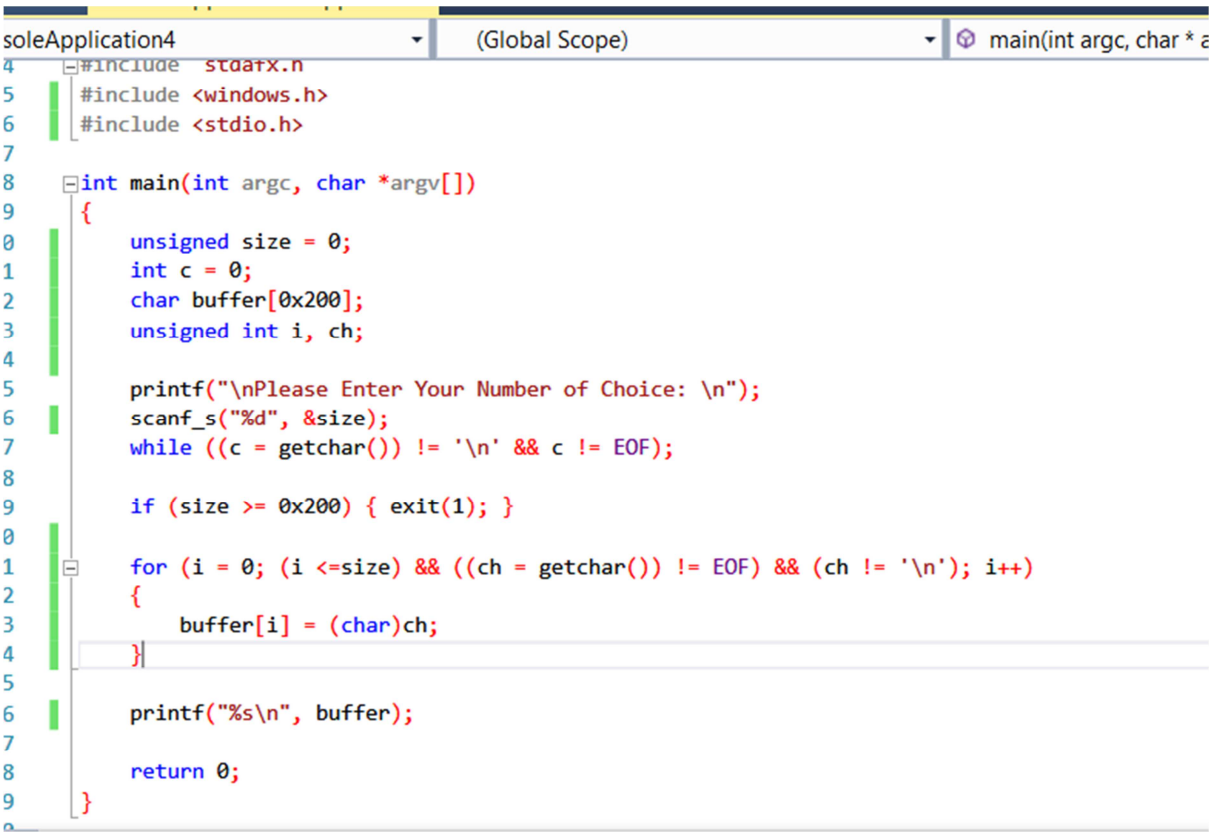
PART 31

Before doing the exercise of the previous part, I will explain more about some points that were a little dark in the speed of solving it and then, we will create the POC. One of the points that is unclear is because sometimes we accept the length of the array that IDA offers us by right-clicking - ARRAY and sometimes we question it as in the previous exercise and we put a greater value. Obviously, that is done by experience more than anything but we will try to explain it with a couple of examples.

Let's look at this simple example.

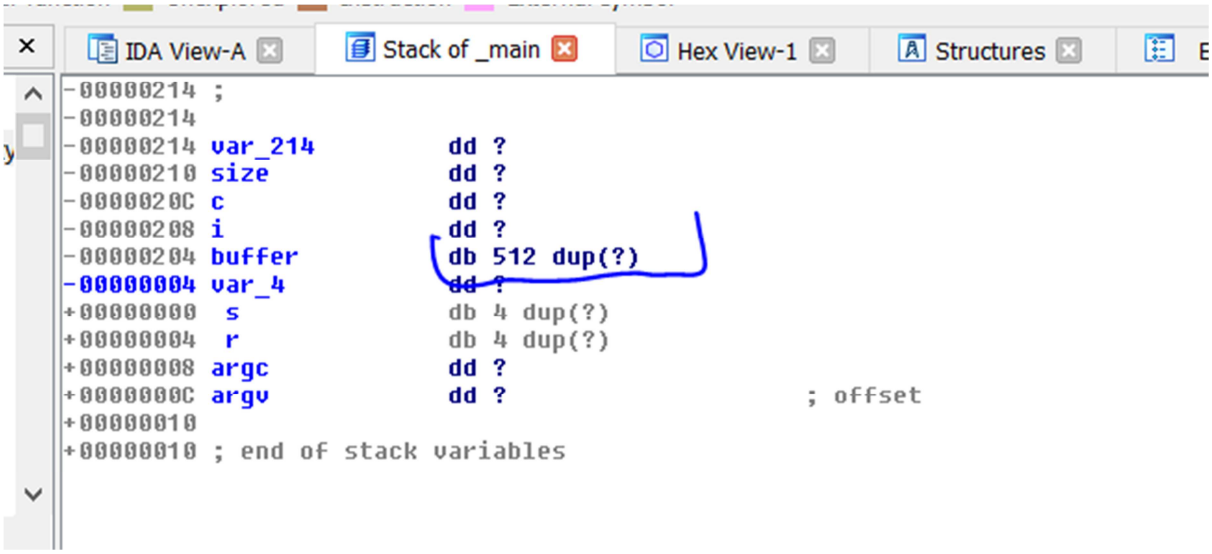


It has a keyboard input for the size which is checked for not being equal or greater than 0x200 (unsigned).

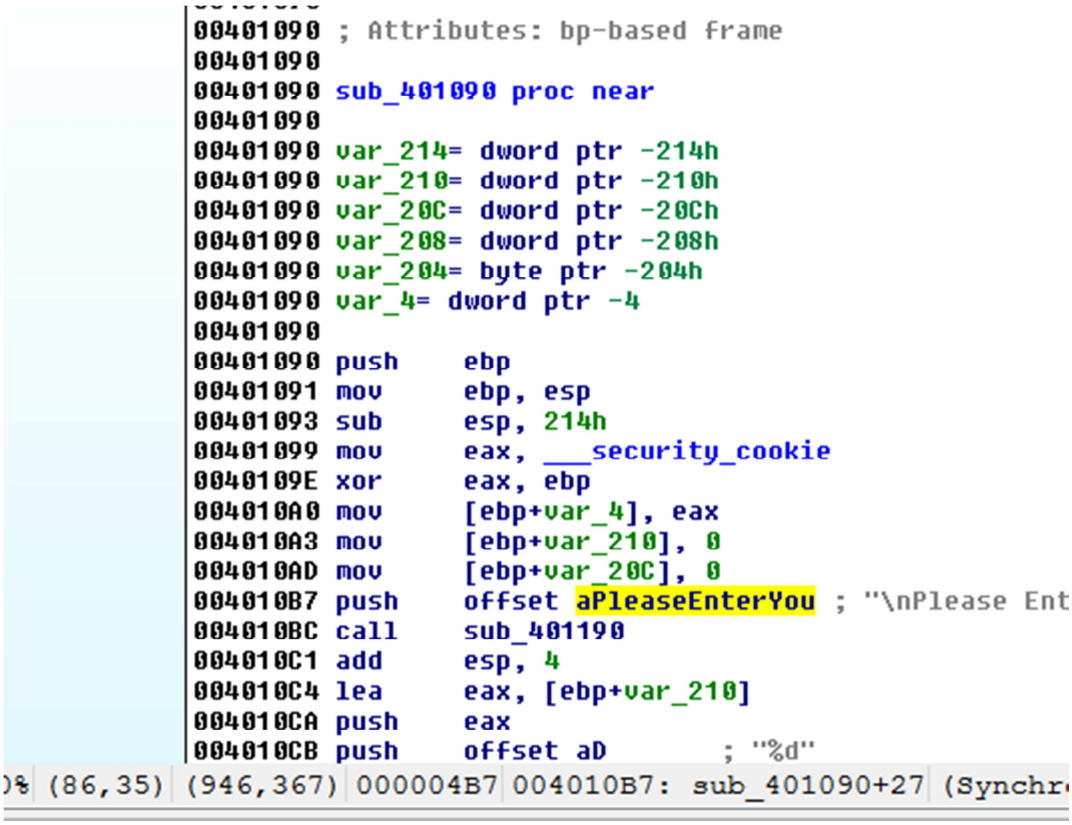
Then, a loop is repeated the same times than size to read from the keyboard a character at a time with **getchar**.

This would not be a buffer overflow because the size is unsigned. So, there are no sign problems when comparing it to 0x200.

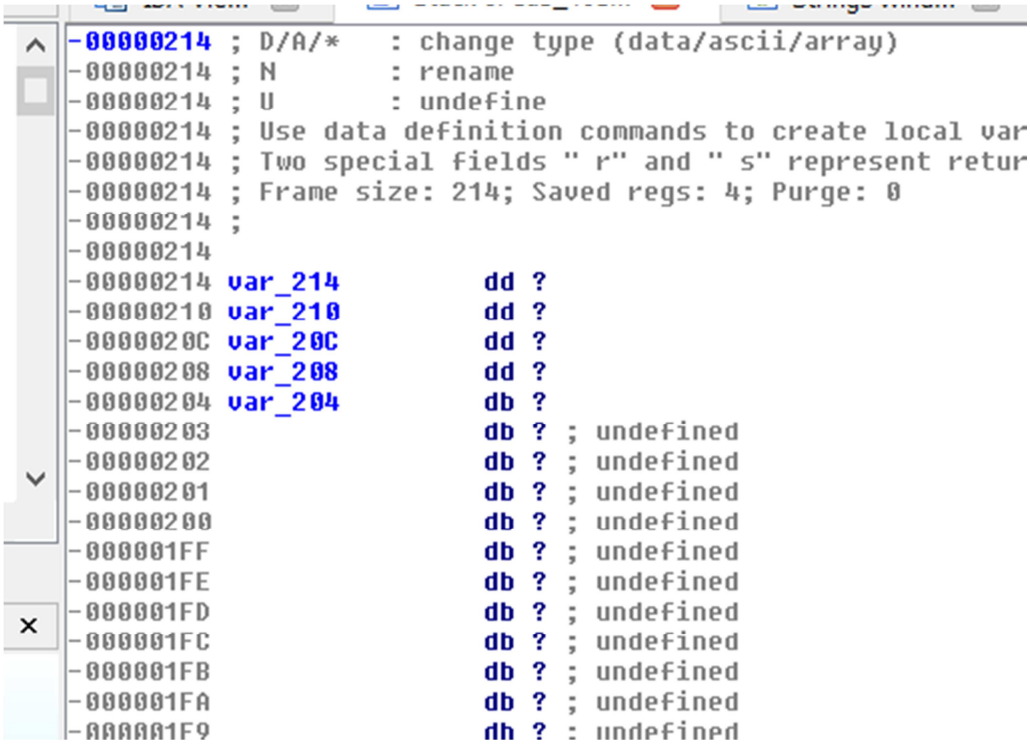
Then, it will read characters and copy them to the buffer. As its size is 0x200, it will not overflow.



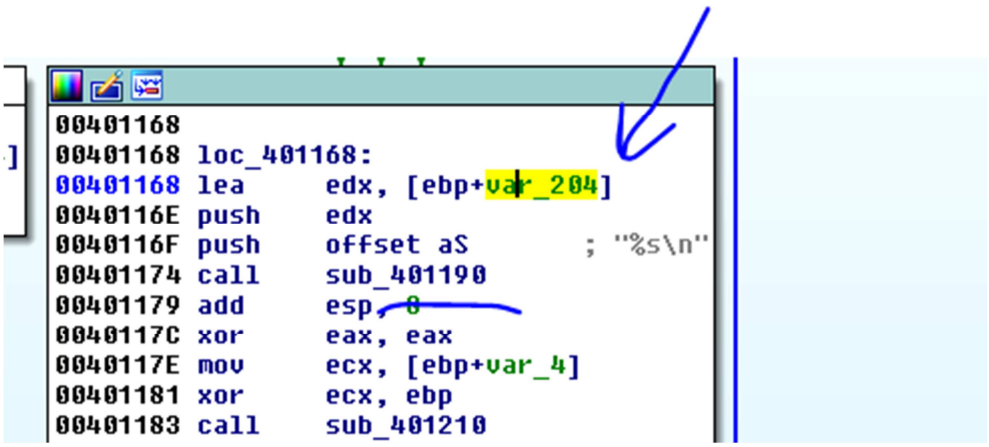
If I see it in IDA, it already detects the buffer correctly (512 decimal is 0x200) because I compiled it with symbols. The same happens if I compile it again without symbols.



Let’s go to the stack view.

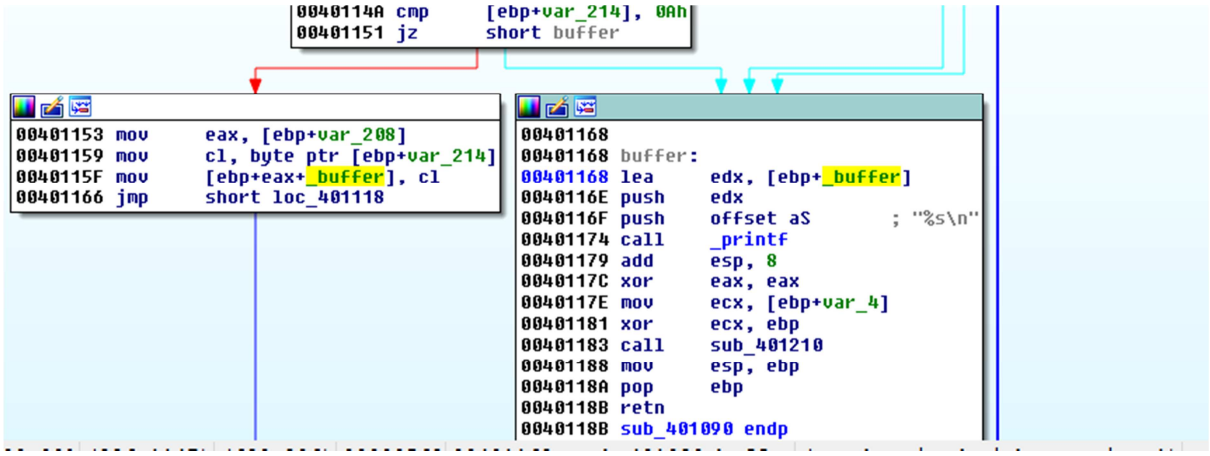


Here, it doesn’t detect the buffer. We have to do it manually. As there is empty space, it is possible that var_204 is the buffer. Let’s see its references.

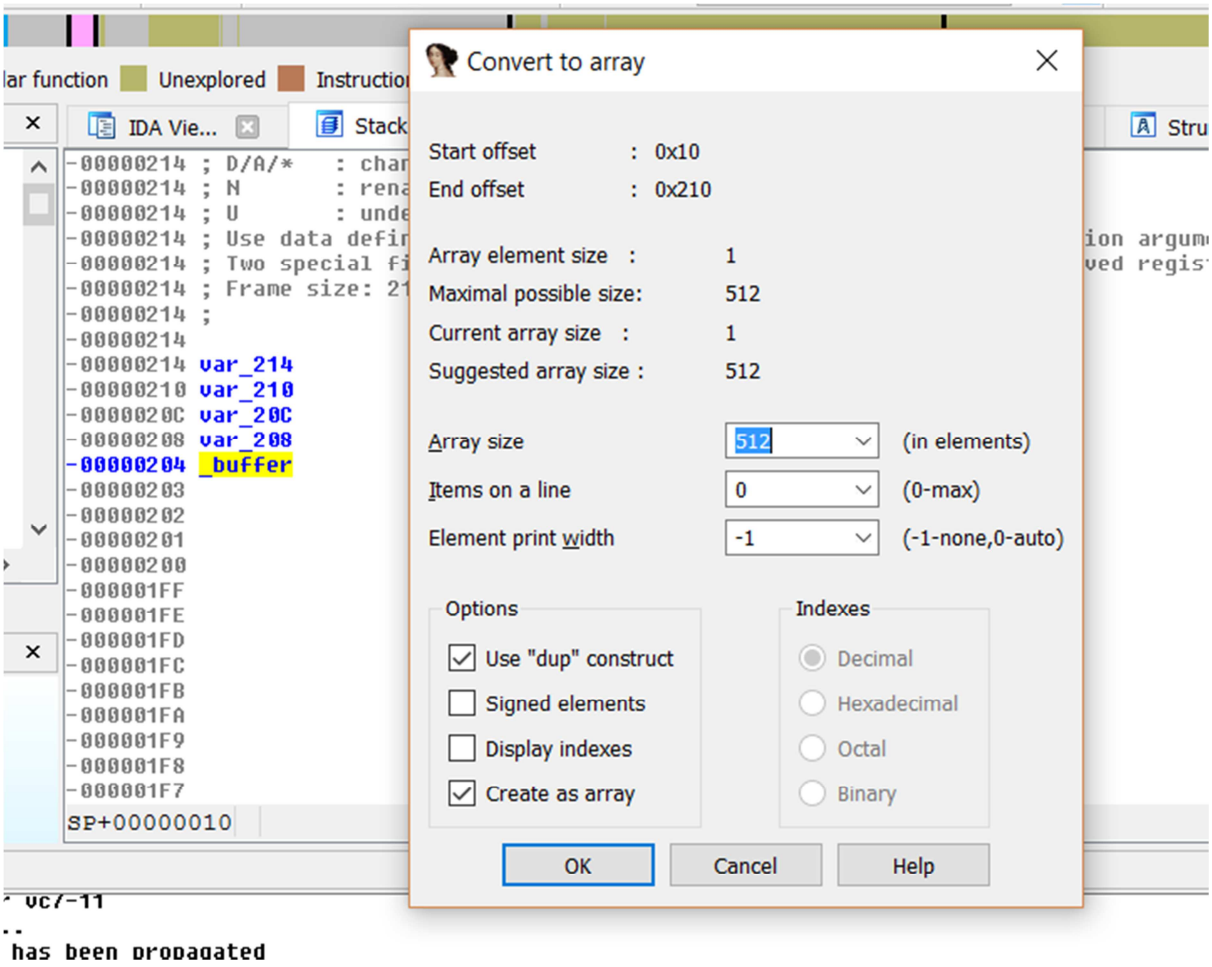


There is a clear reference when it prints the final buffer. It is normally possible that it is a buffer when there is a reference with a LEA. Besides, 0x401190 will be printf.

The other reference to buffer is inside the loop when it fills it with the bytes it reads with getchar.

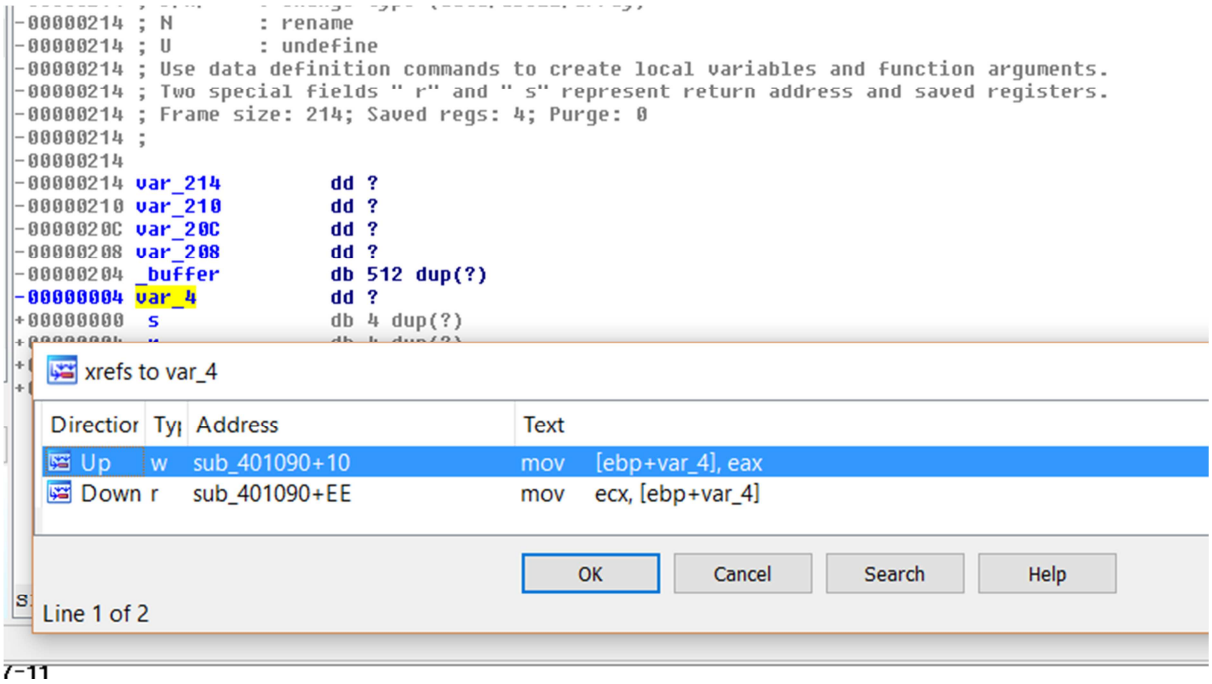


Now, let’s go to the stack view and see the buffer size.



