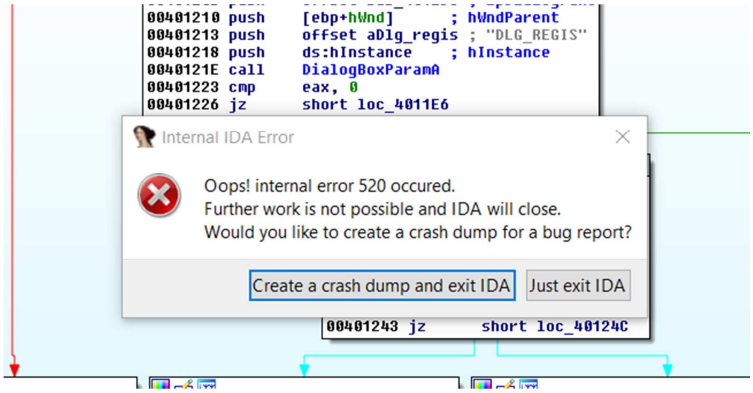
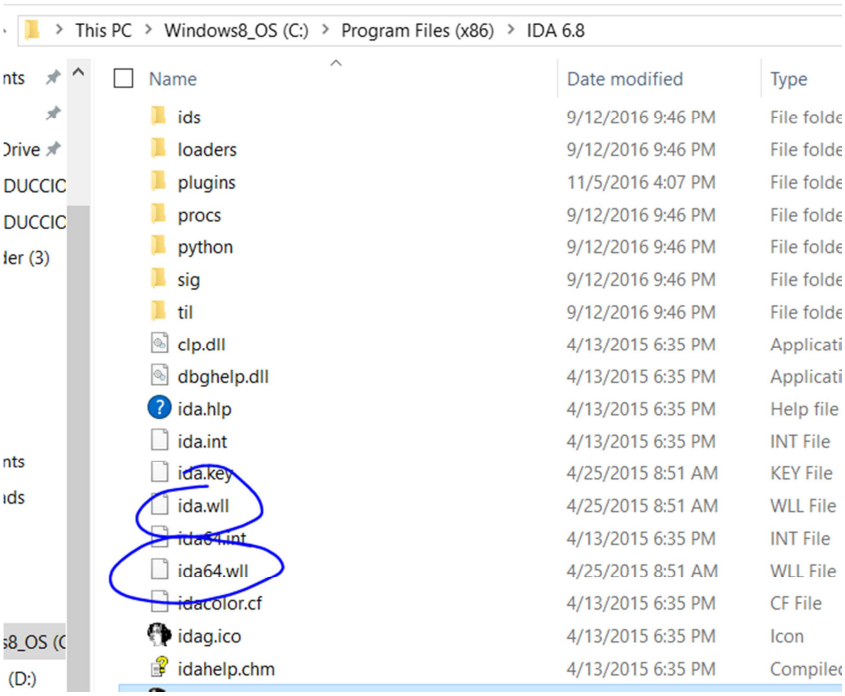
PART 11

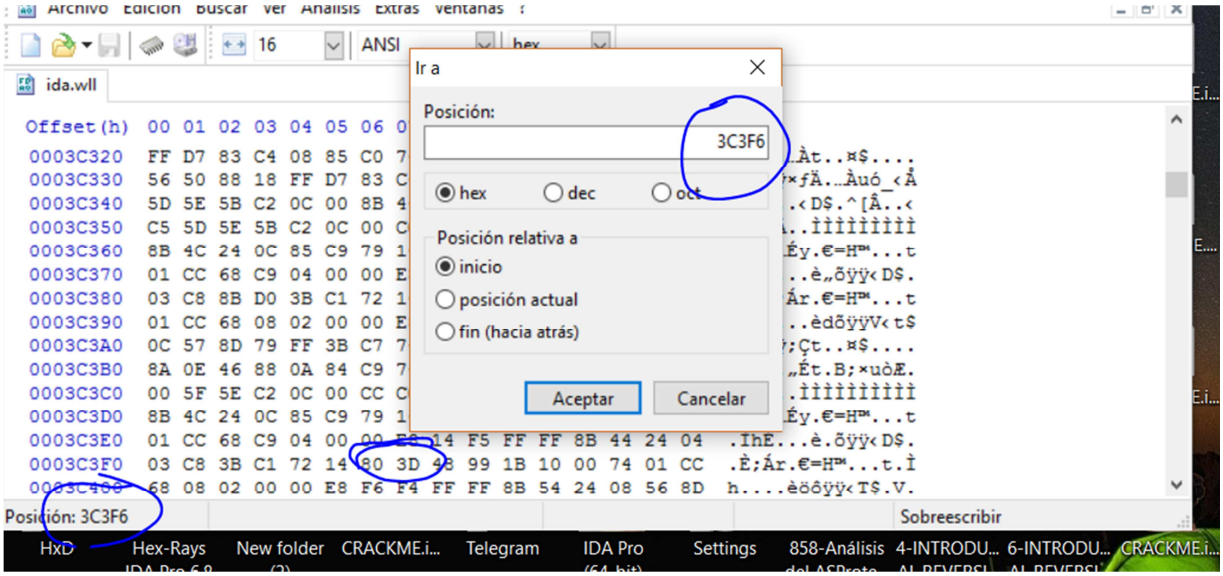
Before continuing, let's see if we can fix the bug in IDA v6.8 which was fixed in v6.9, but we don't have that new version. To see if yours has that bug, open IDA and in an instruction, press ALT+M that is similar as setting a mark (JUMP-MARK POSITION) and then, right click there.



