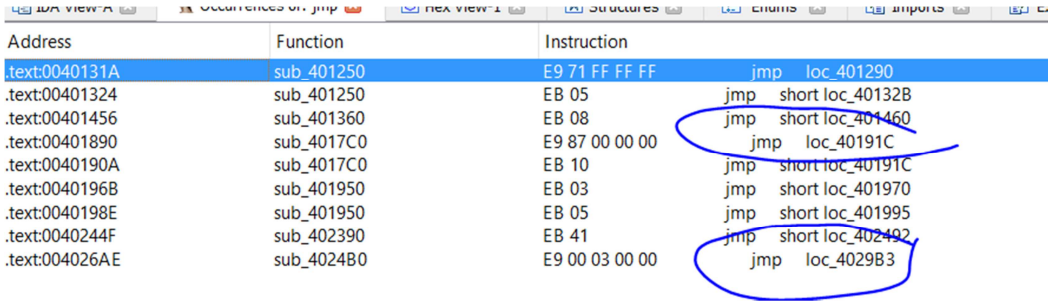
```
Python>hex((0x401324 + 2 +0xFFFFFFFE) & 0xFFFFFFFF)
0x401324L
```

This jumps to the same instruction start that is known as INFINITE LOOP. It always repeats without going out.

And so on, jumping -3 from the instruction end backwards will be FD. It will jump to 0x401323.

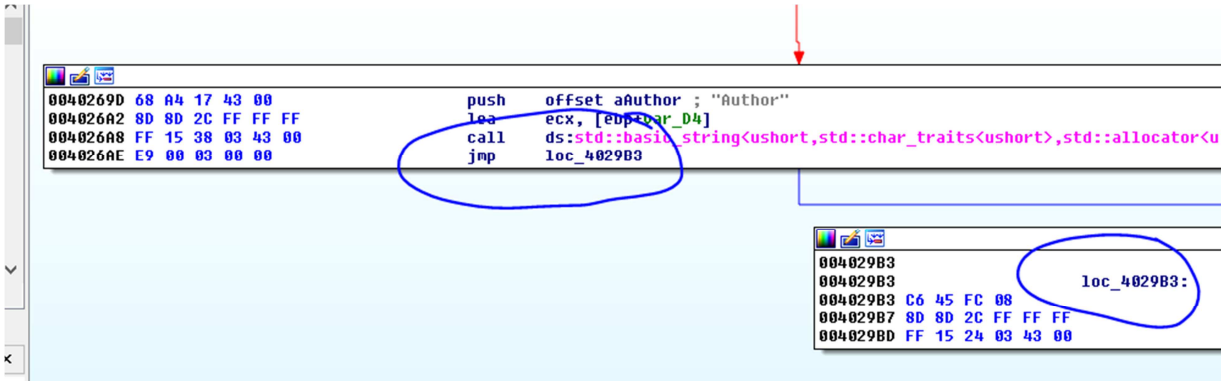


We can’t jump to all addresses with short jumps because we are limited to a few bytes around we are, so we use long jumps.

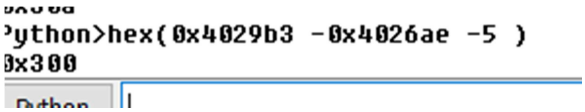


Address	Function	Instruction	
.text:0040131A	sub_401250	E9 71 FF FF FF	jmp loc_401290
.text:00401324	sub_401250	EB 05	jmp short loc_40132B
.text:00401456	sub_401360	EB 08	jmp short loc_401460
.text:00401890	sub_4017C0	E9 87 00 00 00	jmp loc_40191C
.text:0040190A	sub_4017C0	EB 10	jmp short loc_40191C
.text:0040196B	sub_401950	EB 03	jmp short loc_401970
.text:0040198E	sub_401950	EB 05	jmp short loc_401995
.text:0040244F	sub_402390	EB 41	jmp short loc_402492
.text:004026AE	sub_4024B0	E9 00 03 00 00	jmp loc_4029B3

There, we see some long jumps. Remember that loc_ means that instruction is common.



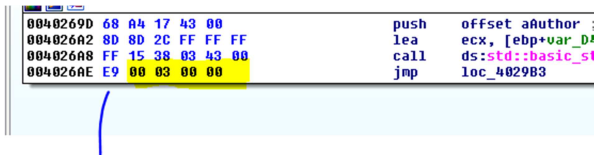
There, we see the long jump. The distance between 0x4026AE and 0x4029B3 is bigger than we can reach with a short jump.