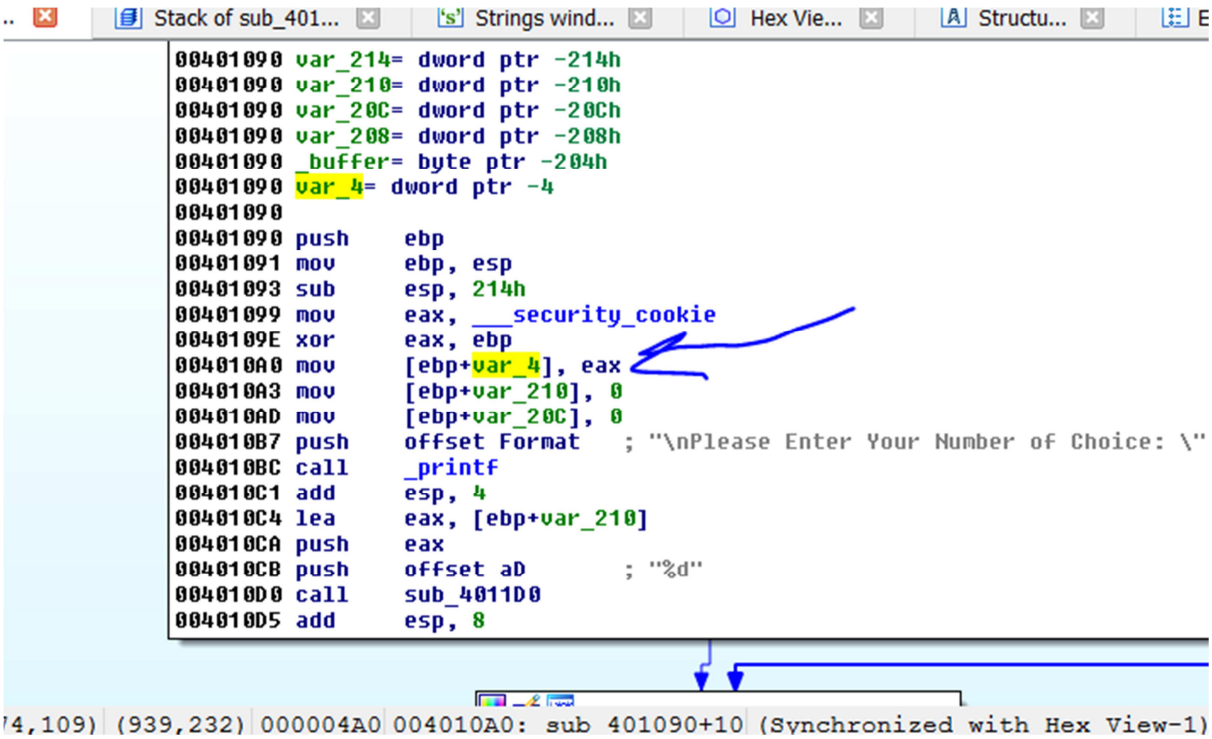
We see it tells us 512 would be OK, but why 512?

Because IDA checks the empty space that there is until the next variable that is var_4. Now, the method is to see where it saves a value for the first time and where that var_4 is used later. Let's see.



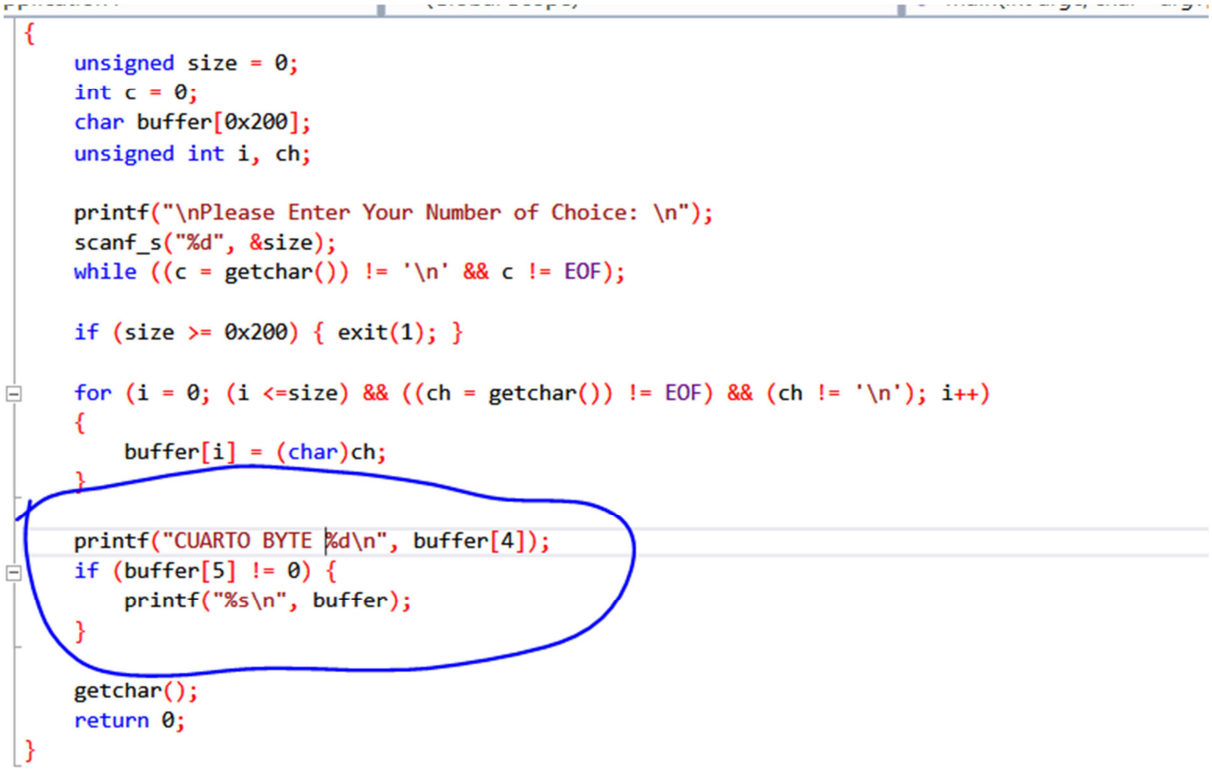
There are two places. One is where it starts with a value (the SECURITY COOKIE is saved) and the other where it is read.

Let's go to the first one.



Before filling the buffer in the loop, var_4 gets value and it is not related to the buffer. So, we determine it is an independent variable. I save it as BUFFER1.exe. Here, IDA was not mistaken.

I'll do another example and I'll call it BUFFER2.exe.



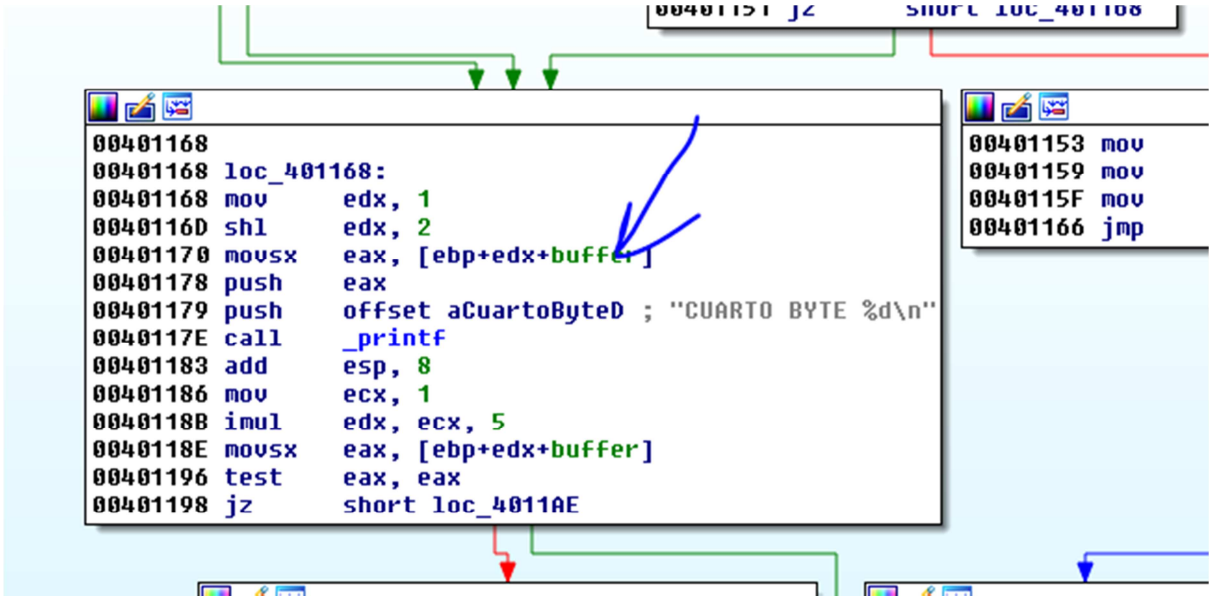
```
{
    unsigned size = 0;
    int c = 0;
    char buffer[0x200];
    unsigned int i, ch;

    printf("\nPlease Enter Your Number of Choice: \n");
    scanf_s("%d", &size);
    while ((c = getchar()) != '\n' && c != EOF);

    if (size >= 0x200) { exit(1); }

    for (i = 0; (i <=size) && ((ch = getchar()) != EOF) && (ch != '\n'); i++)
    {
        buffer[i] = (char)ch;
    }

    printf("CUARTO BYTE %d\n", buffer[4]);
    if (buffer[5] != 0) {
        printf("%s\n", buffer);
    }

    getchar();
    return 0;
}
```

The buffer size is still 0x200. Everything is equal to the previous example, but here, it prints the fourth byte of the buffer and it compares it to the fifth byte to zero and if it is equal, it prints **buffer**.

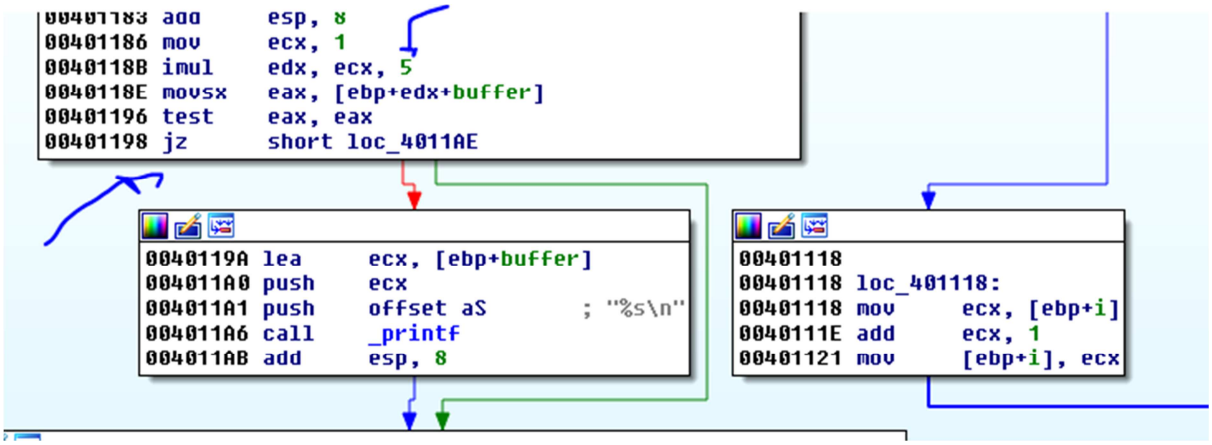
What happens if I compile it with and without symbols?

```
-00000214 ; Two special fields " r" and " s" represent return address
-00000214 ; Frame size: 214; Saved regs: 4; Purge: 0
-00000214 ;
-00000214
-00000214 var_214         dd ?
-00000210 size            dd ?
-0000020C c               dd ?
-00000208 i               dd ?
-00000204 buffer          db 512 dup(?)
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 argc            dd ?
+0000000C argv            dd ?                    ; offset
+00000010
+00000010 ; end of stack variables
```

With symbols, there is no problem. It still detects the buffer as 0x200 and the fourth byte that it prints.



It doesn't take it as an independent variable which was my idea. It reads the fourth byte of the buffer to which it accesses adding 4 (SHL EDX, 2 is equal to multiply EDX * 4) and then, it adds it to buffer the start address to look for the fourth byte value.



Here, it multiplies EDX by 5 and it adds it to the buffer start to point and it moves the content to EAX with MOVSX. If it is different to zero, it doesn't print.

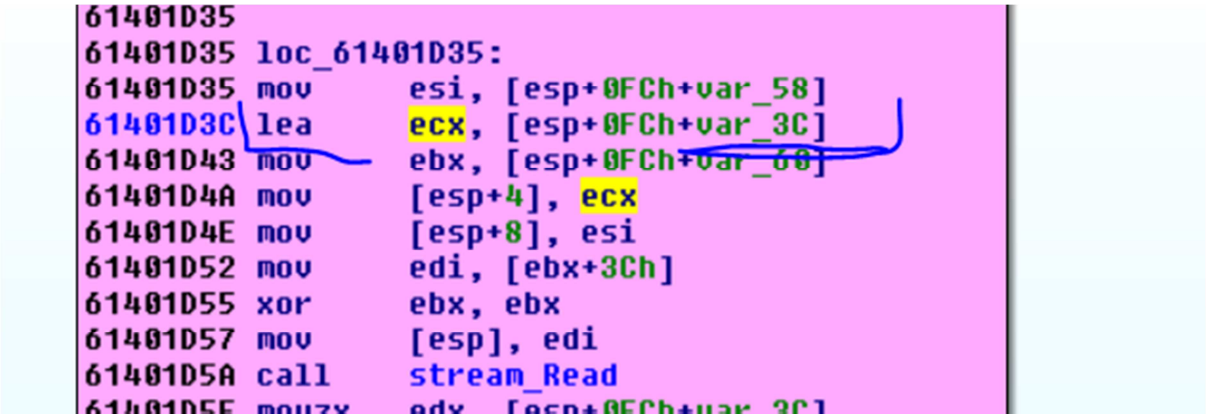
The buffer was not affected and IDA still detects the size well with symbols. Now, let's see it without symbols.

I open it and see that IDA wasn't wrong. It doesn't assign a variable to the fourth and fifth buffer values (although, there are cases that it does)

The size is still 512 because the next variable is the CANARY var_4. If IDA detected the fourth and fifth values as variables, they would not be independent. They are filled when the buffer does. There is no other place to save values there. So, I must consider them as a part of the buffer.

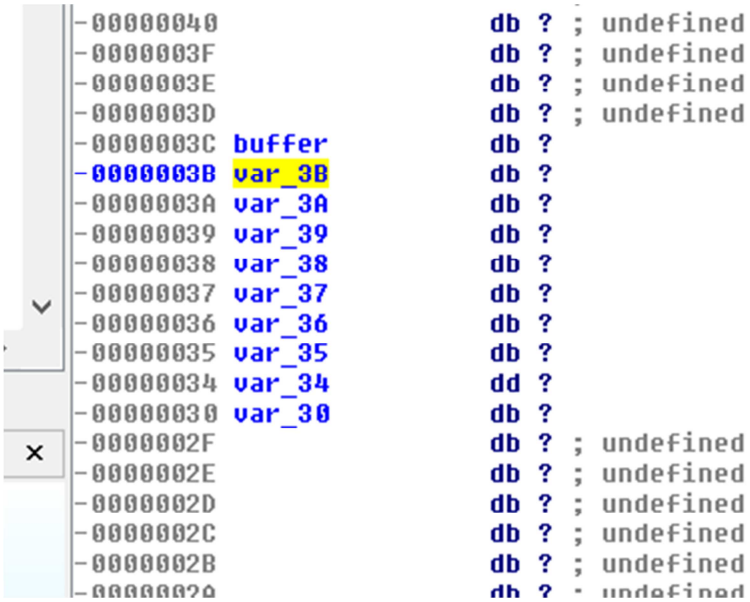
As I can't compile an example where IDA is mistaken, I compare it to the exercise buffer, taking into account that when IDA suggests us a buffer size, we have to look for the first independent variable downwards that would be the real buffer limit.

Let's see the exercise buffer.

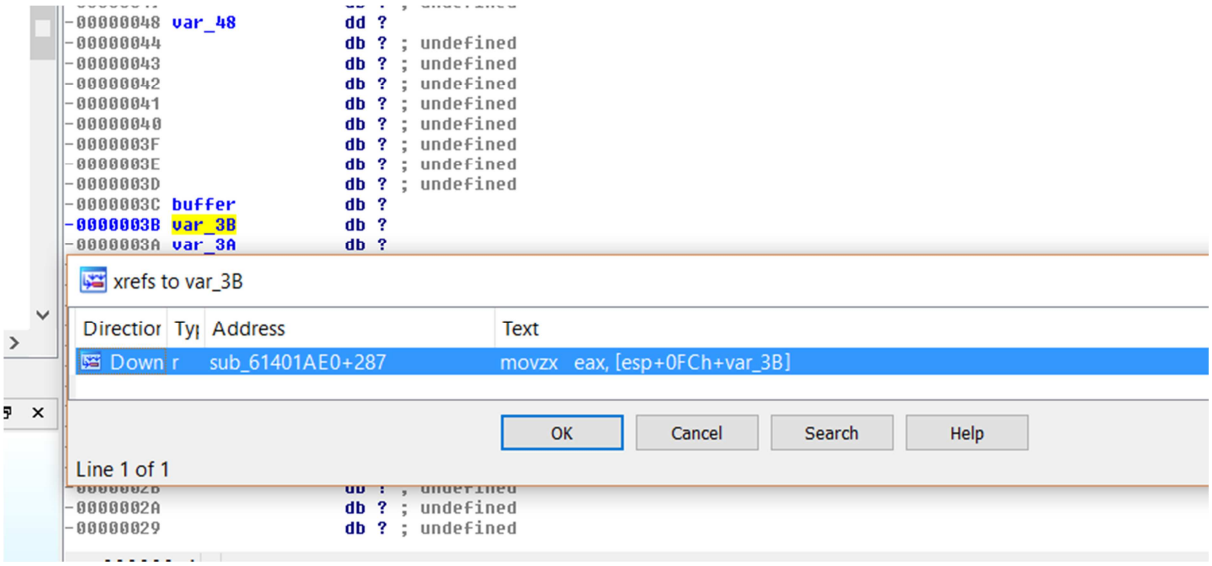


That LEA is possibly a stack buffer that is passed to the stream_Read function. Let's rename it as buffer.

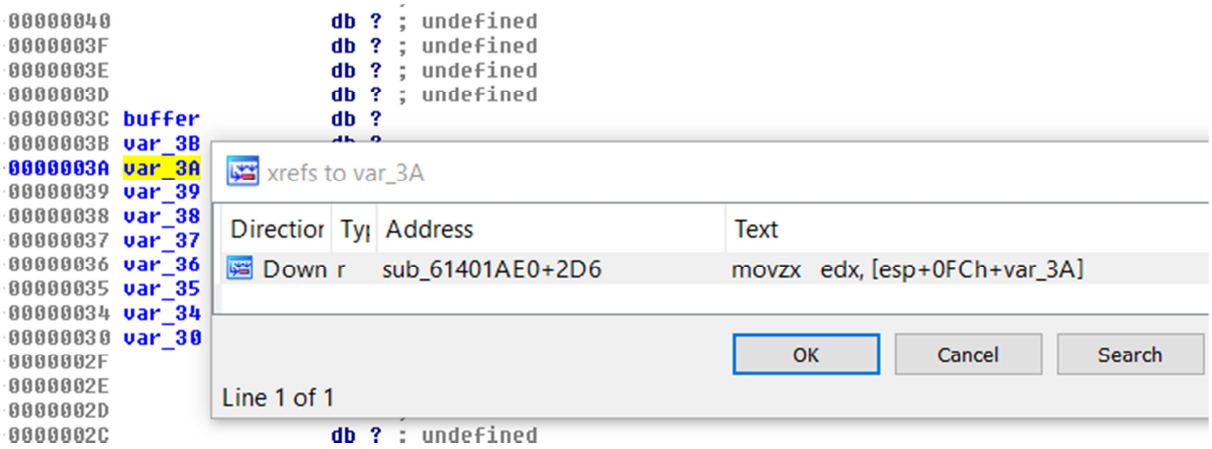
Let's see the variables downwards to find the first independent buffer variable.