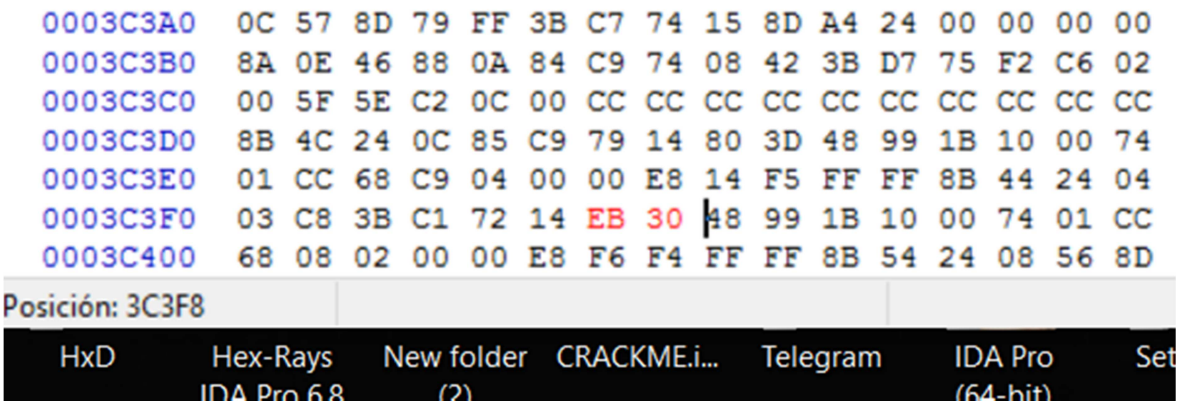
On Internet, we find how to fix it. Let's see if it works well. Make a backup of all saved original modules.



Open **ida.wll** in a hex editor like HXD with admin permission.



Change the bytes **80 3D** from offset **0x3C3F6** by **EB 30**.



Copy the original one in other place before editing it and then rename it and copy it in the same folder as backup.

Open **ida64.wll** and do the same in **0x41606,** change **80 3D** by **EB 30**.

Let's see if the bug is gone.

Press ALT + M on an instruction and then right click and voilà it doesn't crash. I hope it doesn't cause side effects. ☺

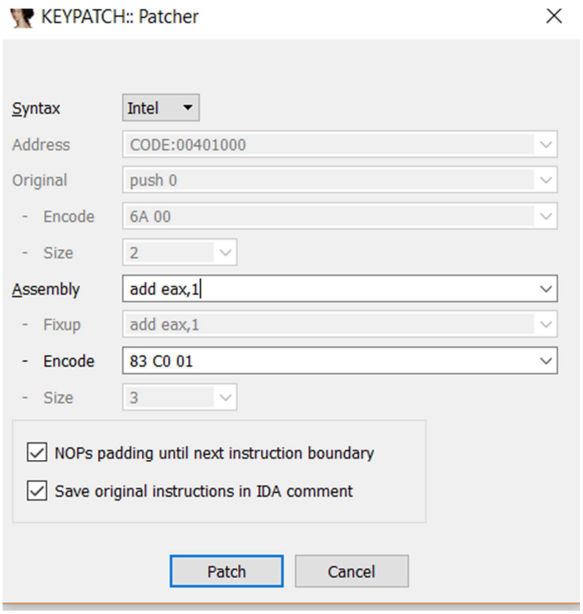
I didn't want to show a bunch of theory from the beginning. That's why I mixed it with some exercises, but we need to see some important flags.

FLAGS

CARRY FLAG

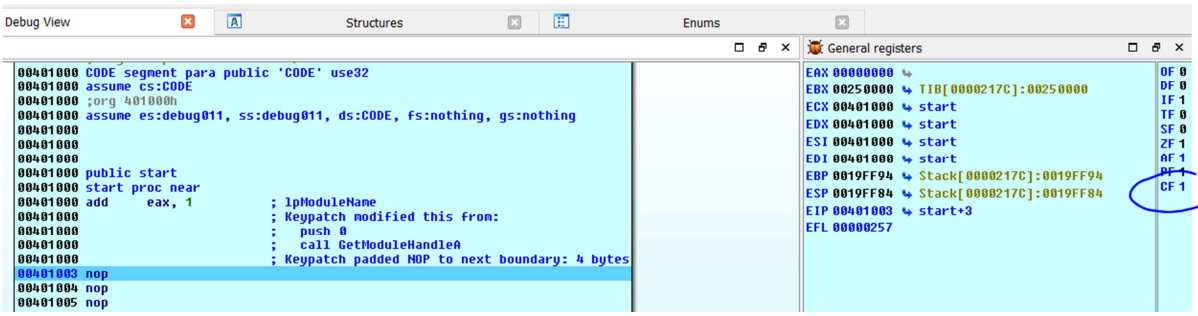
We saw something about the CARRY FLAG in the previous chapter. It activates when the result is negative there is overflow in some addition. Let's see some examples in the debugger.

If we run the crackme.exe in IDA as debugger and stop at the Entry Point.