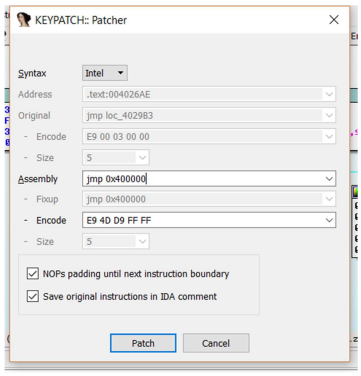


There, we see that the distance is calculated with the formula:

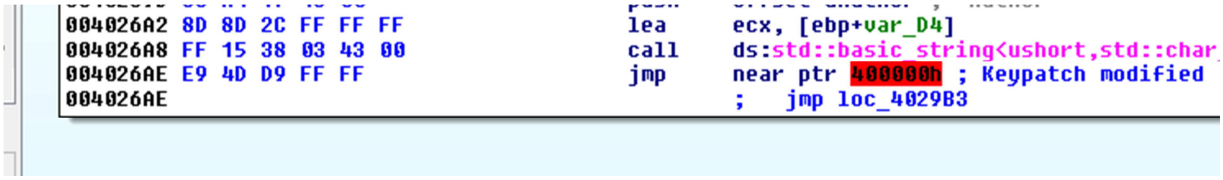
Final address - start address - 5

5 is the instruction size. The result is **0x300** that is the dword next to the long jump opcode 0xE9.

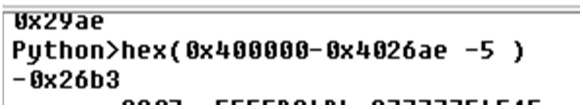




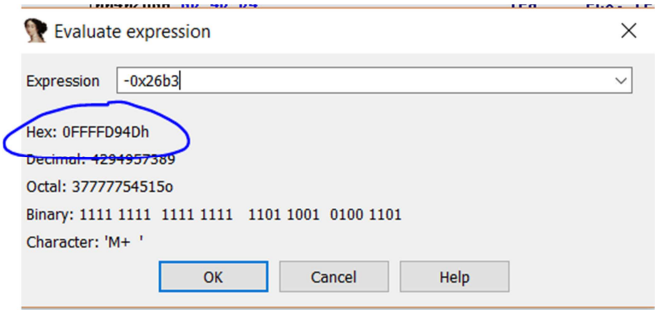
If I change the destination address of that jump to an address backwards with KEYPATCH, for example, 0x400000.



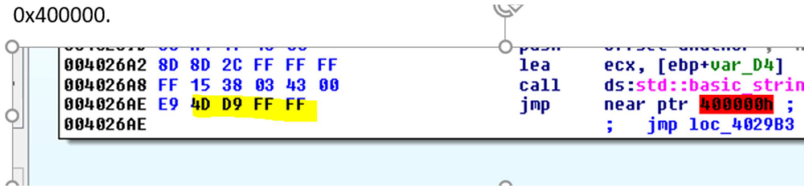
Although, it is marked in red because it is not a valid address at this moment. I will see if I can make a Python formula to jump backwards.



The result is -0x26b3 of distance using the same previous formula.



These bytes in hexa are FFFFD94D which are the ones next to the opcode 0xE9 but inverted.



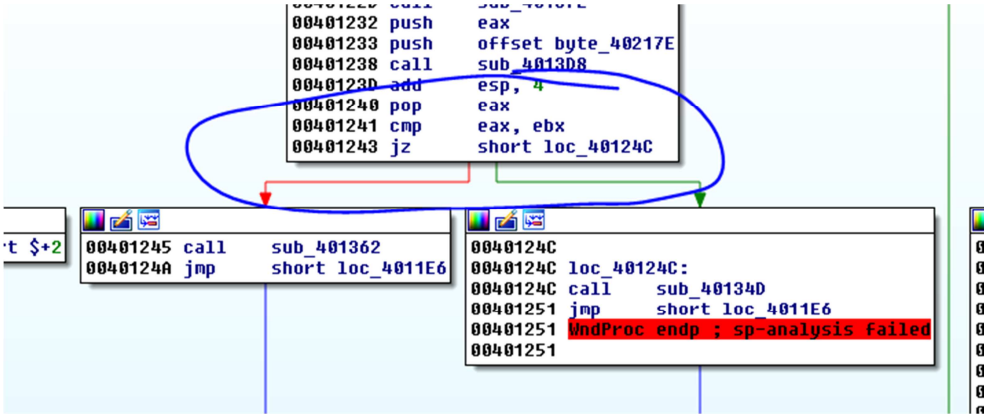
CONDITIONAL JUMPS

Normally, programs have to make decisions and according to the comparison of some values it can detour the program execution to a point or another one.

For example: **CMP A,B**

I can compare **A** and **B** and according to the relation among them, the program does something if not then does something different.

Normally, after the comparison that changes the famous flags, according to their state the conditional jump will decide what to do.



There, we see some example of a JZ conditional jump. It jumps if the Z or zero flag is activated. This happens when in the previous CMP EAX and EBX are equal because CMP is similar to SUB internally but without saving the result.

CMP subtracts both registers and if they are equal, the result will be 0 and that activate the Z flag that is what the JZ jump checks to jump. If the Z flag is activated it goes to the green arrow way and if not it goes to the red arrow way if they are different.