I press X on each one.

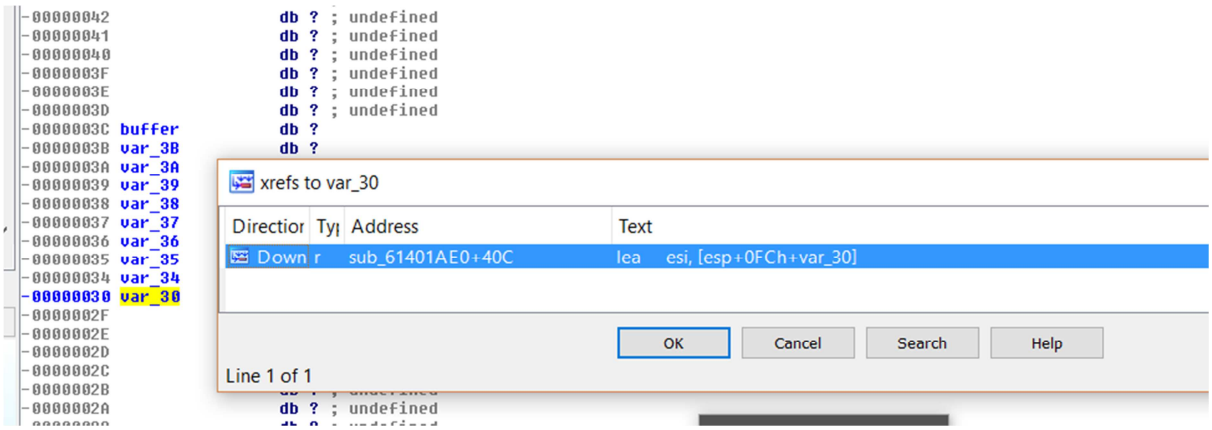


As I placed the cursor on the LEA, I see that variable is used **down** and it doesn't make sense to use it if there is no other reference where a value is saved. The only possible place is when it fills the buffer. The same happens with all variables.

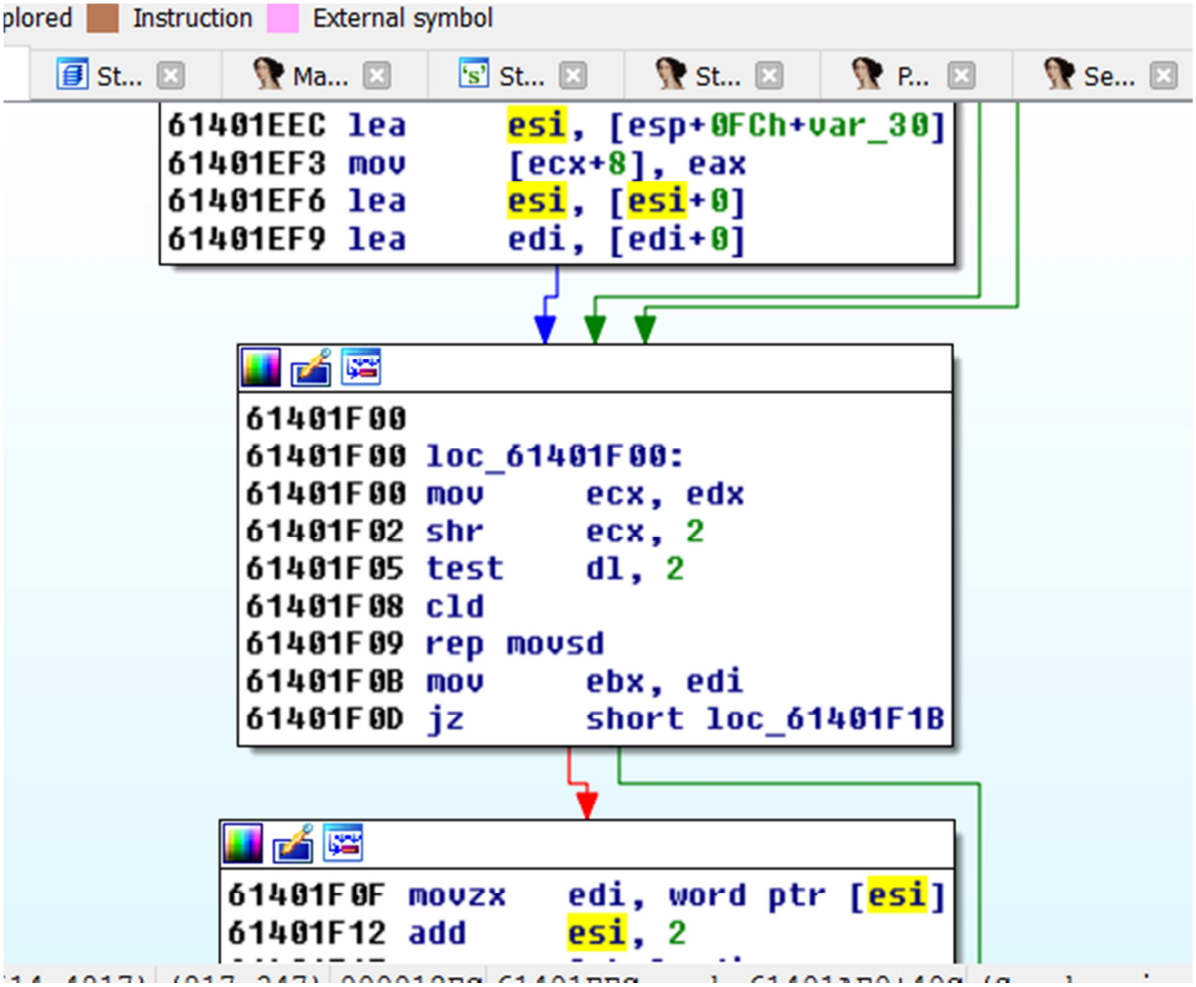


Same case. It belongs to the buffer.

The first one with a different reference is:

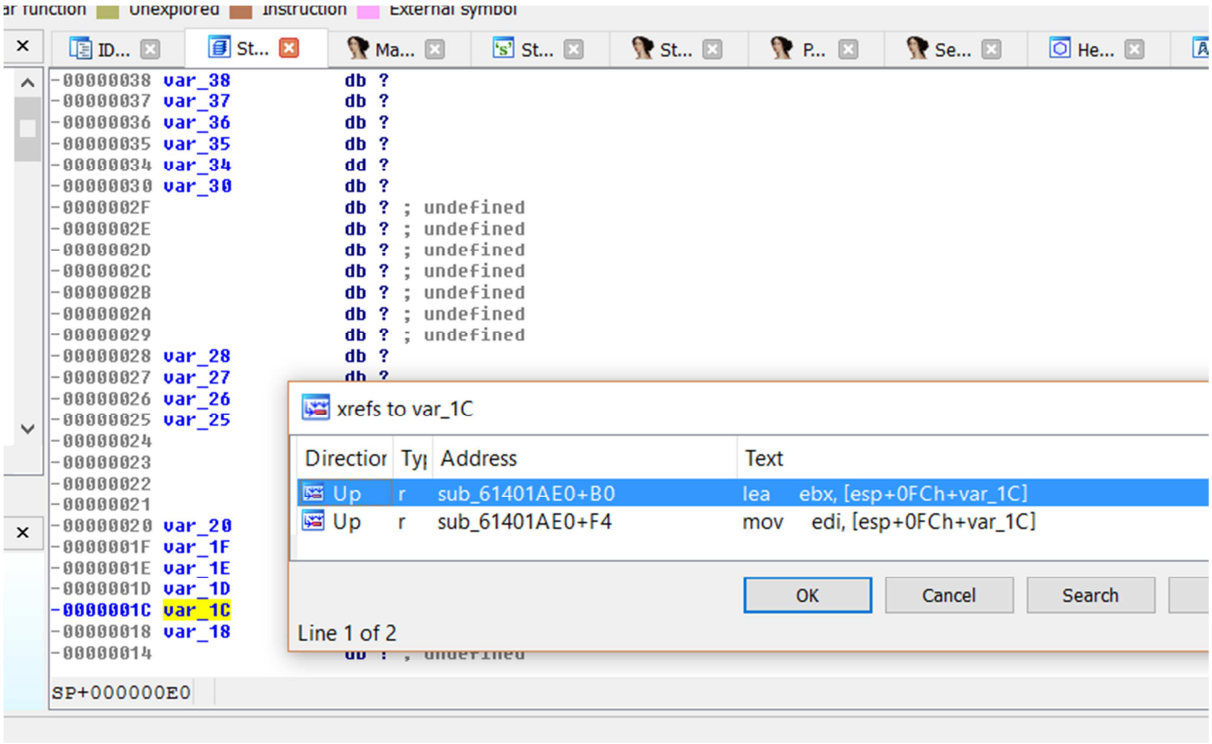


It looks like a buffer because it has a LEA. Let's see.



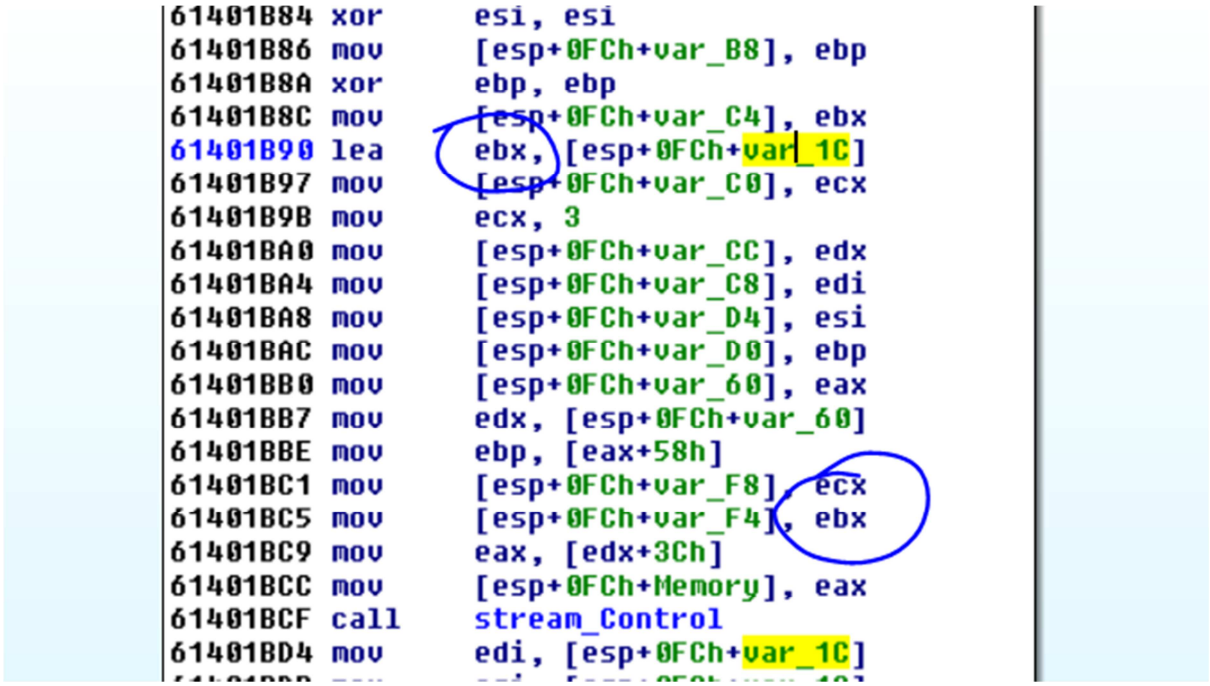
It is really a buffer, but it is part of the previous buffer because it has no any independent place to be filled. The first reference is passed as **source** to a **reps movs** which must have saved values to copy them to another place. It also marks **DOWN** meaning that it is used after filling the original buffer. Let's continue scrolling down.

It continues reading in all next variables below the place where the buffer is filled until here.



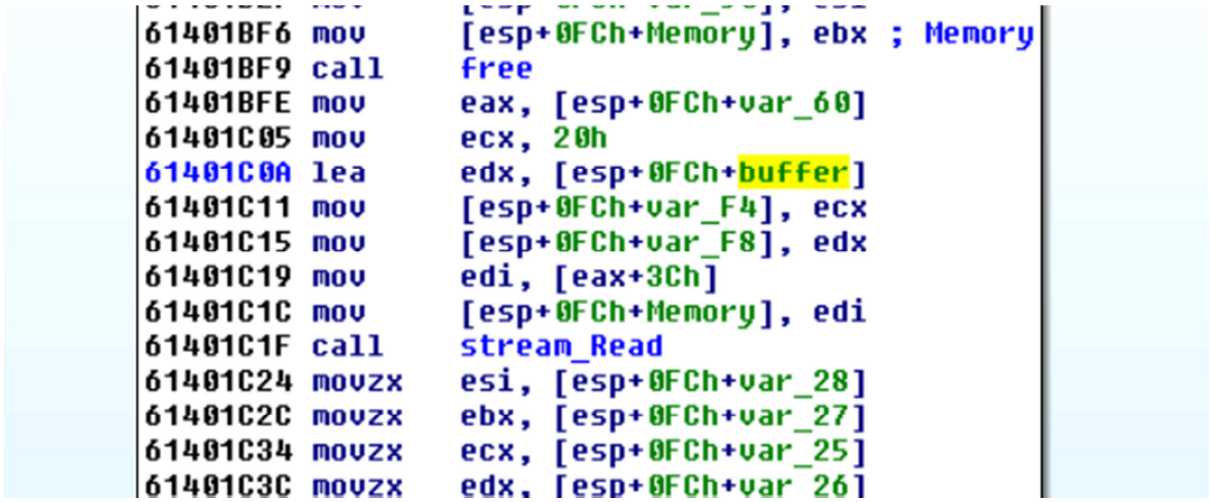
This shows a LEA and UP. It is above the place where the original buffer is filled.

We see that this is another buffer, independent from the original one.



It is passed as an argument to stream_Control and filled there. So, we found the first independent variable.

As I was doing this quickly, I hadn't seen that the other **CALL stream_Read** above with the same buffer.

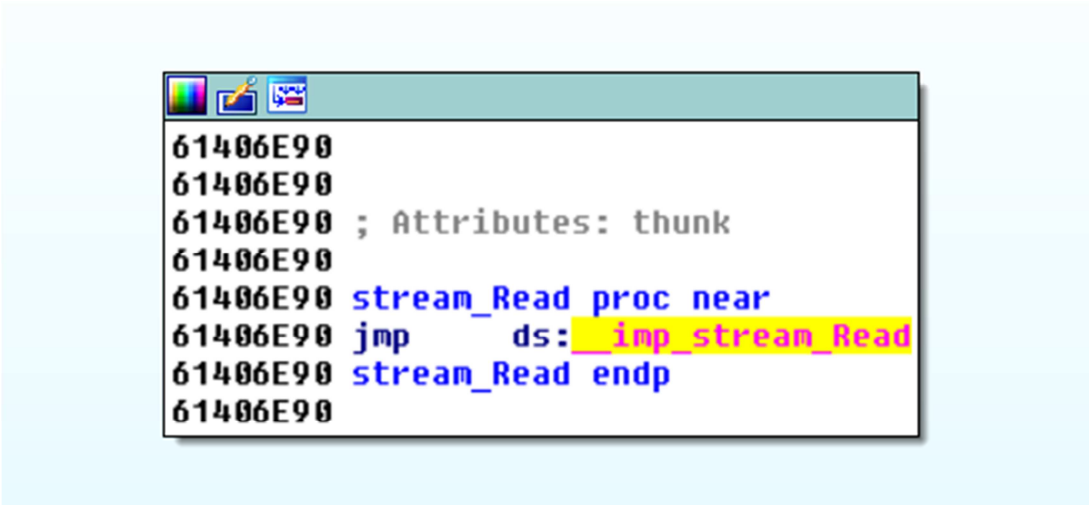


Anyways, the analysis is the same. There are no variables until 1C with references before some of the places where the buffer is filled here or in the other CALL.

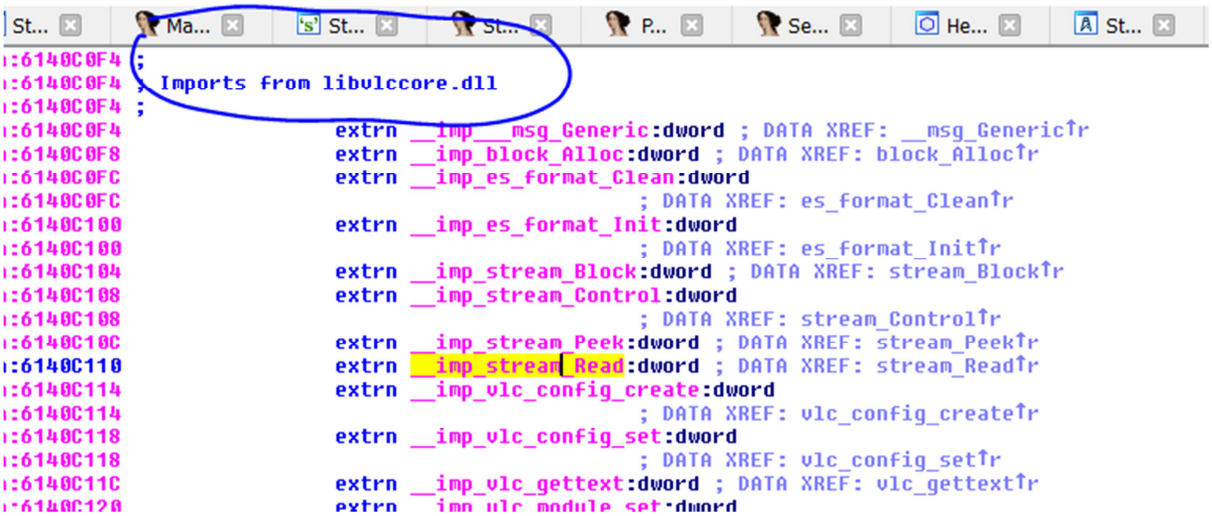
```
-0000003E                db ? ; undefined
-0000003D                db ? ; undefined
-0000003C buffer         db 32 dup(?)
-0000001C var_1C         dd ?
-00000018 var_18         dd ?
-00000014                db ? ; undefined
-00000013                db ? ; undefined
-00000012                db ? ; undefined
```

It is 32 bytes and we could verify that all variables inside the buffer are internal of it and they aren't independent.

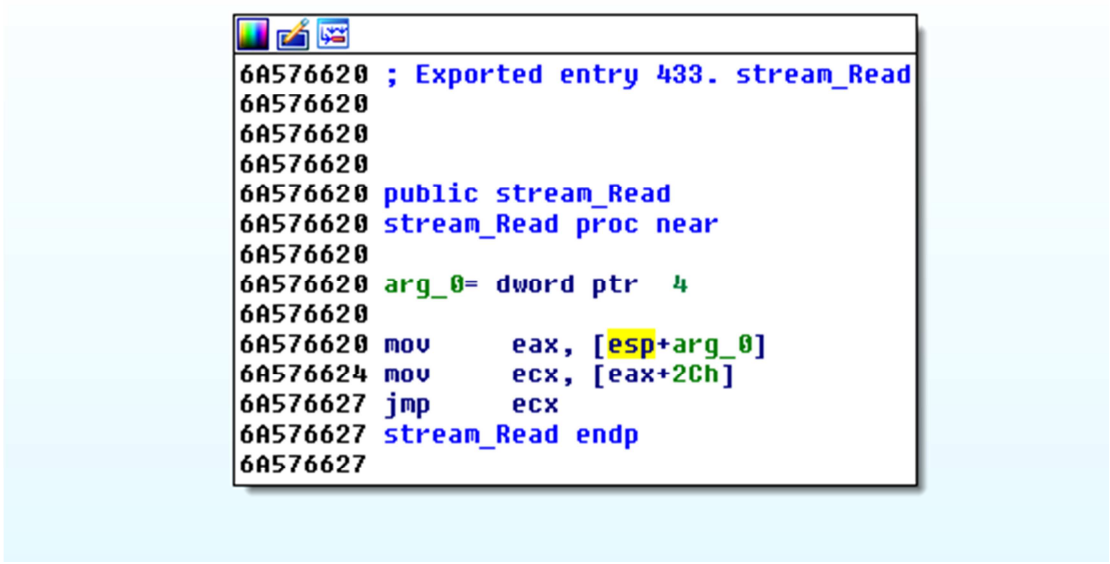
The other stuff I was asked is how I knew stream_Read could write the number of bytes we pass a buffer. Its name suggests that. Besides, the patch limits the value that comes there. If it is greater than 8, it made me suspect that it is the max size it will copy.