If the graphic breaks down, right click - CREATE FUNCTION at the beginning of it to fix it. Right click - MODIFY VALUE and assemble EAX=0xFFFFFFFF.

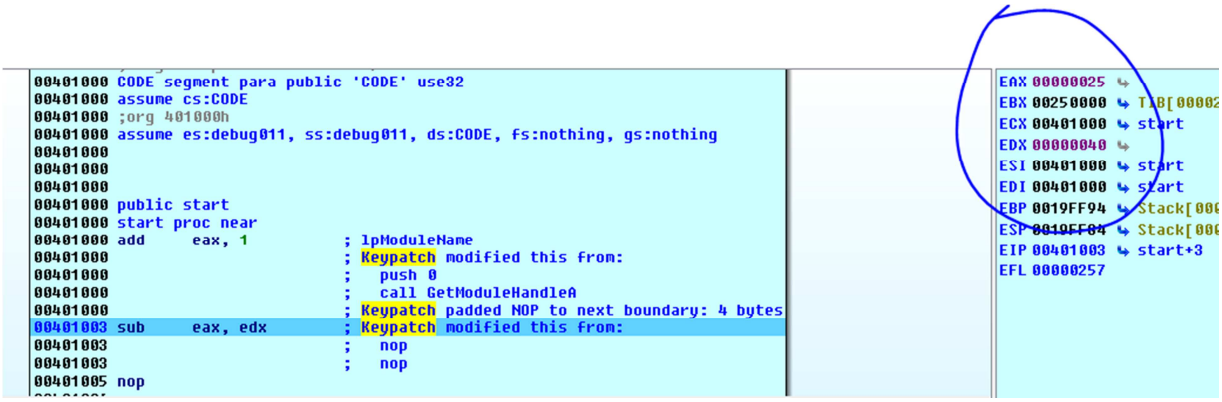
Trace the instruction with F8 to execute it. We see that CF activates and overflows the greater positive value.



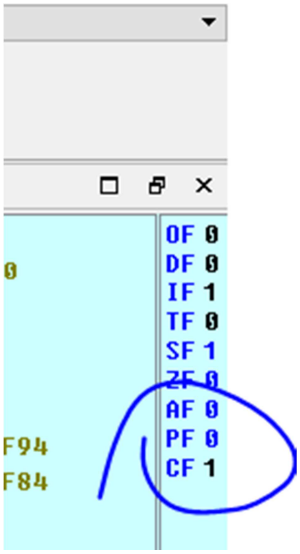
The same happens if we assemble **SUB EAX, EDX** below.

```
00401000 assume es:debug011, ss:debug011, ds:CODE, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 public start
00401000 start proc near
00401000 add     eax, 1          ; lpModuleName
00401000                         ; Keypatch modified this from:
00401000                         ;   push 0
00401000                         ;   call GetModuleHandleA
00401000                         ; Keypatch padded NOP to next boundary: 4 bytes
00401003 sub     eax, edx        ; Keypatch modified this from:
00401003                         ;   nop
00401003                         ;   nop
```

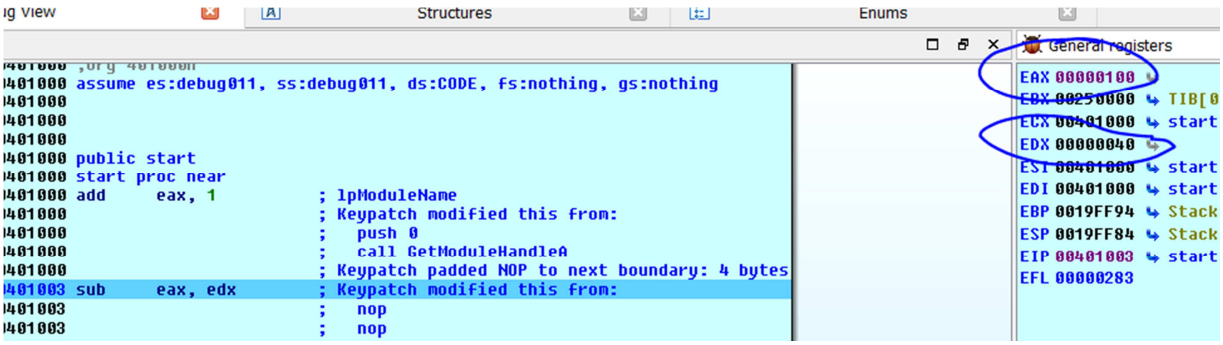
When subtracting two positive values and the result is negative, CF will activate because the result is wrong.



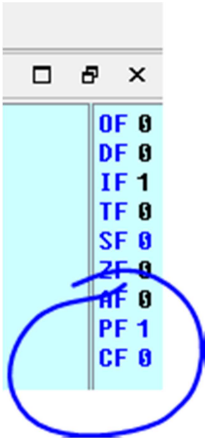
Change EAX by 0x25 and EDX by 0x40 and when we trace it with F8 to see if CF activates.



CF activated. If I change it to subtract again. Right click -SET IP on the SUB EAX, EDX instruction and change EAX by 0x100.