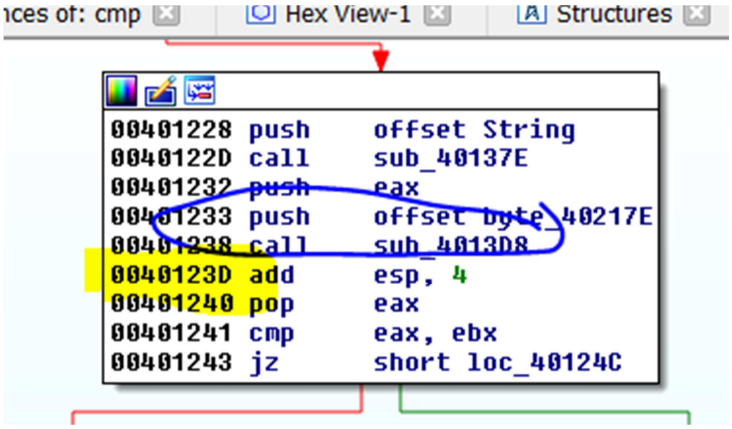
When using the debugger, we will see some examples triggering flags. By now, the important thing is to know that if there is a comparison after it you could see these different jumps.

Hexadecimal	ASM	Meaning
75 or 0F85	JNE	Jumps if not equal
74 or 0F84	JE	Jumps if equal
EB	JMP	Jumps always
90	NOP	No Operation (It does nothing just filling)
77 or 0F87	JA	Jumps if above
0F86	JNA	Jumps if not above
0F83	JAE	Jumps if above or equal
0F82	JNAE	Jumps if not above or equal
0F82	JB	Jumps if below
0F83	JNB	Jumps if not below
0F86	JBE	Jumps if below or equal
0F87	JNBE	Jumps if not below or equal
0F8F	JG	Jumps if greater
0F8E	JNB	Jumps if not greater
0F8D	JGE	Jumps if greater or equal
0F8C	JNGE	Jumps if not greater or equal
0F8C	JL	Jumps if less
0F8D	JNL	Jumps if not less
0F8E	JLE	Jumps if less or equal
0F8F	JNLE	Jumps if not less or equal

The JMP and NOP don't belong to that table. The rest are conditional jumps that evaluate different comparison states. If the first one is greater, greater or equal, less, etc, there are many possibilities which we will see later in deep when learning the use of the debugger.

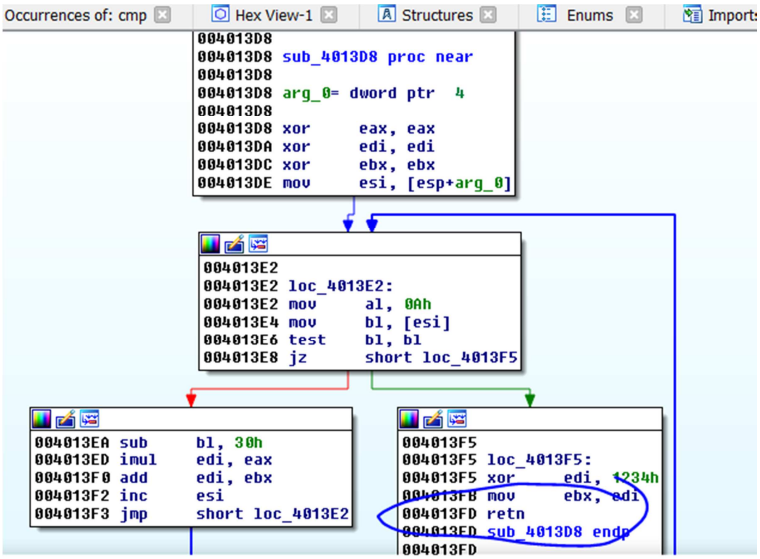
CALL and RET

CALL is used to call a function and **RET** to go back or return to the next instruction where it was called.

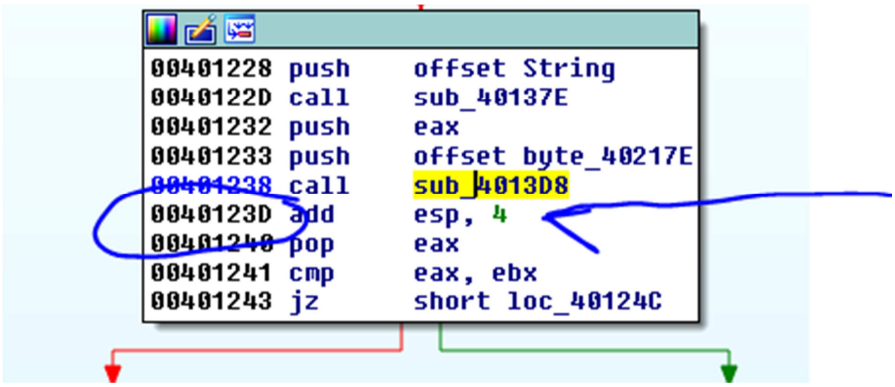


There, we see a CALL that will jump to 0x4013D8 to execute that function (we see the **sub**_ before the address 0x4013D8 which tells us that.)

The CALL saves, on the stack, the value where it will return, in this case, 0x40123D. I can enter that CALL by pressing ENTER.



After that function finishes, it will go to a RET that takes the return address 0x40123D saved on the stack and jumps there continuing the execution after the CALL.



We finished the main instructions review. If we need to talk about some instruction more ahead, we will do it with more details.

Ricardo Narvaja

Translated by: @IvinsonCLS