There, we see it goes to stream_Read. Press ENTER there.



That is an imported function of libvlccore.dll. So, I find it in the vulnerable version and open it in IDA.

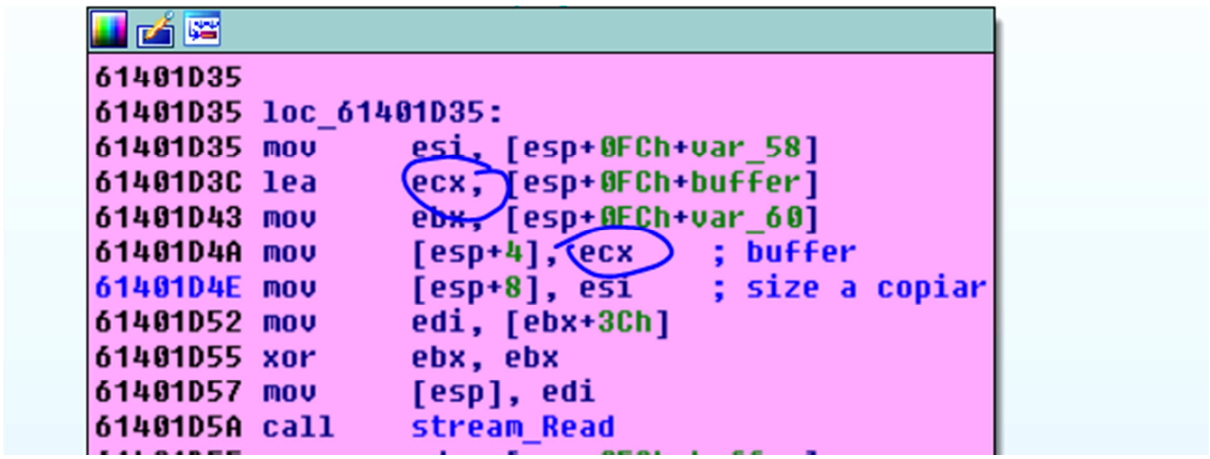


That function depends on an arg_0 which is a constant that comes from a CALL. According to that constant, it decides where to jump. [eax+2c].

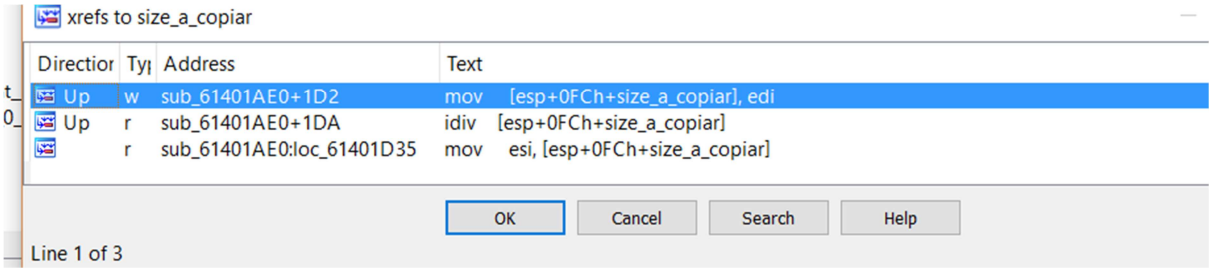
I could reverse it to find out where it jumps, but as I am going to create a POC and for that I have to debug, I will solve it debugging it.

For those who asked what a POC is. It is a **P**roof **O**f **C**oncept that is not a complete exploit, but it shows the vulnerability creating a ty file, in this case, that overflows the 32-byte buffer.

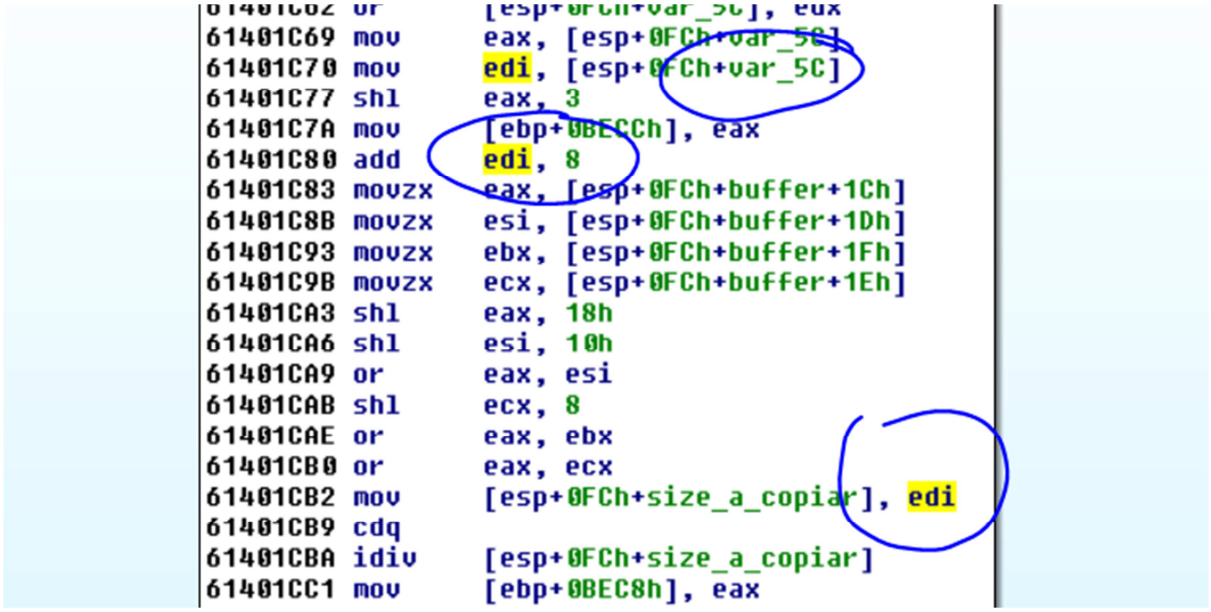
As I have it remotely to the program, I will attach it with the remote debugger. If you have it locally, attach it with the local debugger.



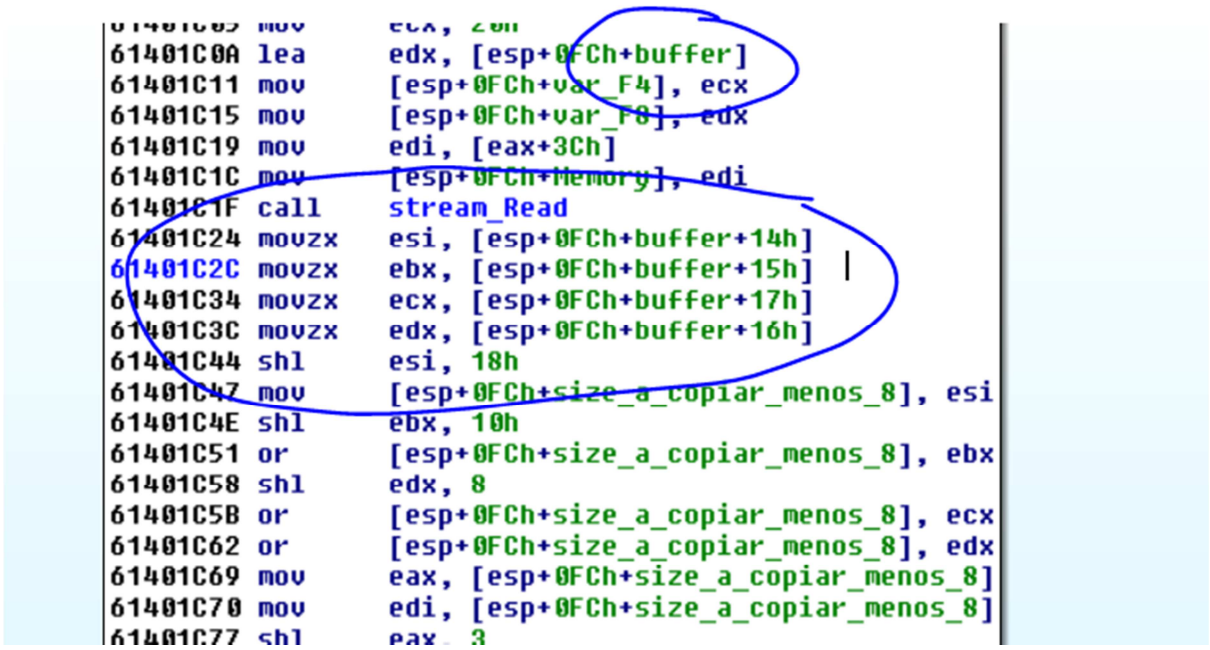
I will see where that ESI value comes from that has the size to copy and it comes from var_58. Rename it as **size_a_copiar** or **size_to_copy**.



We see where it saves it and that it makes a division with IDIV, but let's go to the place it saves it first.

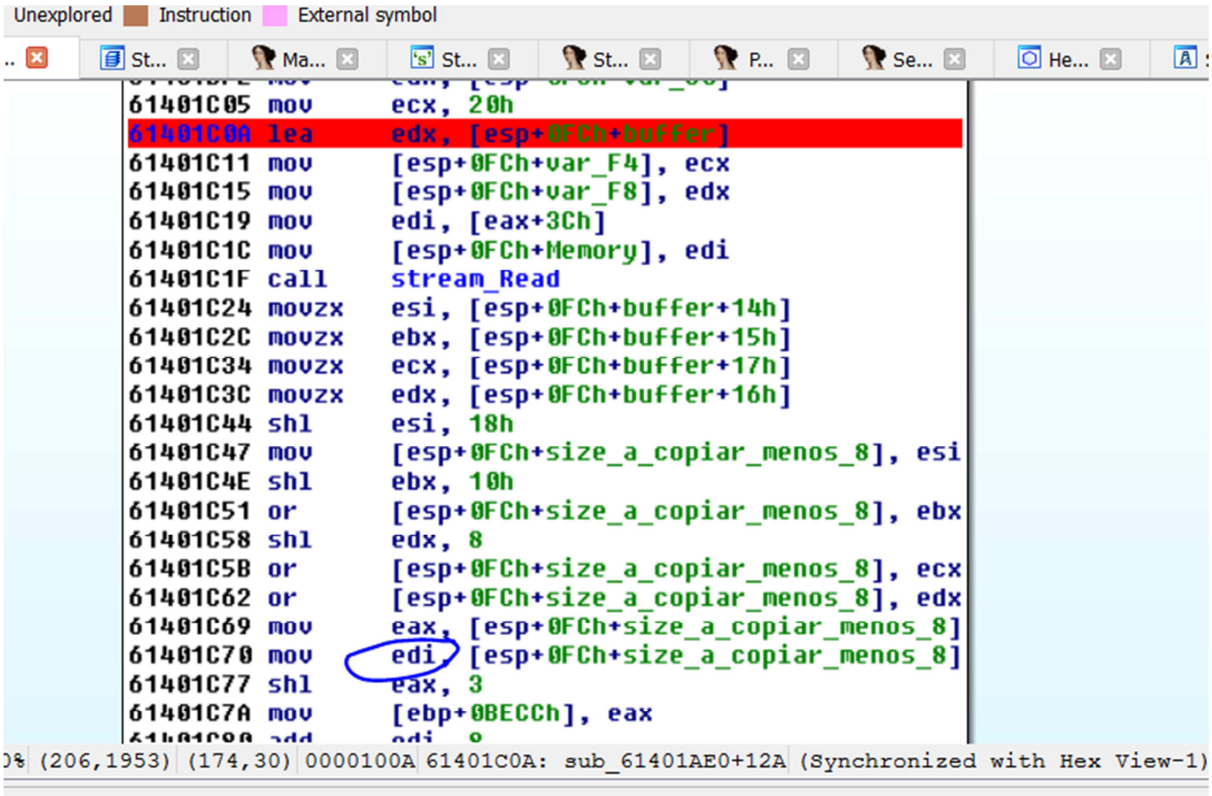


That EDI saved in size_a_copiar comes from another variable var_5c and it is added 8 to it. Rename it.

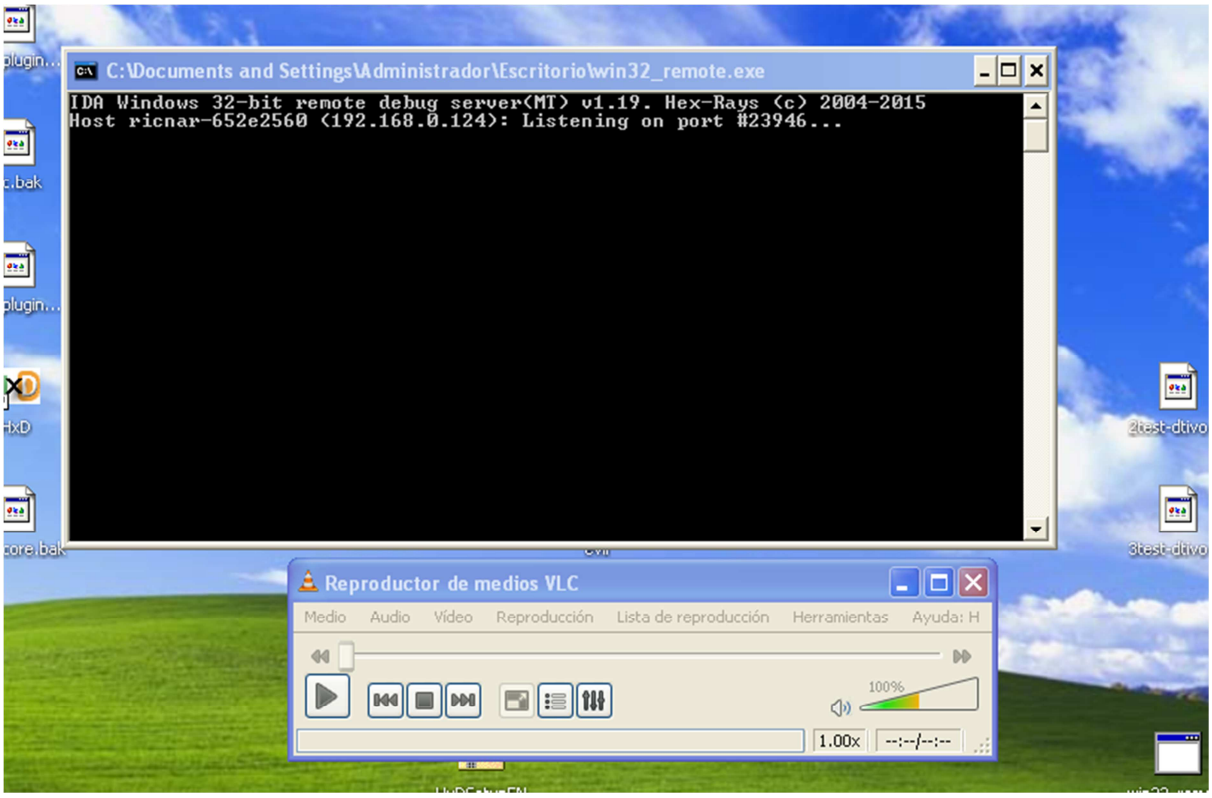


All comes from the first **CALL stream_Read**. It takes the bytes from the 14-15-16 and 17 positions and it will create it to leave the DWORD in size_a_copiar_menos_8 through SHL and OR surely.

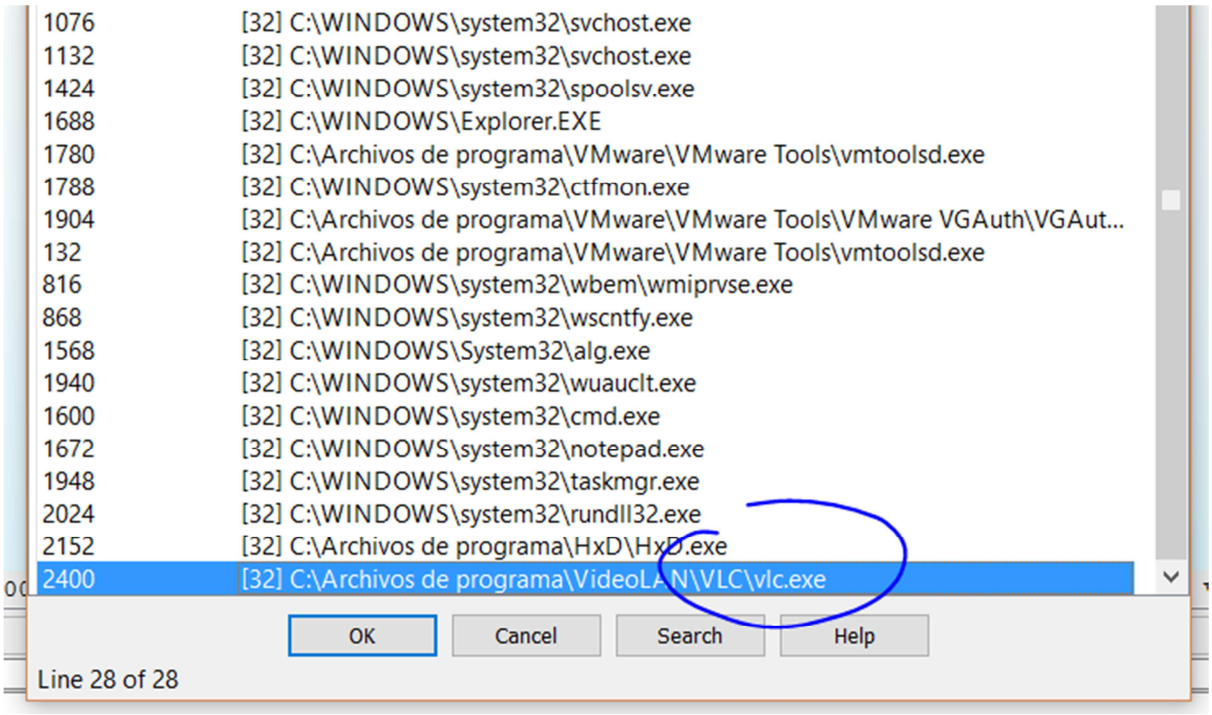
Set a breakpoint on the LEA.



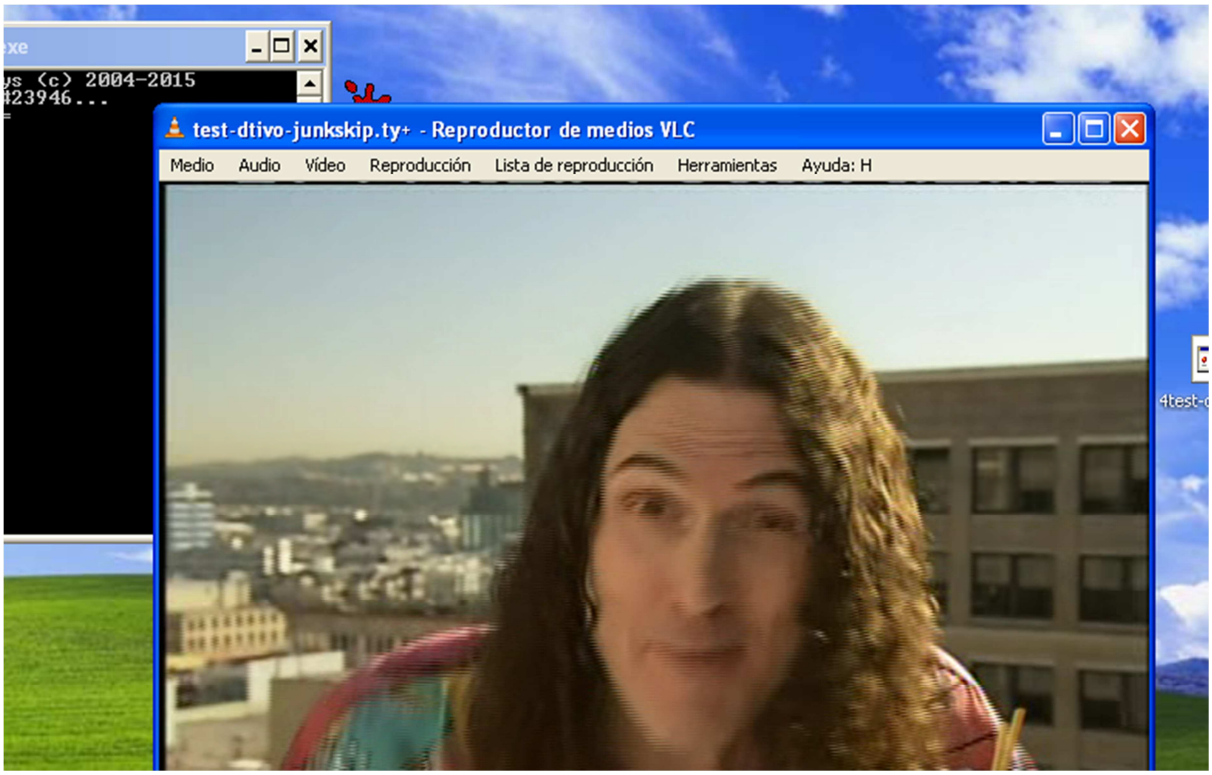
Attach the program and drag and drop the .ty file on the original VLC.