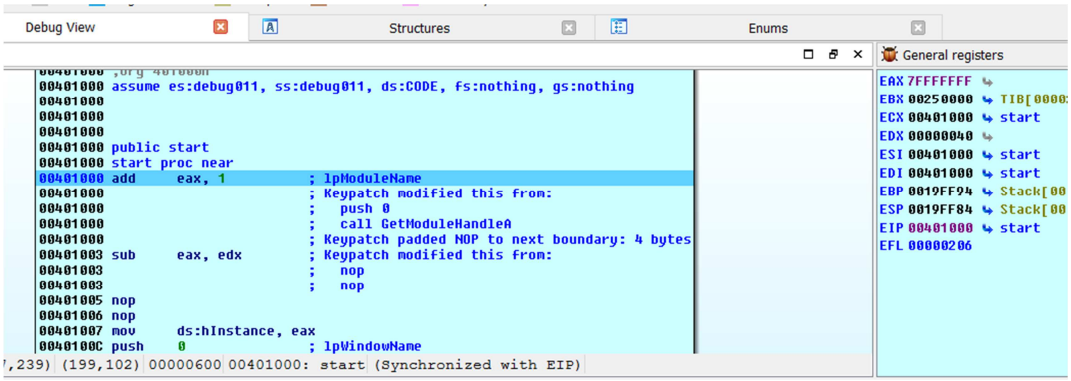
Press F8.



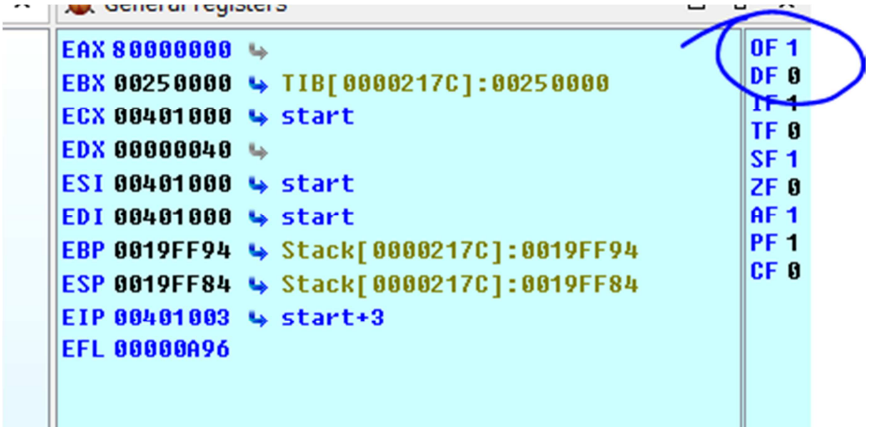
It wasn't activated. As a general conclusion, we can think that in a formula with unsigned values if CF activates means there was an error of any type.

OVERFLOW FLAG.

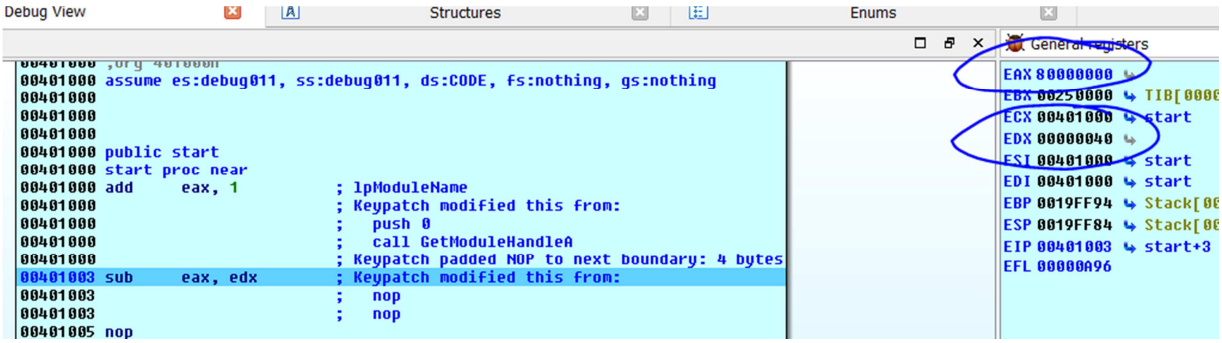
It is similar to CF, but with signed operations. Let's change EIP by ADD EAX,1 and EAX by 0xFFFFFFFF.



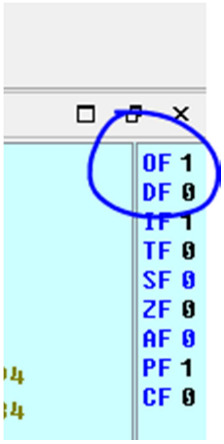
Press F8.



The Overflow Flag activated because adding 1 to the greater positive 0xFFFFFFFF in a signed operation the result is negative and false.



If I subtract EAX and EDX with these values.

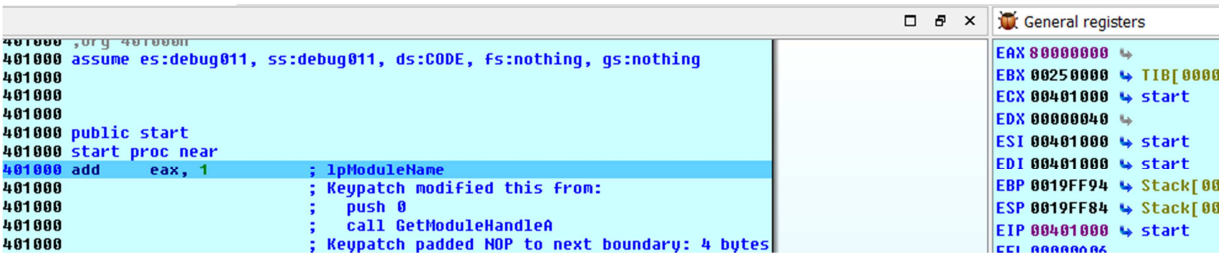


It also activates because the greater negative 0x80000000 - 0x40 is a too big positive value and the result is wrong.