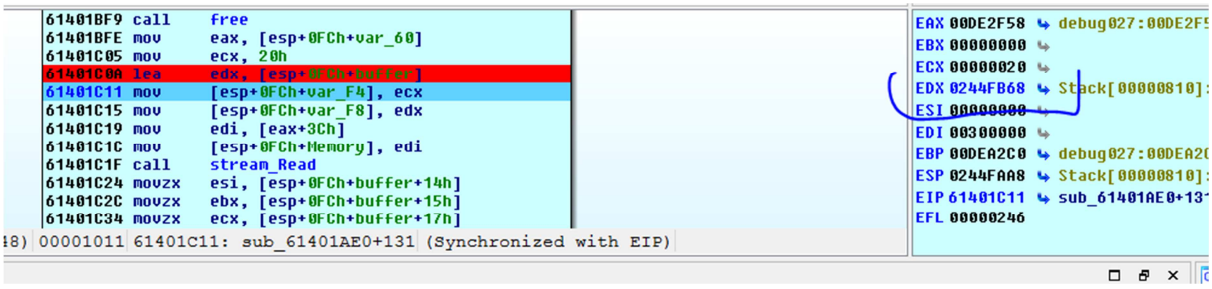
I have VLC and win32_remote.exe opened in my virtual machine.



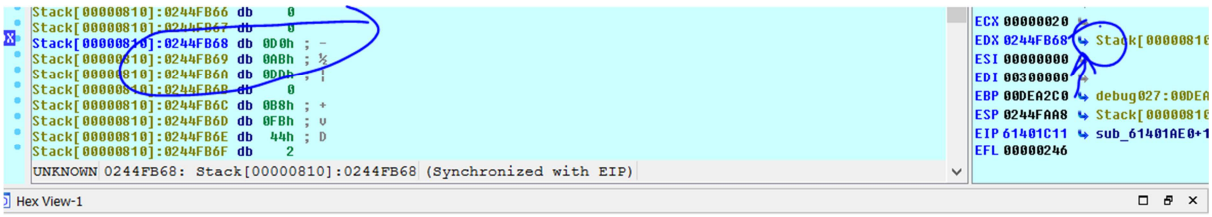
I attach it.



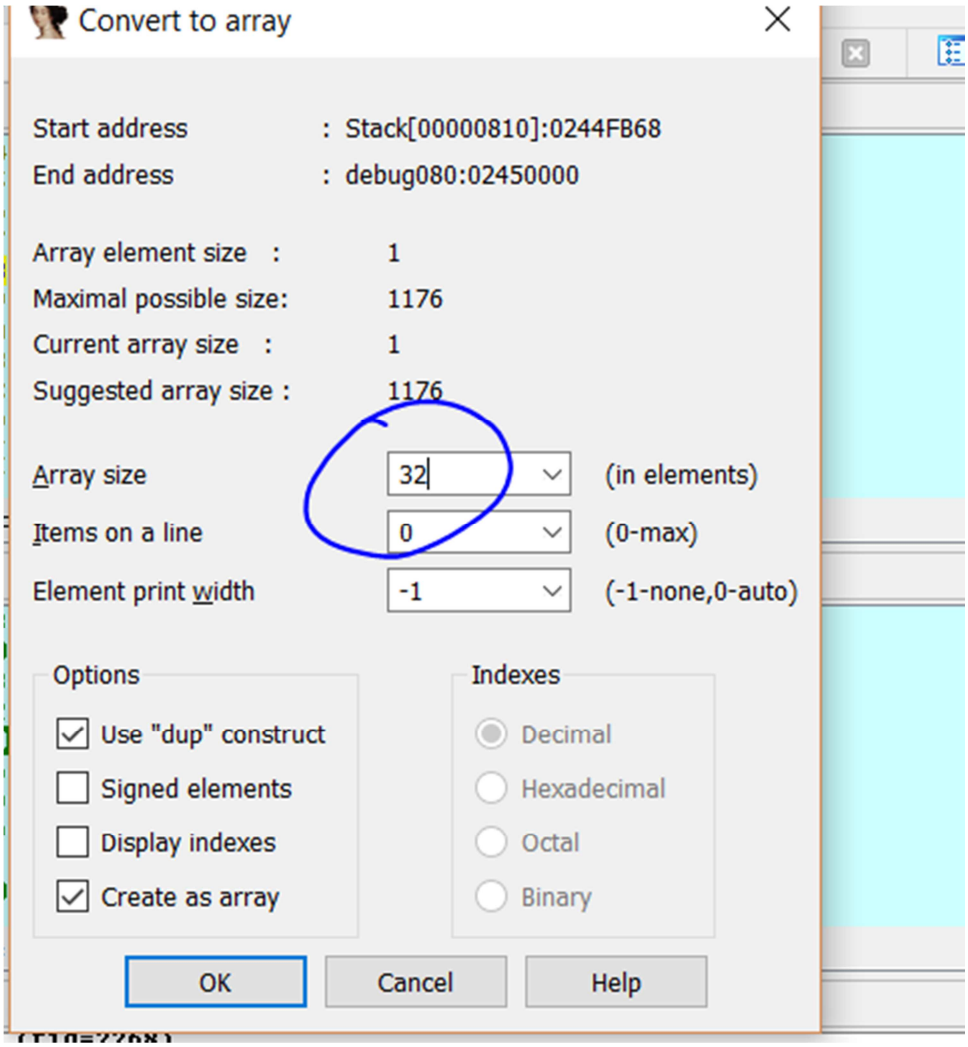
After playing it a while, it stops at the breakpoint.
I trace it with F7 and I see the buffer address in EDX.



Click on the little arrow next to EDX with the focus on the disassembly.



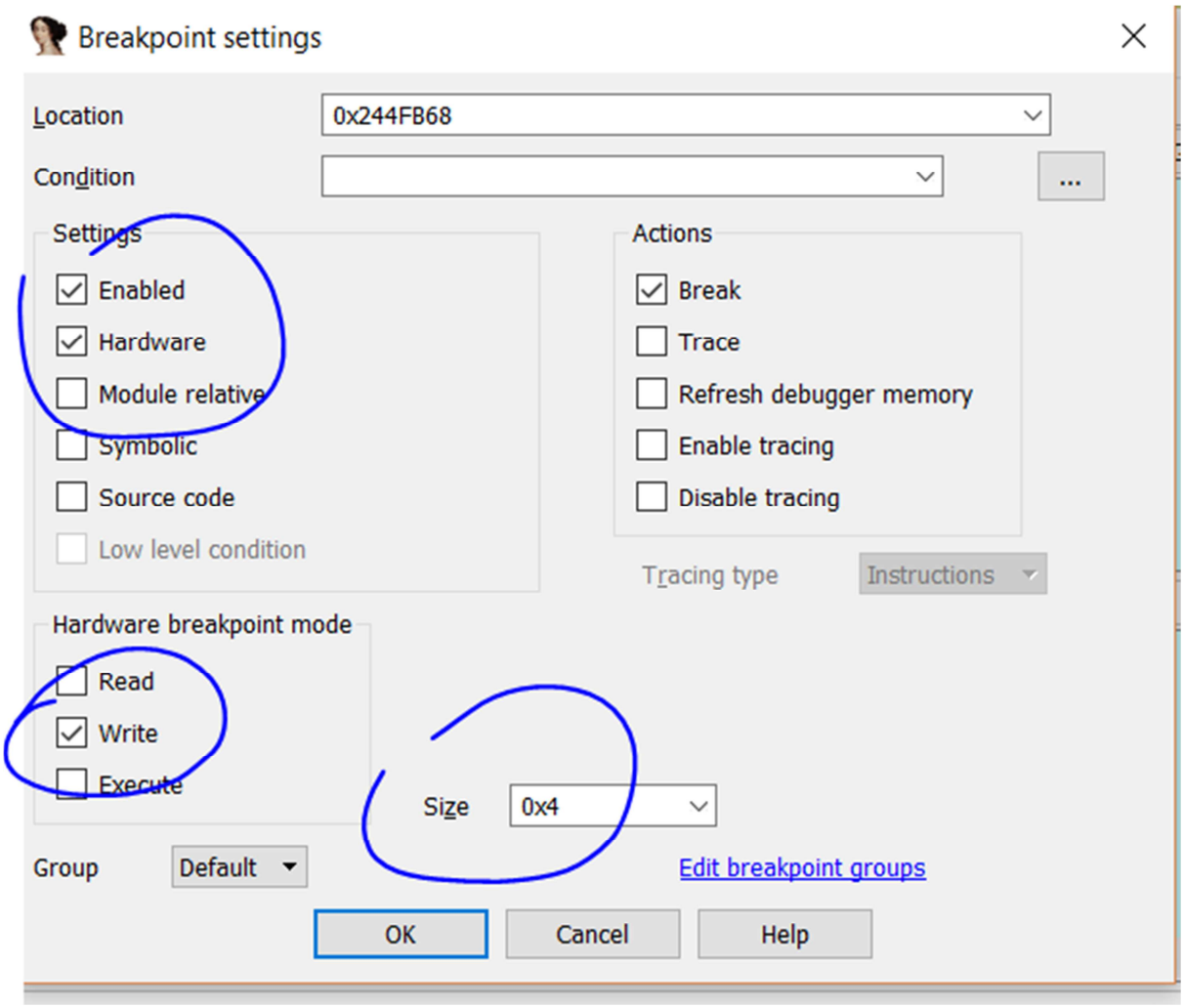
Now right click on it and create the 32-byte decimal array in memory.

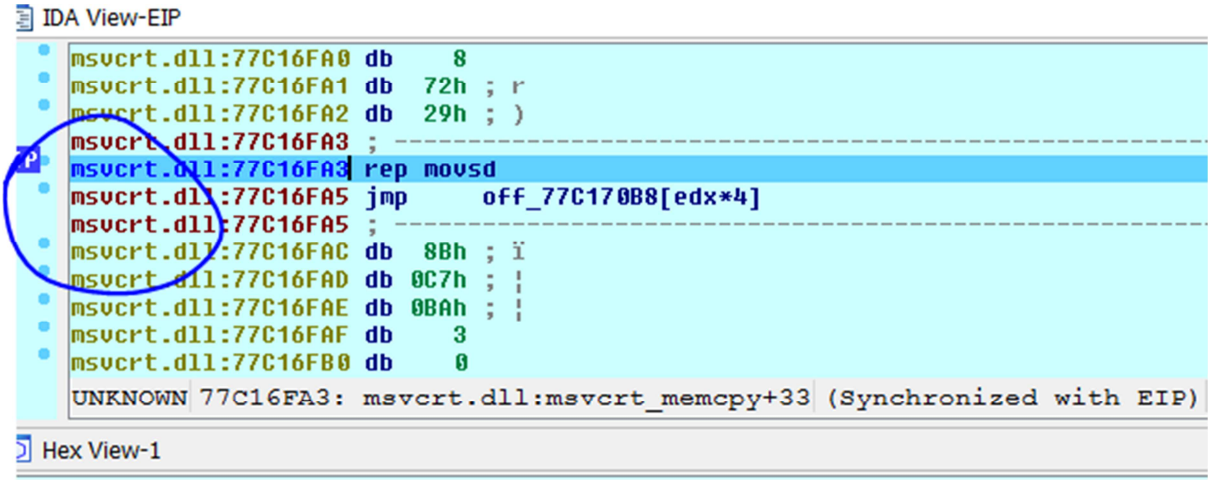


There, it is.

```
Stack[00000810]:0244FB66 db    0
Stack[00000810]:0244FB67 db    0
Stack[00000810]:0244FB68 db 0D0h, 0ABh, 0DDh, 0, 0B8h, 0FBh, 44h, 2, 0C0h, 0Bh, 10h, 0, 90h, 0ABh
Stack[00000810]:0244FB68 db 0DDh, 0, 65h, 13h, 45h, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0
Stack[00000810]:0244FB88 db    0
Stack[00000810]:0244FB89 db    0
```

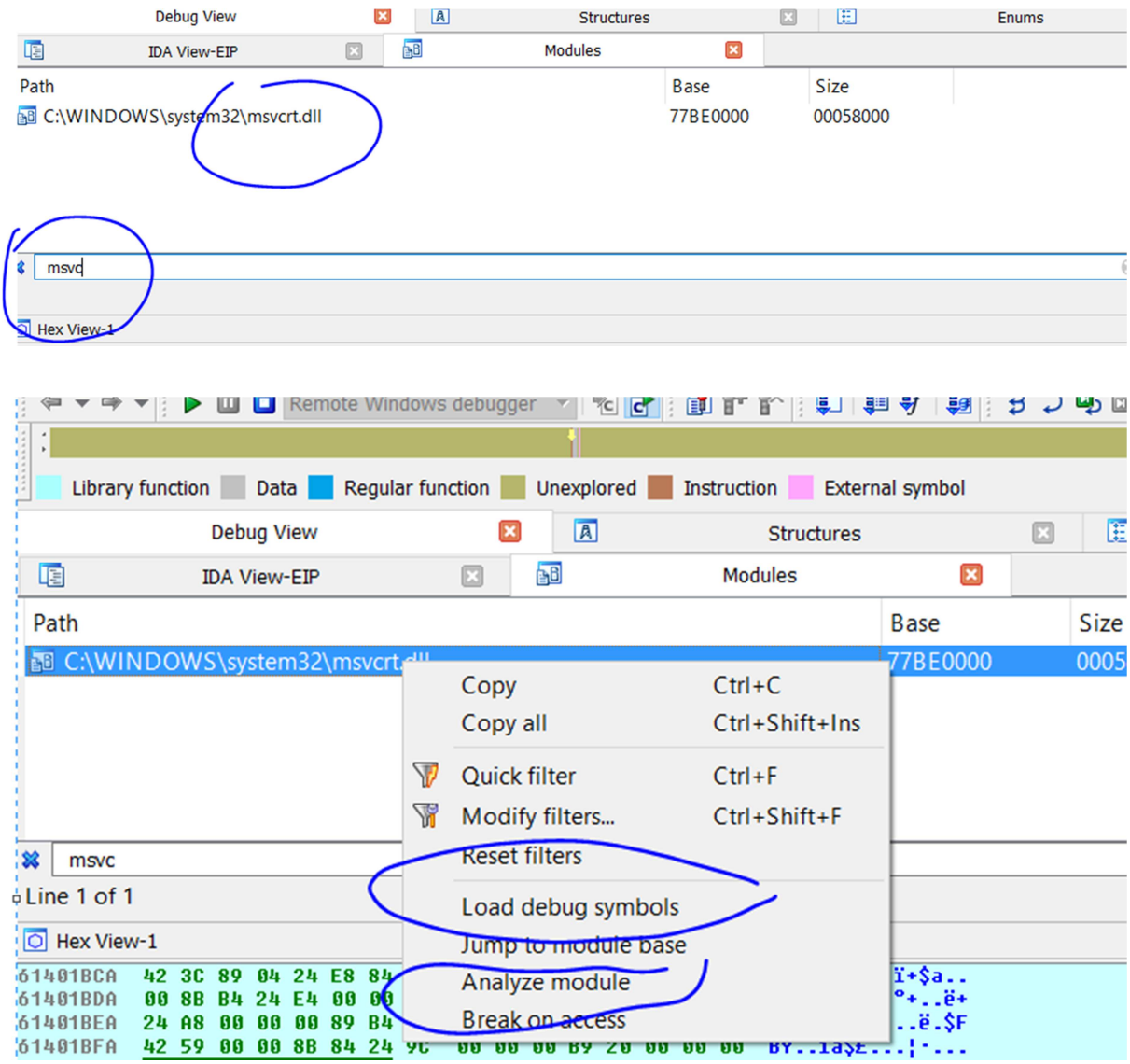
Set a breakpoint on write on the first dword of the buffer to see if when it stops it fills it inside stream_Read.



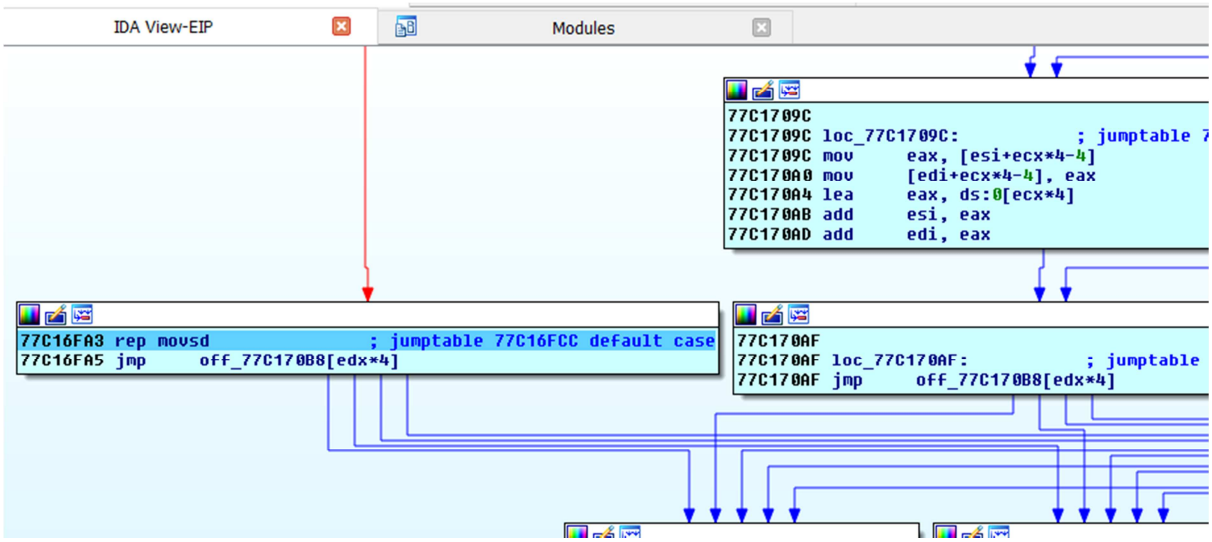


After pressing RUN, it stops on the msvcrt in a rep movsd copying it to the buffer.

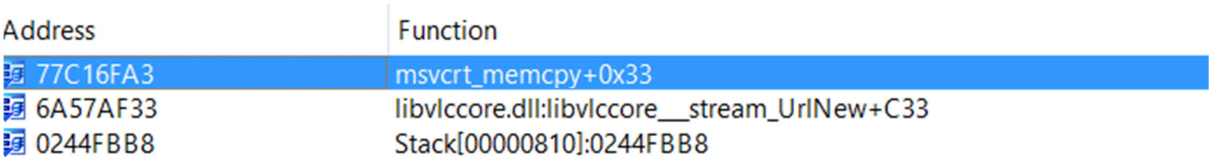
In the DEBUGGER- DEBUGGER WINDOWS-MODULE LIST menu, we see the module list and find msvcrt.dll there.



Right click - ANALIZE MODULE and when it finishes, select LOAD DEBUG SYMBOLS and the function looks better now. We can see it in that stack where we really are.

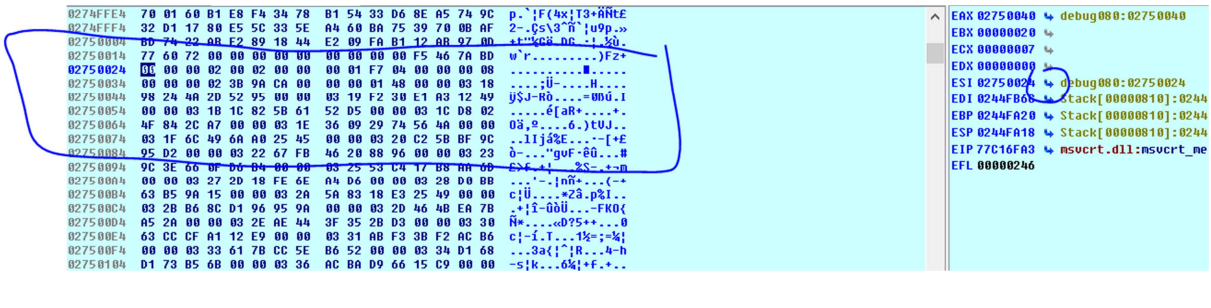


In DEBUGGER-DEBUGGER WINDOWS - STACK TRACE.

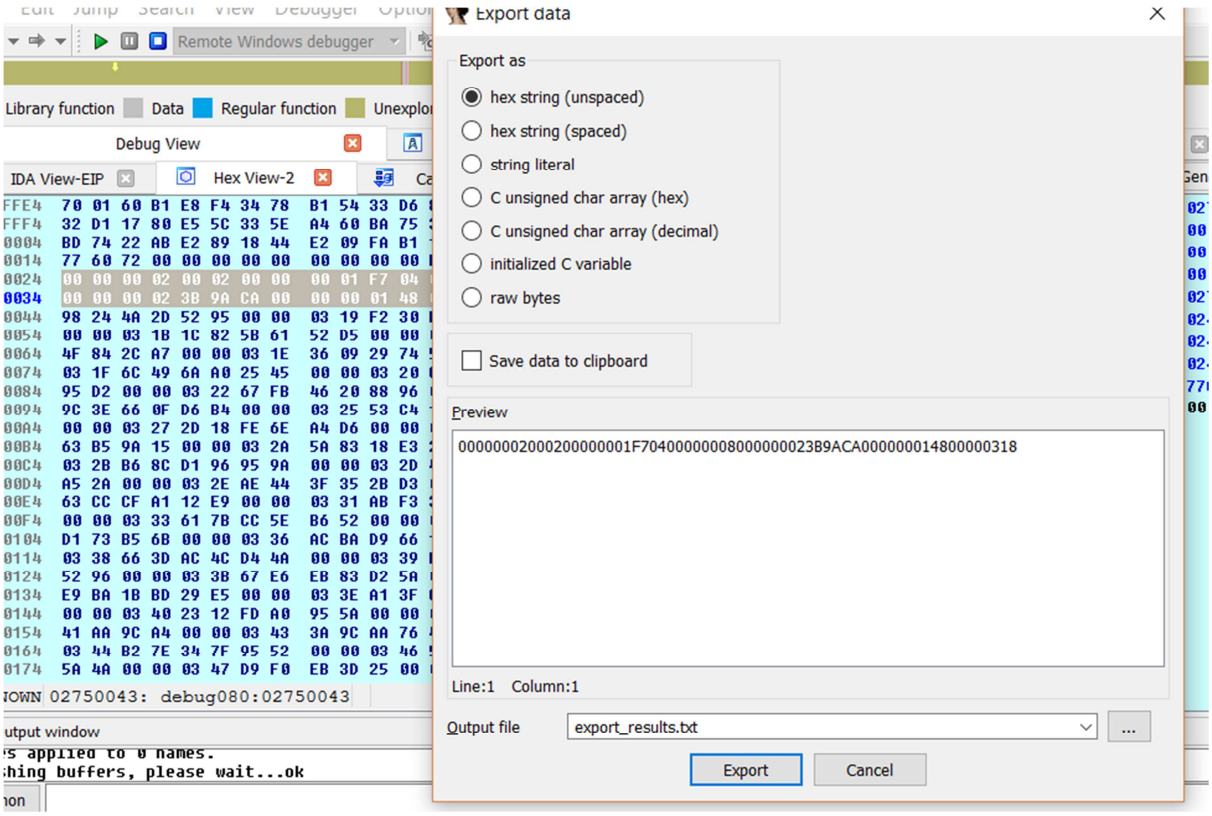


We are inside a memcpy and we see where it comes from.

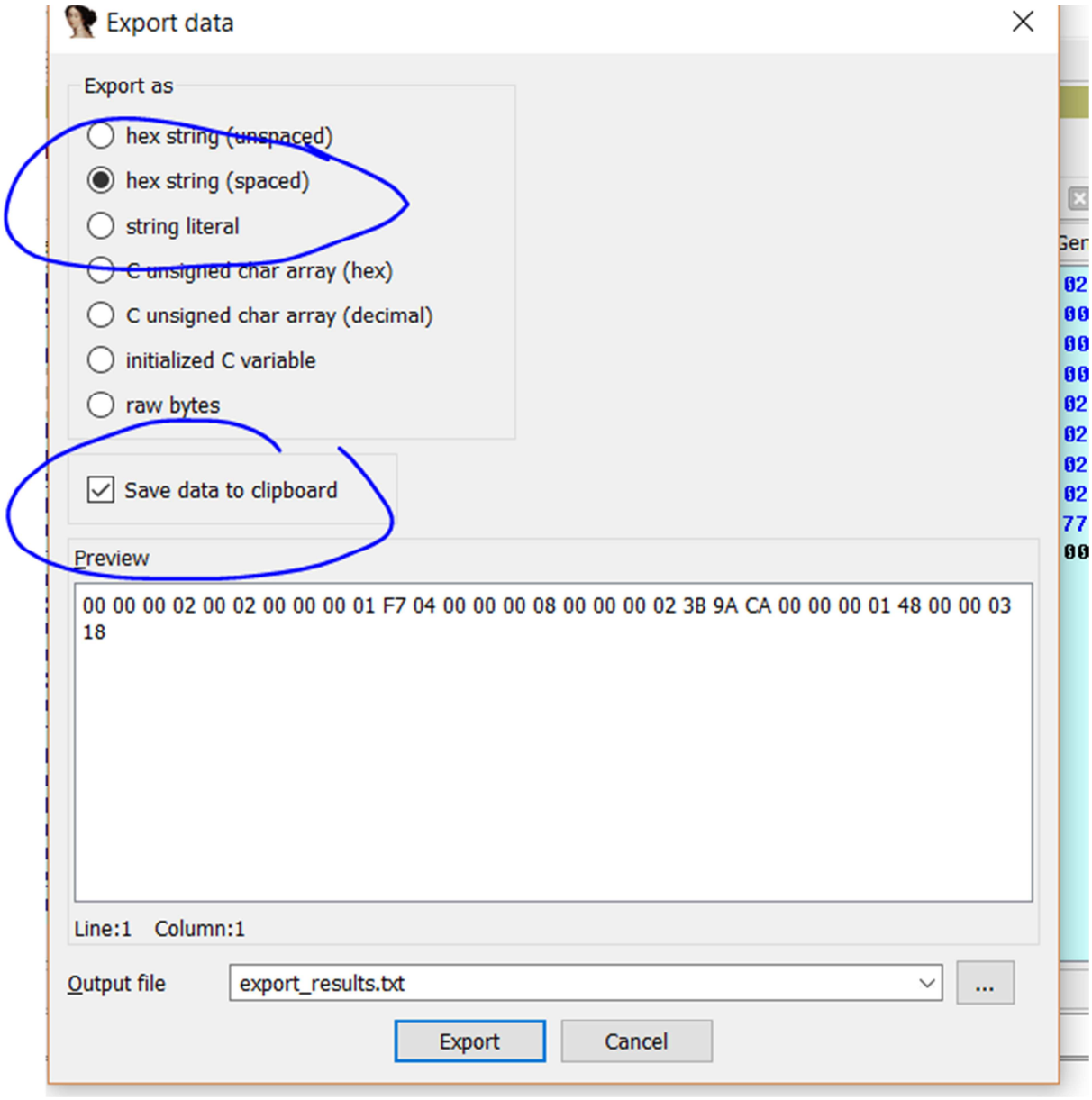
reps movs copies from ESI that is the source to EDI that is the destination. ECX is the number of dwords to copy. Click on the little arrow next to ESI to see where it points to, but having the focus on HEX DUMP to see there.



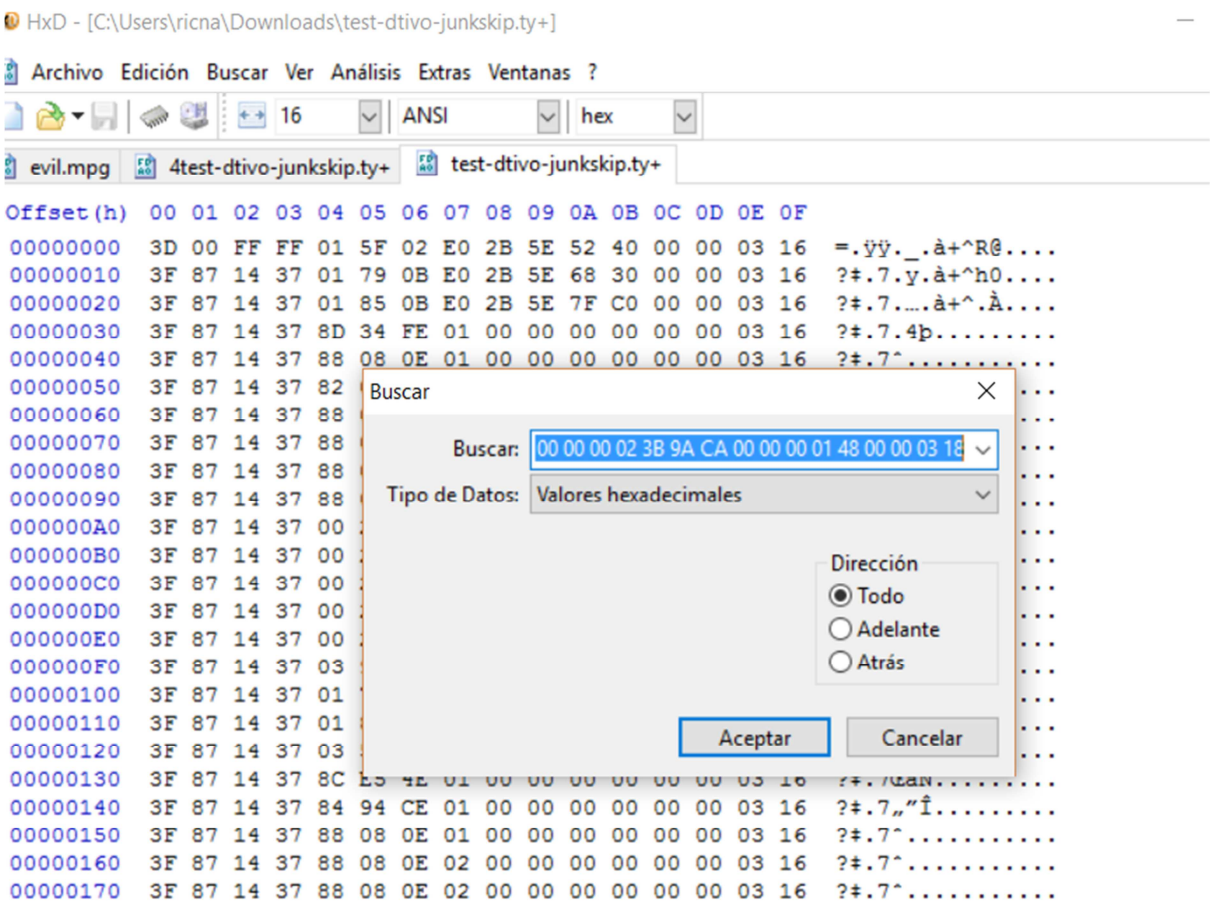
That’s what it copies in the buffer. Let’s open the .ty file in a hex editor and let’s do here the following.

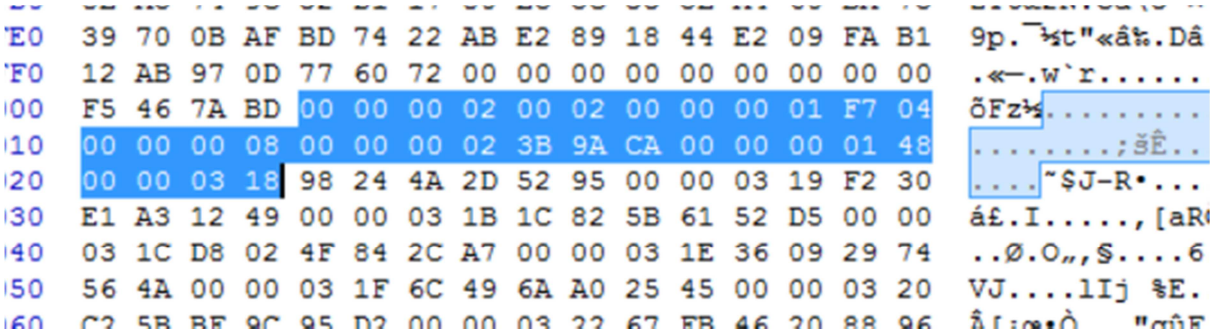


Mark 32 bytes and select EDIT-EXPORT DATA to copy the marked bytes in the format we wish.

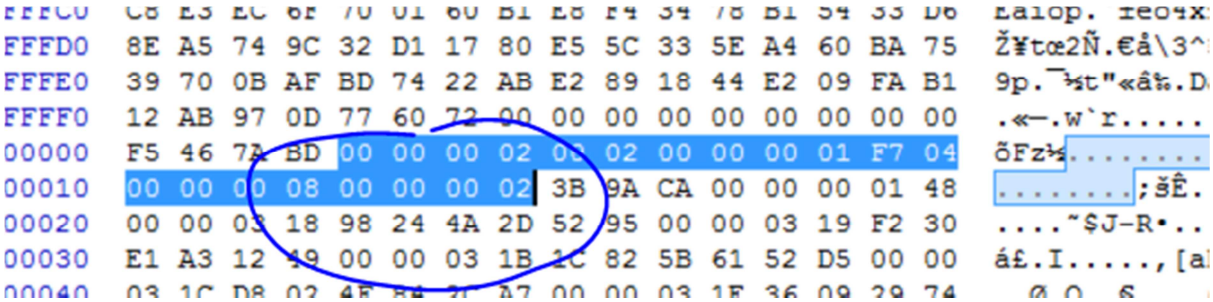


I think it will look better now.

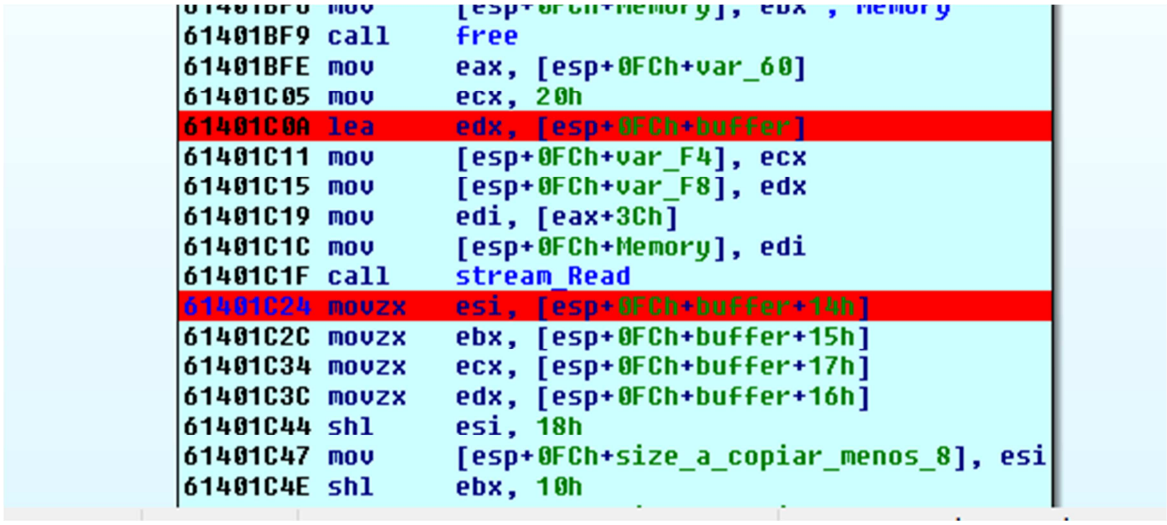




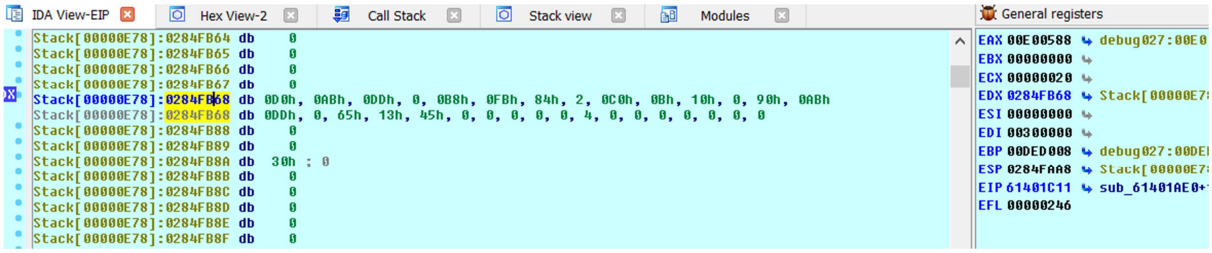
So, we found the bytes that it reads from the file. We know that the 14-15-16 and 17 are the ones that will create the value. It will add it 8, make a division and come to that value. Let's see what it is.



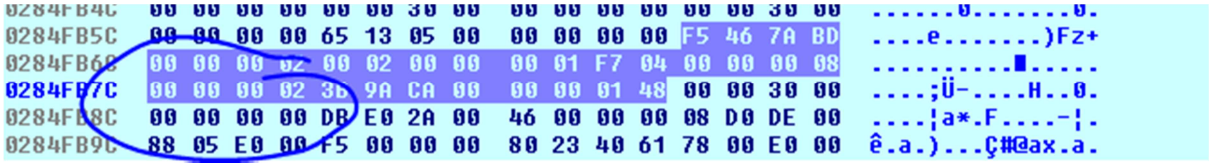
That's **00 00 00 02**. Set a breakpoint to stop when it comes from the stream_Read again in the dll libty_plugin.



I ran it again because the IP was down.

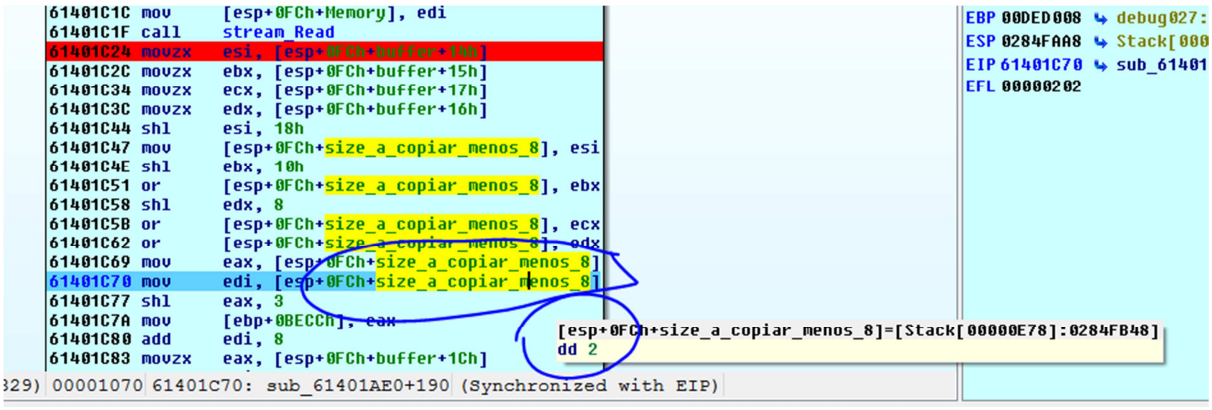


Now the buffer is here.