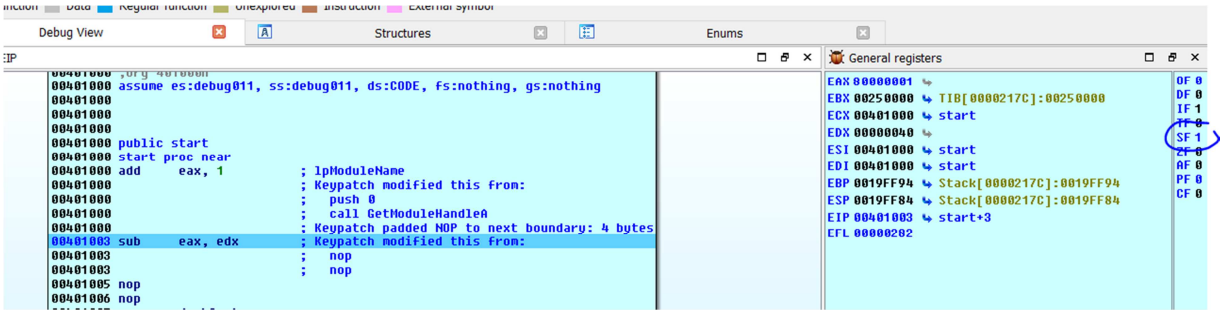
We can conclude that if OF activates, it means that there was a signed operation error.

SIGNED FLAG

This is simple. It activates when the result of an operation is negative in any case. It just determines the sign. It doesn't see if the result is right or wrong.



0x8000000 + 0x1 is in the negative numbers range. The result is 0x8000001 and it activates the SF. OF and CF don't activate when there isn't any error in the signed or unsigned operation.

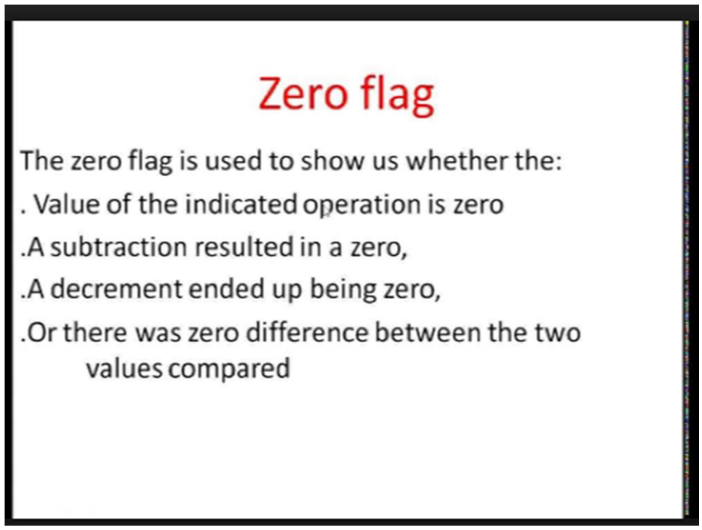


When the processor executes a two-register operation instruction, it doesn't know if they are signed or unsigned. We know that because we see the next conditional jumps. In any operation, it will evaluate the instruction as signed or unsigned at the same time and it will vary the necessary flags.

As the conditional jumps depend on the flags, the program will see the result of the unsigned CF or the signed OF according to the jumps. If there is an unsigned JB, it will only see the CF and it doesn't care the OF although both change.

ZERO FLAG

This is unsigned.

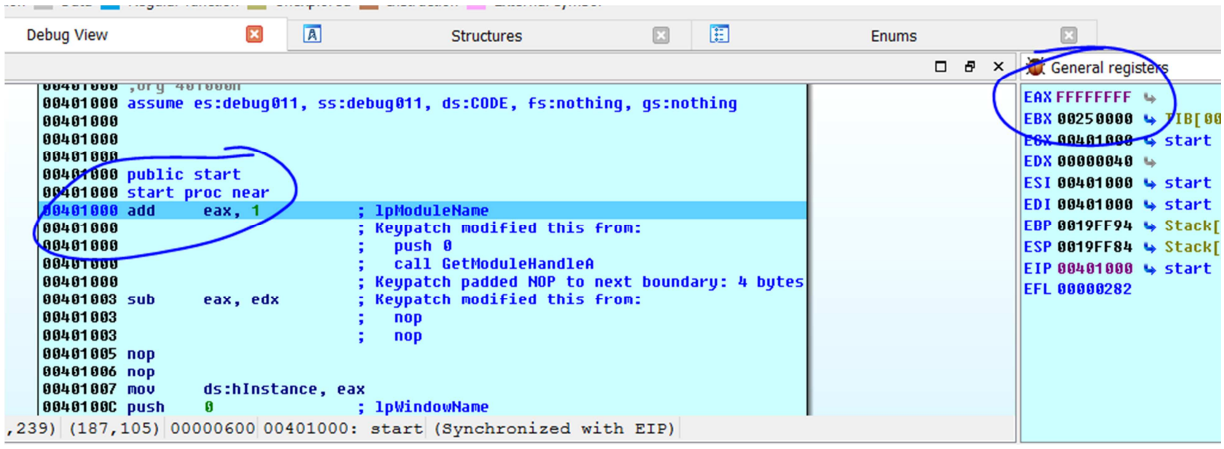


It activates when:

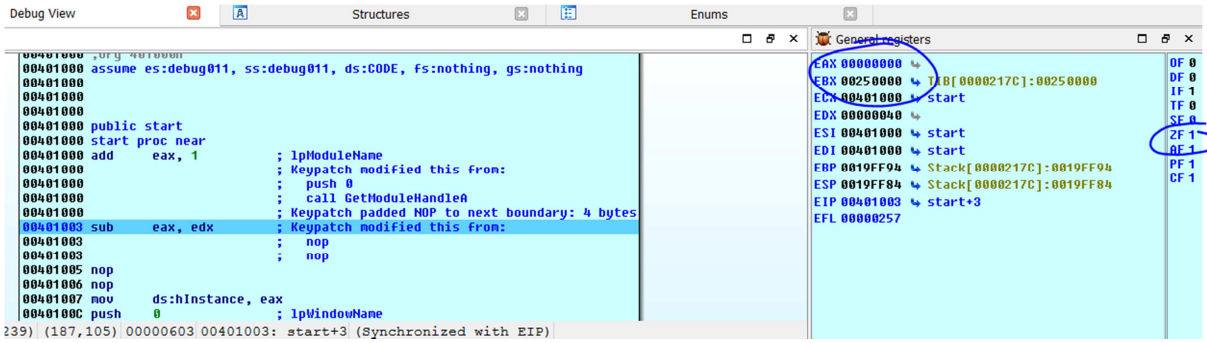
-Both members are equal in a comparison which is a subtraction internally.

-There is an increase or decrease and the result is 0 or in a subtraction whose result is 0.

We can try it.

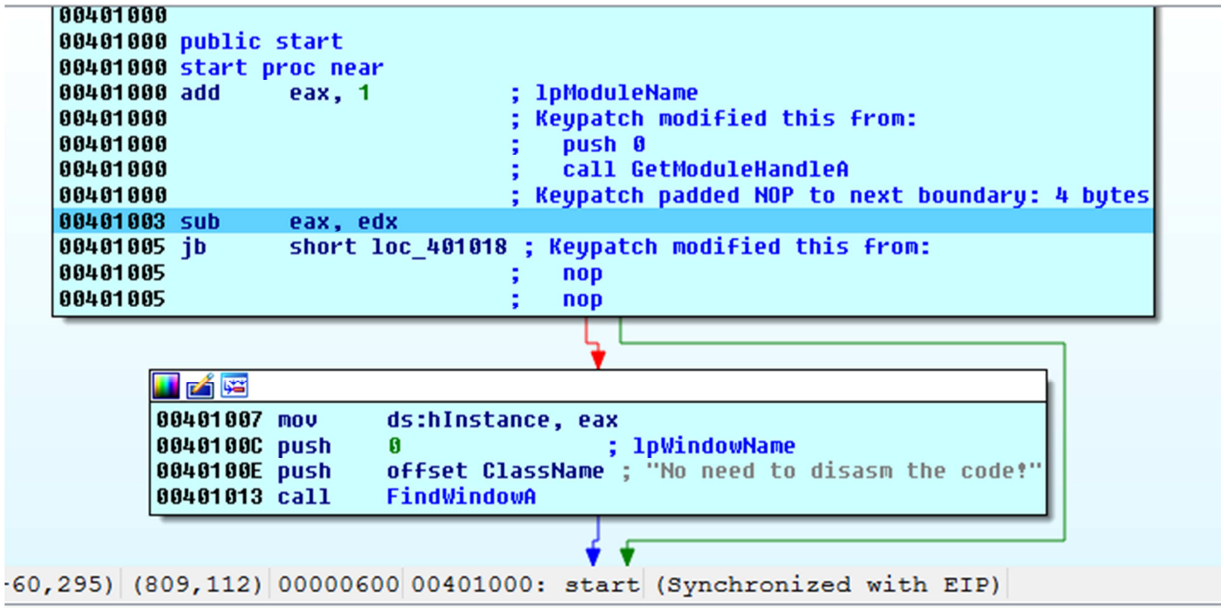


I change EAX by 0xFFFFFFFF and if I add it 1, what happens?