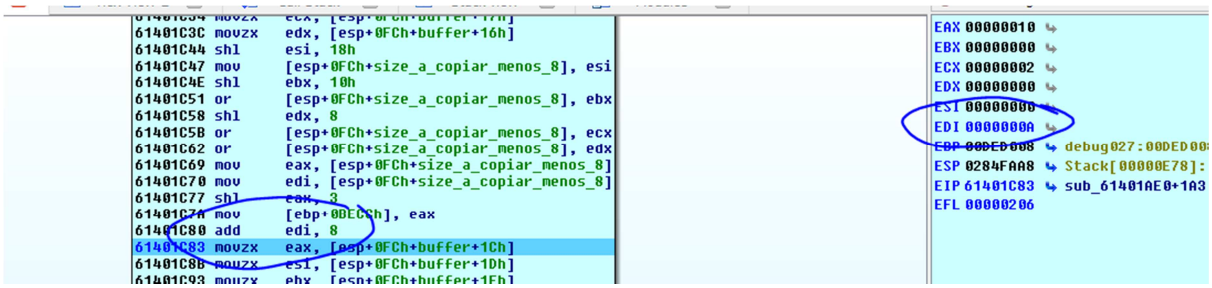


I can put it in hex dump and add 0x14 to the buffer address and I see that it is the 00 00 00 02 we had seen in the file.

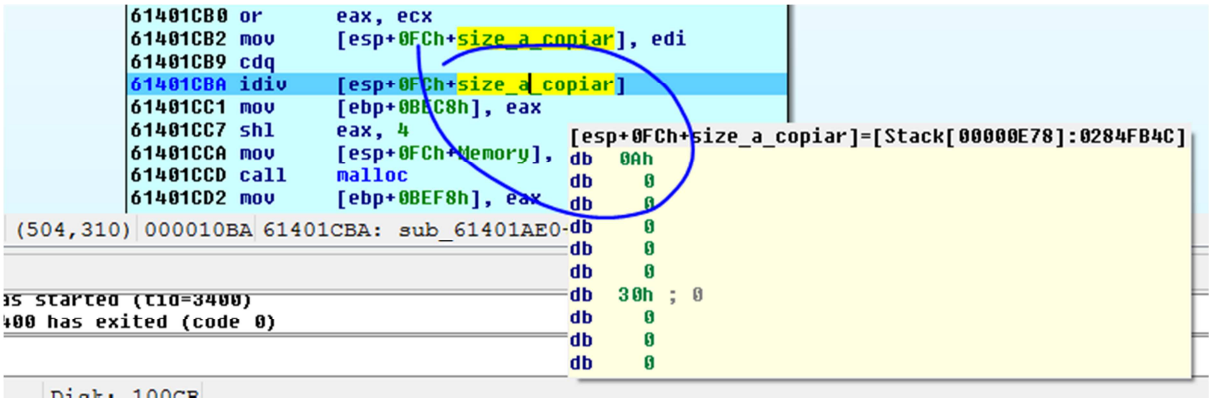
Let's trace to see what it does.



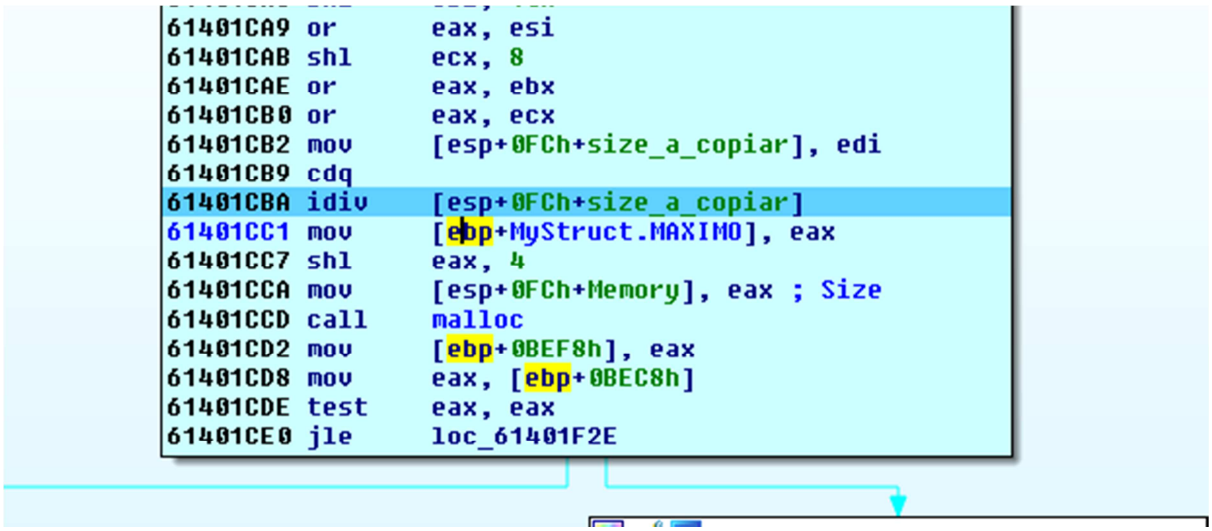
All that is to create the DWORD and move it to EDI. Then, it should add 8.



And more ahead, it will use an IDIV. Go there.

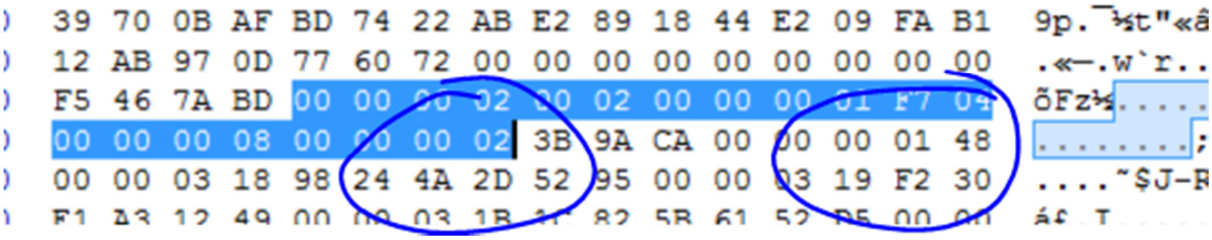


There, it is the 0x0a that came from the 0x02 read by the file and it added 8. The result is 0xa.

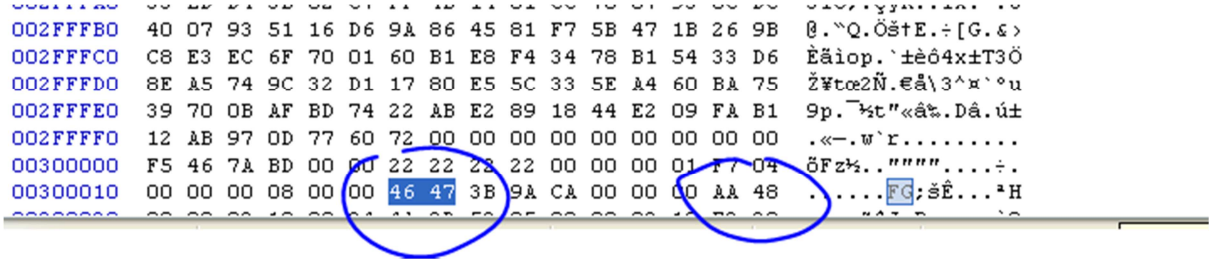


IDIV is the signed division. It will divide EDX:EAX by the size to copy which won't be modified. The problem is that if I increase the size to copy much, the division will be 0 and that is the value that goes to the MALLOC after multiplying it by 16. So, we must handle this division well for the result not to be 0.

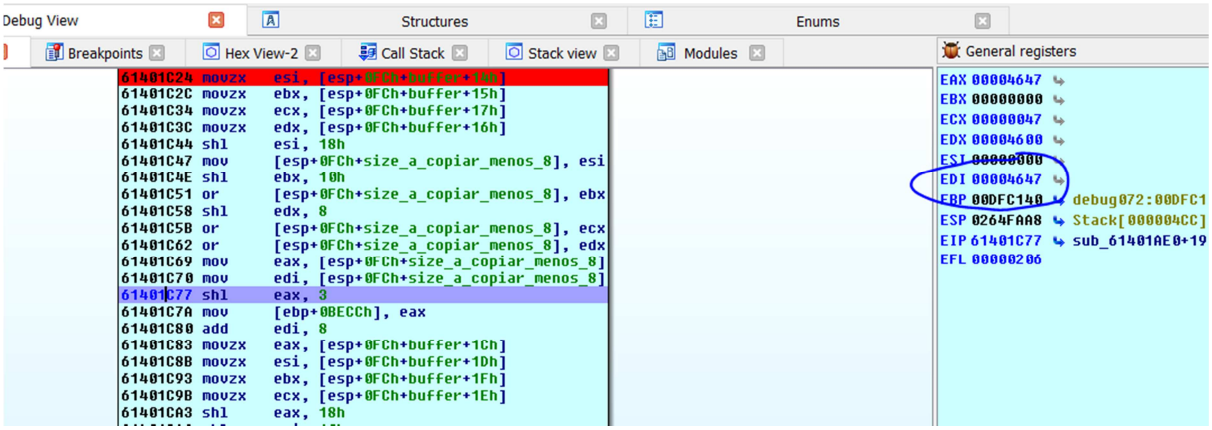
EDX:EAX is 00000000:00000148 and it is divided by 0A. If we see in the file, the 0x148 is close to 00000002.



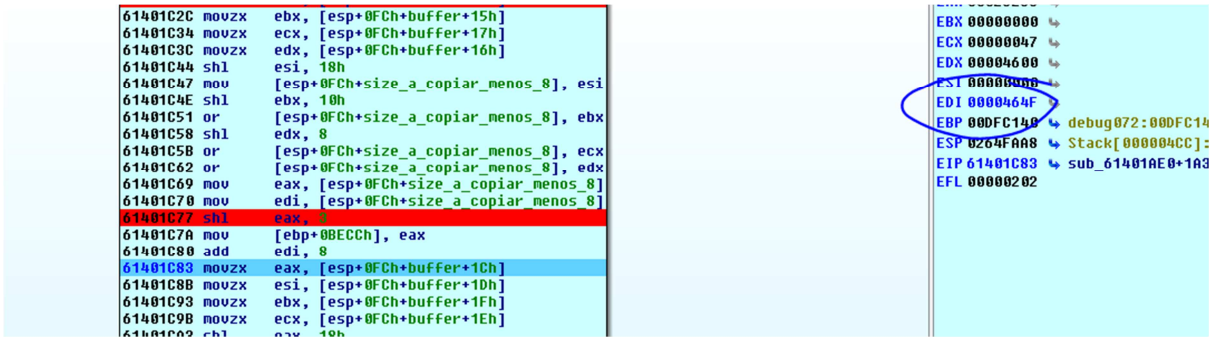
If I increase it to 02, I will also have to increase the 0x148 for the division not to be 0. I will do it.



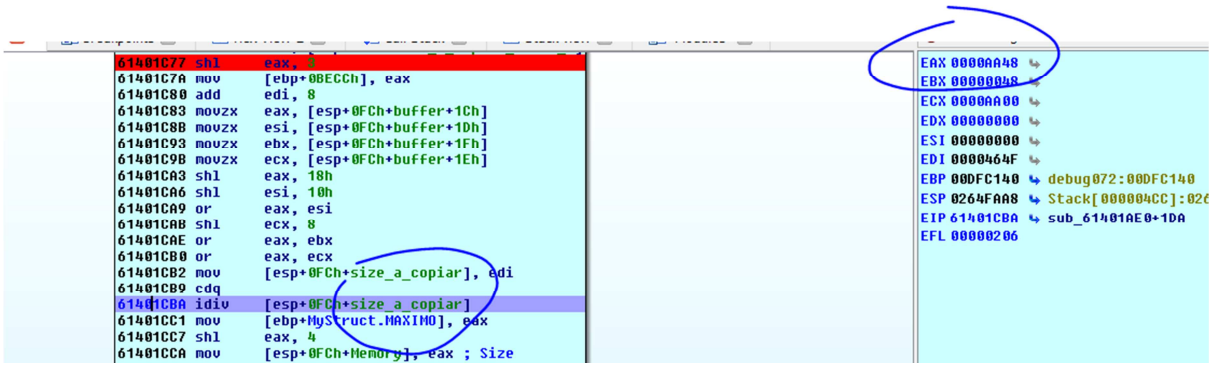
That way, we will see if it comes to the stream_Read with a size greater than 8 to overflow the buffer. I will run it again with this modified file.



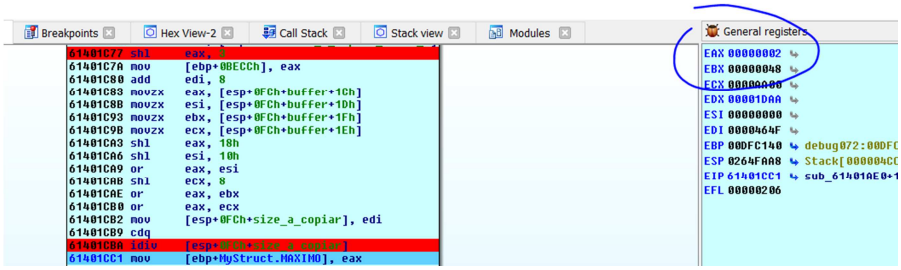
There, it creates the 0x4647 in EDI. Then, it will add it 8.



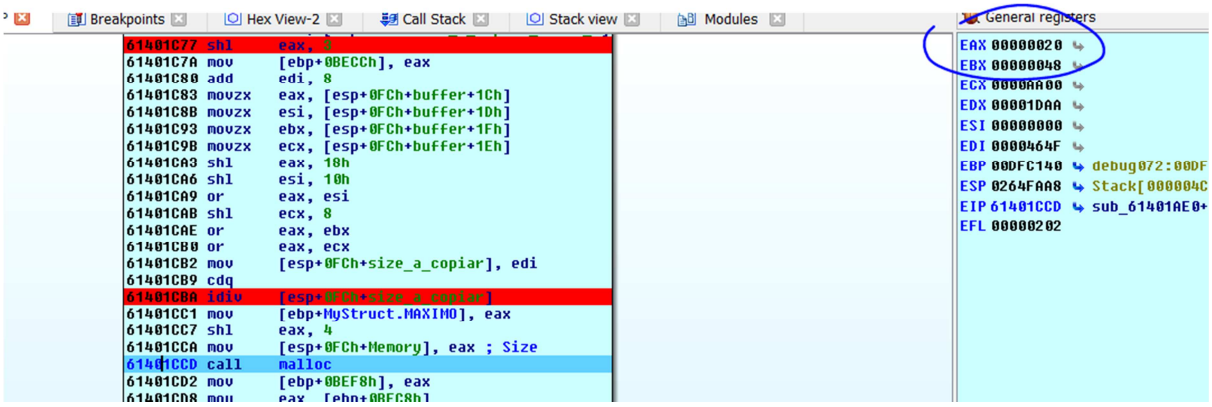
Then, the IDIV 0000000:AA48 divided 0x464f.



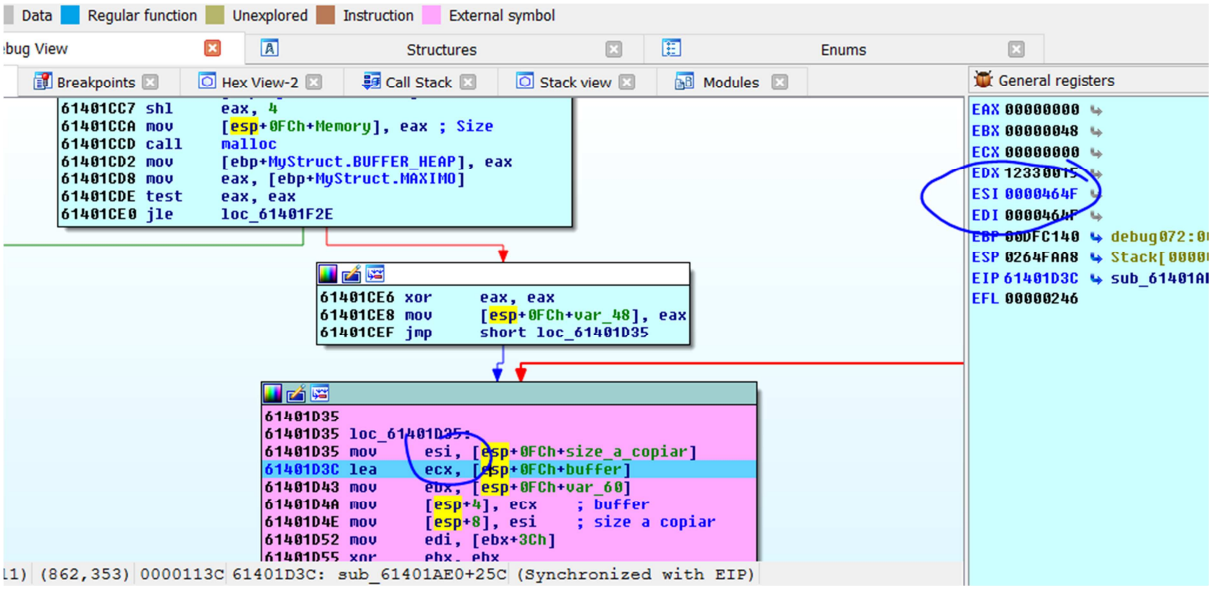
The division result is in EAX and it is 2.



That is the max value it will multiply by 16 and call malloc. As we are not exploiting the heap overflow, while it allocates, it will be fine.

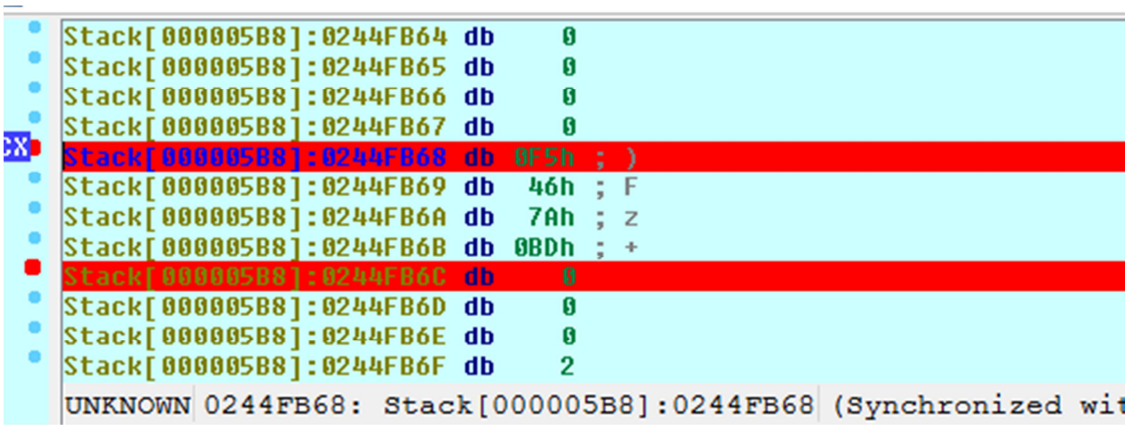


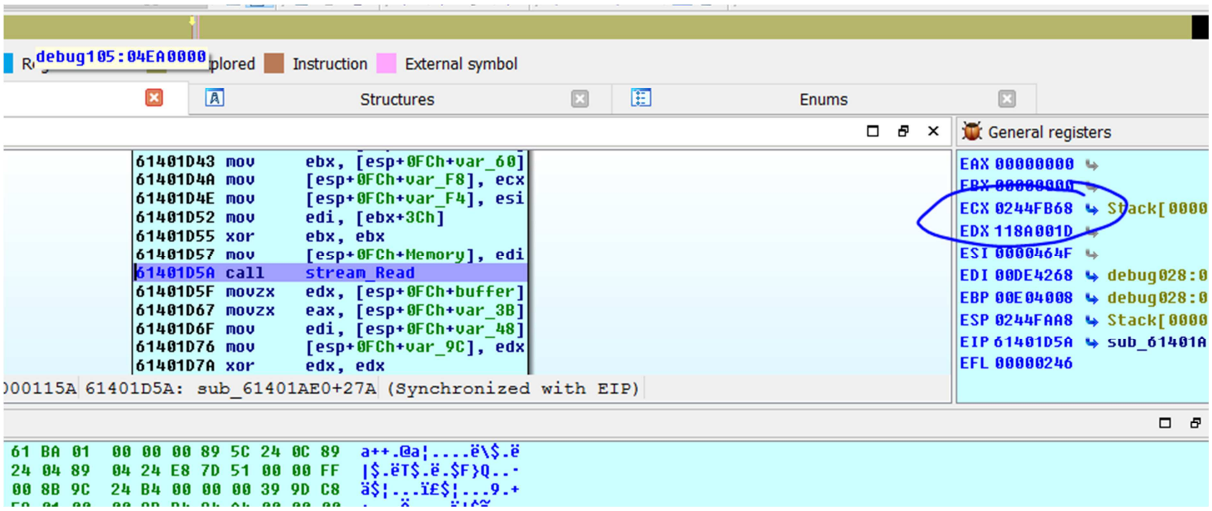
So, it calls malloc with the size 0x20. It will allocate it without problems.



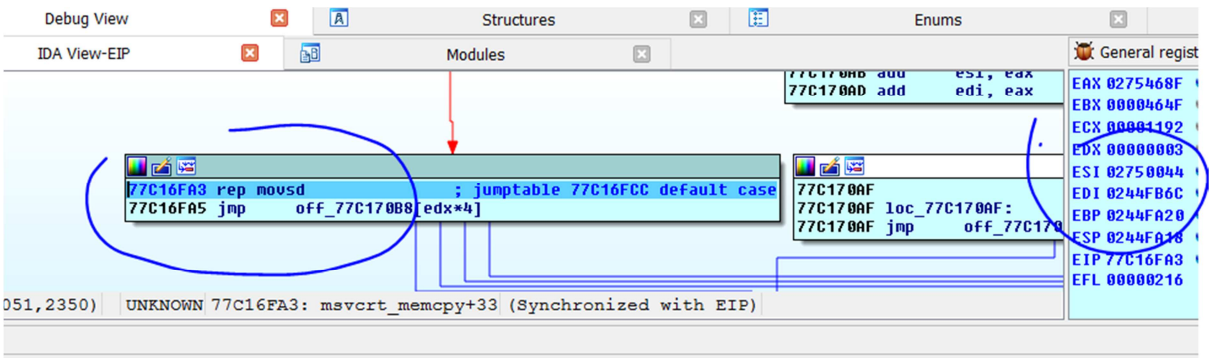
It goes to the block where the vulnerability is, with the size_a_copiar 0x464f that is obviously greater than 8. In the patched version, it will ignore it and avoid overflow.

I set breakpoints in the buffer. ECX points to it. Go there and set a hardware breakpoint on read write to stop when it starts filling the buffer.

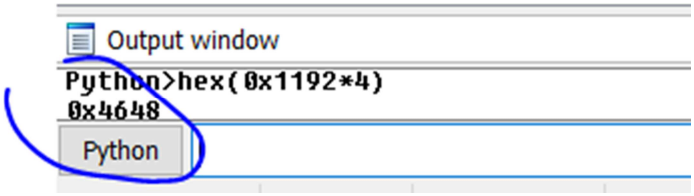


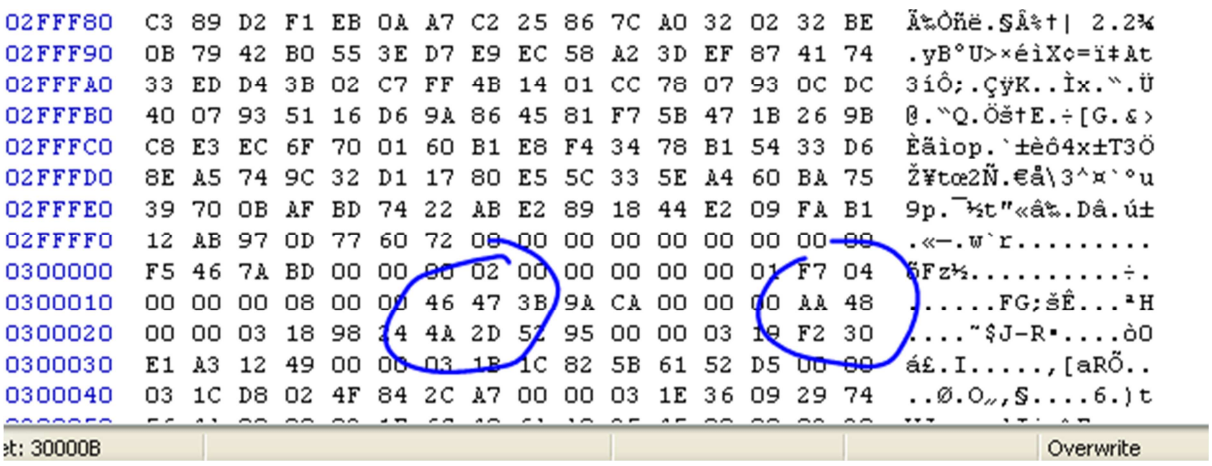


F9 to stop when it copies in the buffer.



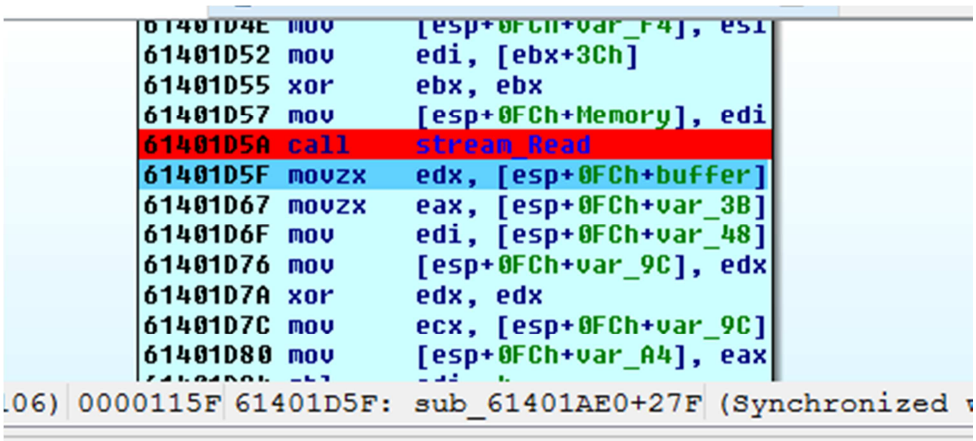
ECX is copying 1192 dwords because rep movsd means REPEAT MOV DWORDS. So, the total bytes written in a 32-byte buffer is 0x1192 x4 or 0x4648 that comes out of rounding the 0x4647 I put in the file.



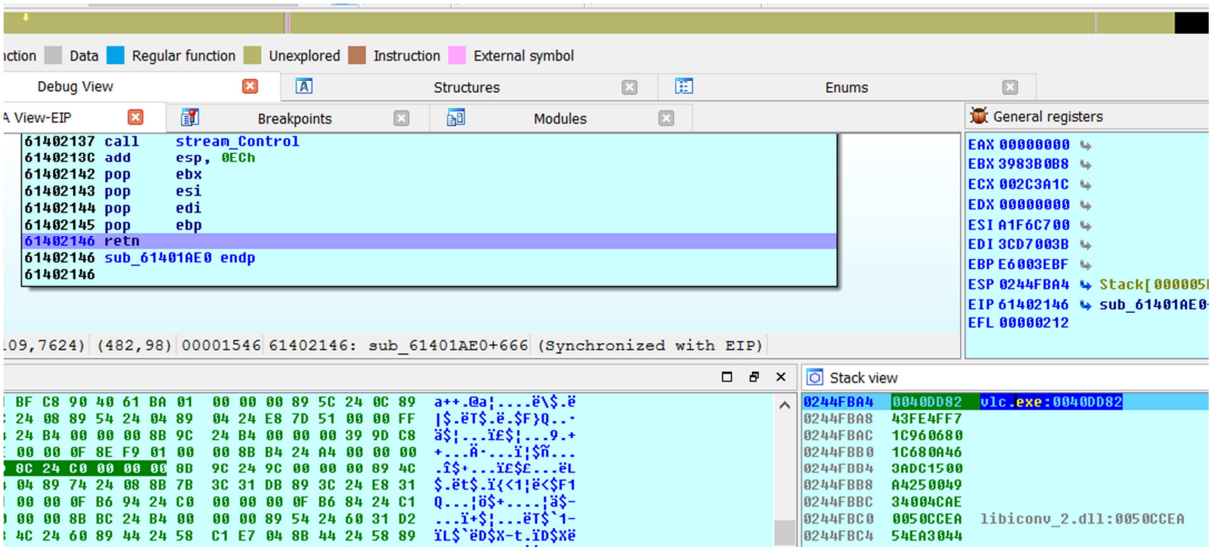


I will go to the RET because it could overwrite it because of the size it has to copy to the buffer.

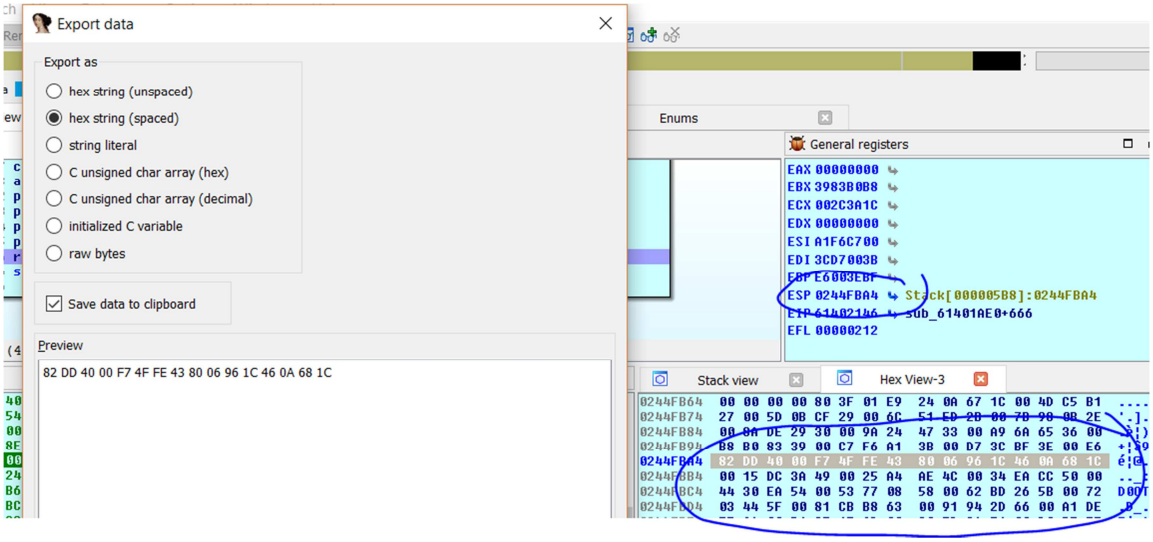
Select RUN TILL RETURN or CTRL+F7 and it comes back to the main function where the buffer is located when it comes to the RET, it should crash.



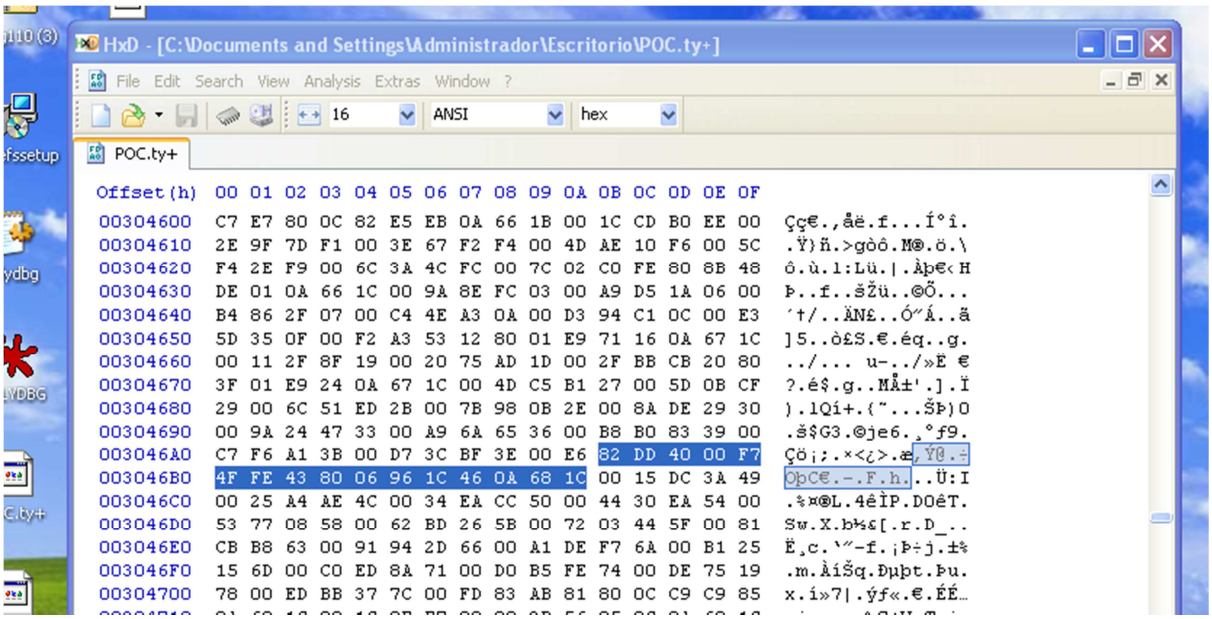
Set a breakpoint in the function RET and disable the other BPs.



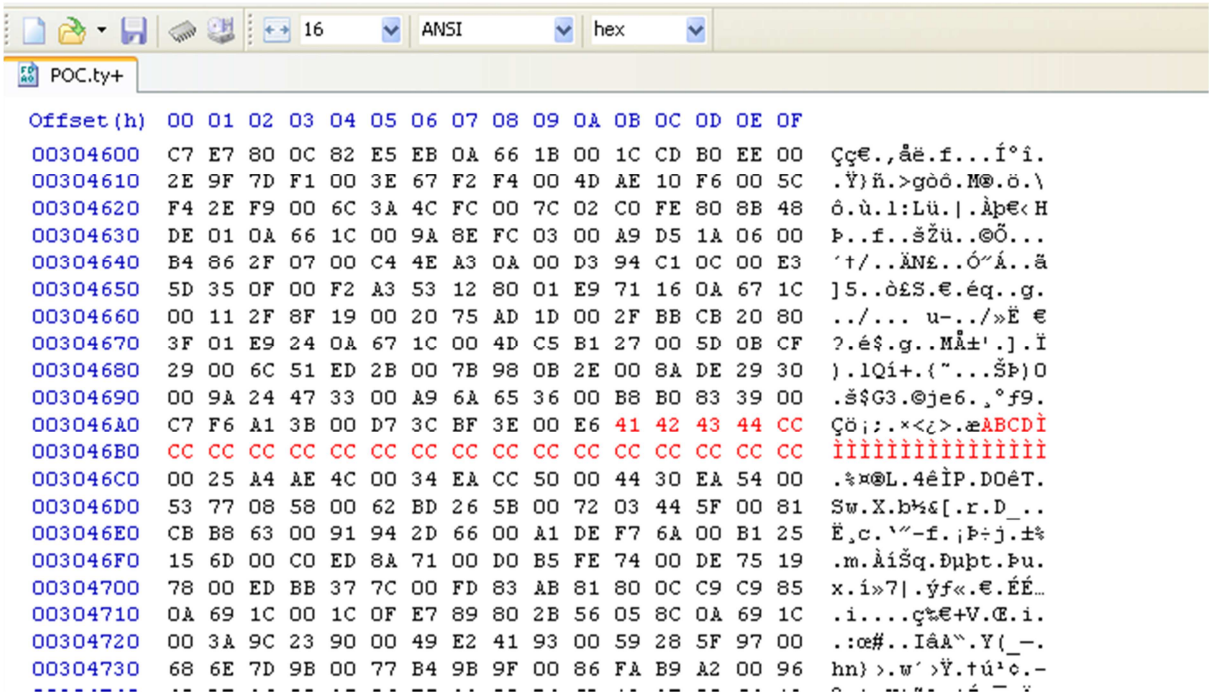
The stack is destroyed because I overwrote all there. Go to HEX VIEW and press the little arrow next to ESP.



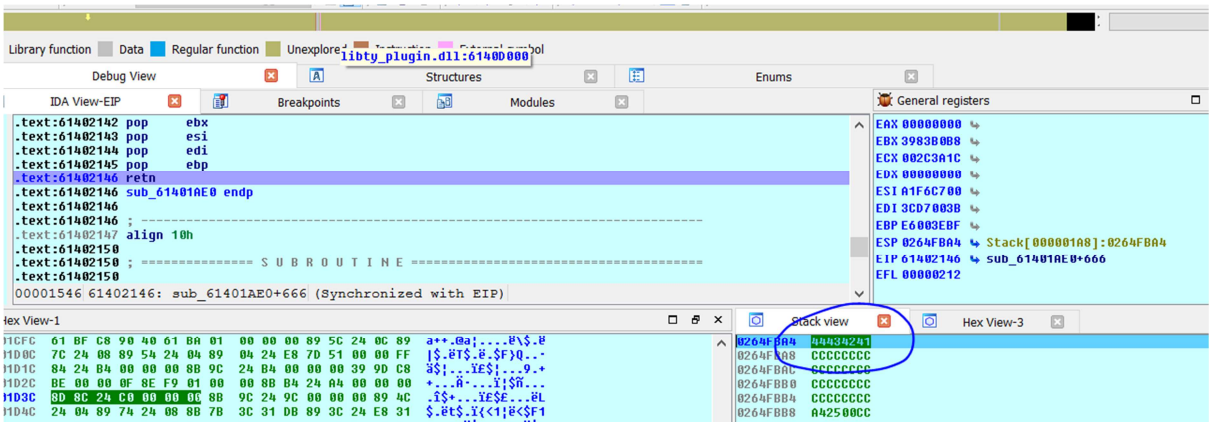
I find that string in the file.



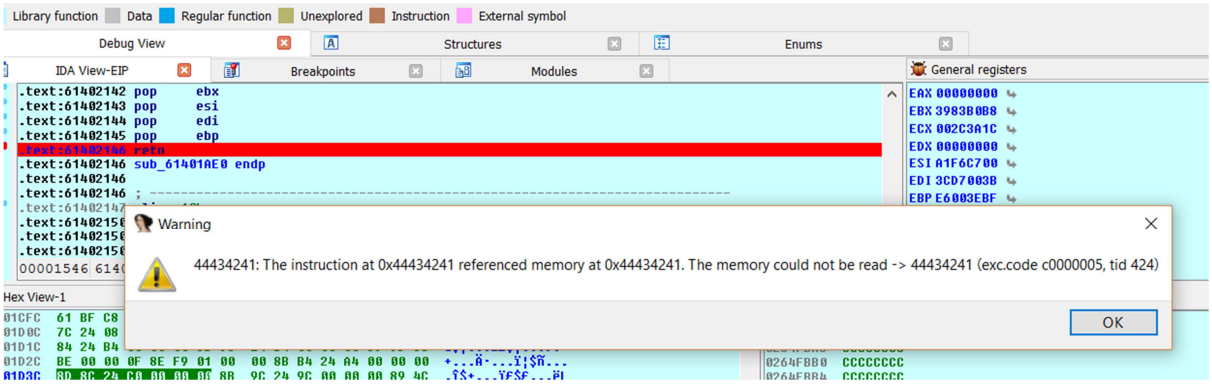
Those are the values where it will jump because we overwrote the return address.
I'll change it by:

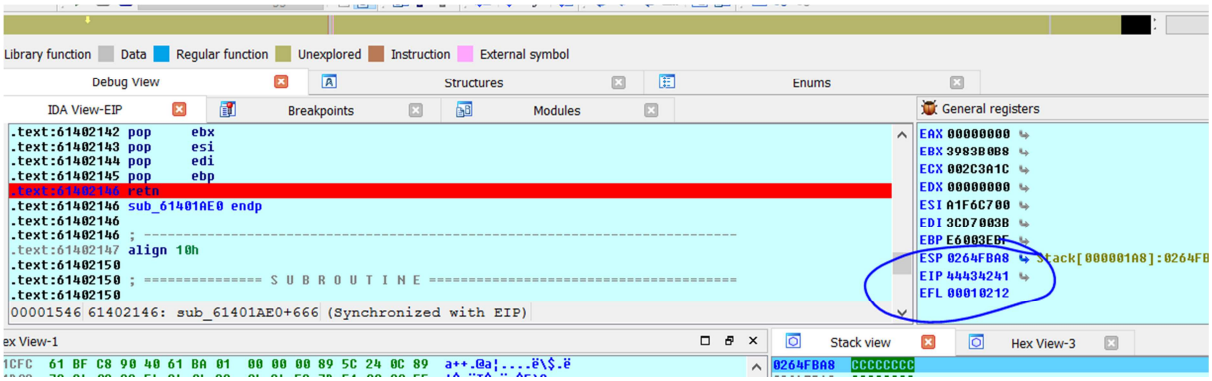


Now, I run it with this modified file.



Ready. Now, if I run it, I take EIP control that is the target of a POC. Some POCs don't do that. They just break the program.





If all is OK, it could be exploited. We'll see how to continue with the exploitation of this example more ahead. In the next parts, we'll continue with the more theory and some easy examples programmed by me for you to practice. I already worked so much. ☺

The file extension must be **.ty**. If we change it, it doesn't go to the vulnerable part.

Ricardo Narvaja

Translated by: @IvinsonCLS