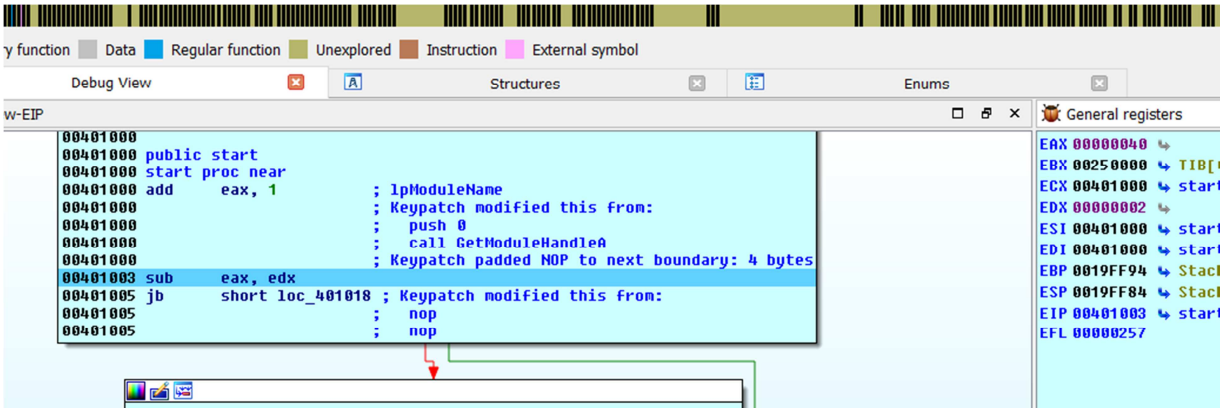


The ZF activates because the result is 0 and if we consider both unsigned, CF activates too because the greater positive value overflows. OF didn't activate because if both are signed -1 + 1 = 0 and there is no error. SF didn't activate either because the result wasn't negative.

Those flags are the most important. Let's see what happens if we assemble a conditional jump.

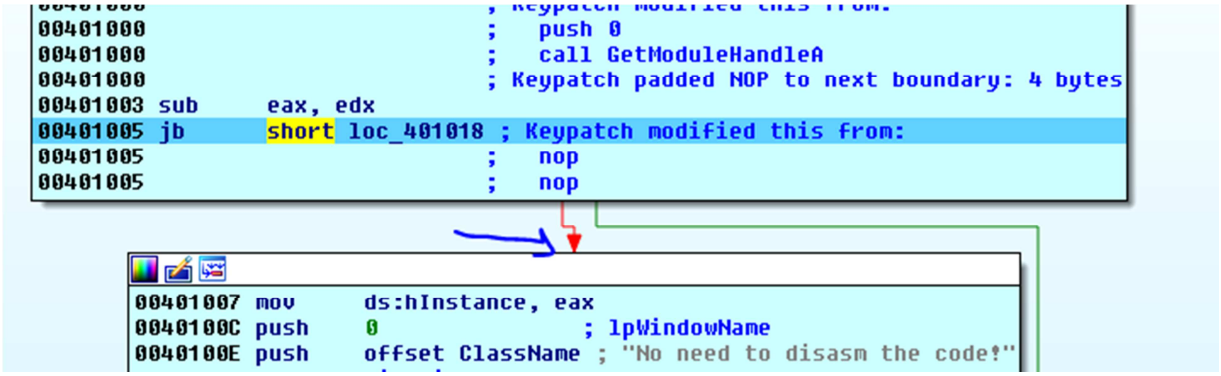


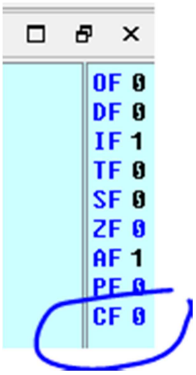
I change the instructions to have a SUB EAX, EDX and then I assemble a JB 0x401018.



I change EAX=0x40 and EDX=0x2 then I execute the SUB with F8.

The red arrow blinks because EAX is greater than EDX and the jump won't be taken. Let's see the flags.

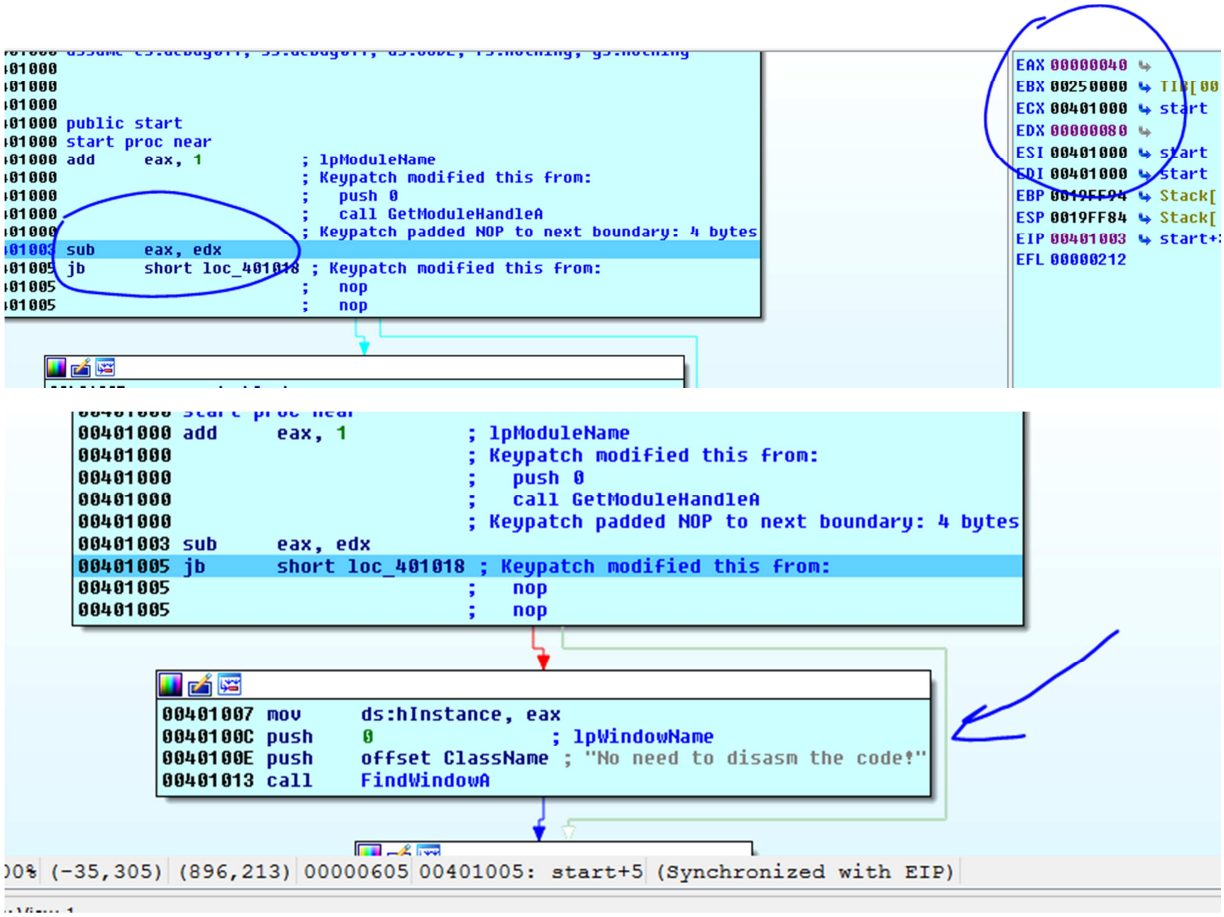




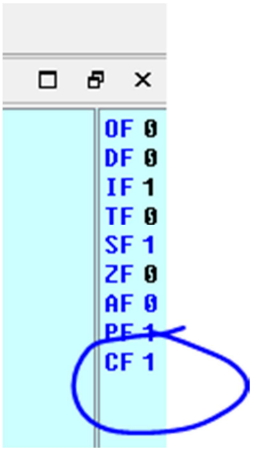
JB is unsigned and it jumps if CF is activated. As CF didn't activate because the operation was correct between two positive numbers and the result is positive which means that the first one is greater than the second one and it doesn't jump.

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

But if we change EAX by 0x40 and EDX by 0x80, and we subtract them again.

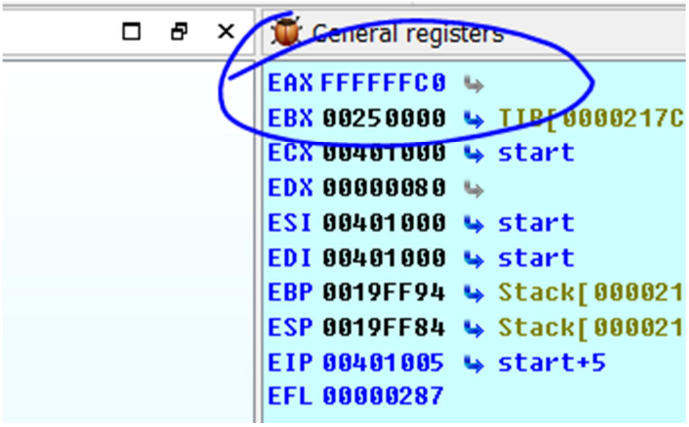


Here, as EAX is less than EDX, it won't jump and it will take the green arrow.

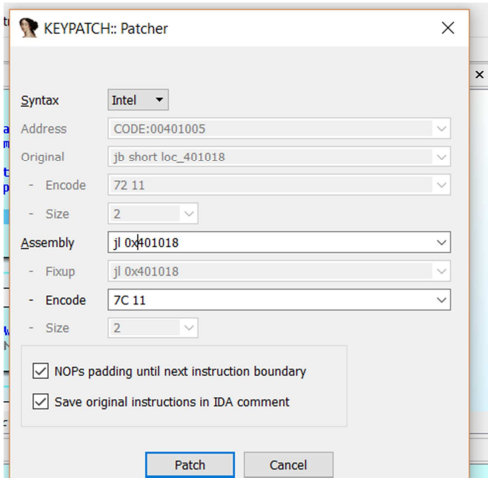


As JB sees the CF, it will jump because it is activated. The result of the unsigned operation was negative and produced the error.

SF activated too because the result was negative and OF didn't activate because if both are unsigned the operation doesn't produce an error. 0x40 – 0x80 is negative as the result.



JB jumps depending on CF state, but if I change it by JL.



In that case, it changes and follows the green arrow because the first one is less than the second one, but which flag does **JL** see?

JL jumps if SF is not 0 and here, it is 1 and it won't jump. It is logical because the first member is less than the second one and the SUB is like a CMP but this saves the result. When the first one is less than the second one it will also jump.

The conclusion of this topic is that it isn't necessary to see the flags to know what will happen in a conditional jump. That belongs to the internal function. We just need to know that if they are equal, a JZ will jump. If it is less and unsigned it will jump if it is a JB. If it less and signed a JL will jump and so on. We just need to see the third column in the signed and unsigned jump table, but it's good to study that in deep.

Ricardo Narvaja

Translated by: @IvinsonCLS